

Original Article

Transforming UML Models to MongoDB Schemas Using Model-Driven Architecture and JavaScript

Hamza Natek¹, Aziz Srail², Abdelmounaim Badaoui³, Fatima Guerouate⁴

^{1,3,4}LASTIMI Laboratory, Superior School of Technologies of Sale, Mohammadia School of Engineering, Mohamed V University, Rabat, Morocco.

²ERCI2A, FSTH, Abdelmalek Essaadi University, Tetouan, Morocco.

¹Corresponding Author : hamzanatek@gmail.com

Received: 01 July 2024

Revised: 18 March 2025

Accepted: 20 March 2025

Published: 26 April 2025

Abstract - Relational Database Management Systems (RDBMS) have served the data management needs of organizations over the past decades with strong solutions for solving structured data. But with the advent of Big Data, complexities in dealing with high volumes, high variety and high velocity of data in production, these systems have been challenged. NoSQL databases are seen as a potential answer, hosting flexible schema arrangements and extensible execution. Conversion of structured UML models to dynamic NoSQL schemas is a very resource-hungry process. This article will introduce an approach to automate this transformation in JavaScript for MongoDB using Model-Driven Architecture (MDA). We cover upfront technical requirements, a safe connection to MongoDB, transforming UML models to JSON objects, and the M2M and M2T transformation to generate and validate MongoDB schemas. To ensure that the generated schemas were correct and had a high degree of fidelity, I validated them with MongoDB Atlas. This automation of the process not only speeds up database design but also makes software development more agile. The approach was applied to UML-based descriptions of school or college types, showing its suitability and correctness in producing the database schema that accurately reflects the design stipulated in the UML model.

Keywords - Model-Driven Architecture (MDA), UML, NoSQL Transformation, JavaScript, Schema Generation.

1. Introduction

Traditional relational database systems (RDBMS) have long been the foundation of data management, providing reliable mechanisms for structuring, storing, and querying data through SQL. These are structured systems usually defined using schemas, support for ACID (Atomicity, Consistency, Isolation, Durability) transactions, and guarantee the integrity and reliability of the data being processed. However, with the growth of Big Data, these applications are often insufficient to manage data with high volume, variety, and velocity. One such solution is NoSQL databases, which provide a way to design a schema without feeling confined to it, as well as the means to scale performance as the size of the dataset or the number of concurrent users grows. However, moving from structured data models such as Uniform Modelling Language (UML) is not so smooth when it comes to flexible NoSQL databases. This is where Model Driven Architecture (MDA) plays its most critical role. The Model-Driven Architecture (MDA) is a way of the framework for software design and implementation which utilizes models as the primary artifacts. As we can see, MDA can help solve the engine of transformation, and mapping those UML models into NoSQL database schema can be automated [1-3], meaning reducing the manual part as well as making it

systematic and consistent. This approach has been used in many fields, including web service and its specification generation on [4] web frameworks [5, 6], blockchain and IoT [7-9], AI [10], and mobile application development [11], databases generation. In this article, we introduce a novel mechanism to implement this transformation by JavaScript, considering MongoDB as a NoSQL database. First, we need to set up the tech prerequisites for transformation. The second step is to figure out the tools from code, like Mongoose, XML, JSON Parser, etc. In the transformation script, we connect to the MongoDB NoSQL database with the right credentials. We next analyze the UML models and translate them into exploitable JSON objects.

Then, a series of Model-to-Model (M2M) transformations are used to produce intermediate NoSQL-specific schemas. Finally, Model-to-Text (M2T) transformations generate MongoDB schema definitions and the required deployment scripts. MongoDB Atlas serves as the case study for this method, where a second user stores the transformed objects and asserts that the schemas generated are faithful and have no biases. Training on this approach runs up to October 2023. It provides database design in an automated process that speeds up the development time while also



increasing the flexibility of software development and making it easy to adapt to changes. The presented approach was validated by implementing a transformation for UML classes representing educational generalizations. The resulting MongoDB schemas were thus used to store sample objects in a MongoDB instance to prove the method in practice. Our method's transformation results show that the entire database structure was generated correctly while faithfully preserving the structure described in the UML model.

1.1. Data Privacy and Security Considerations

One of the important ethical and security aspects is the conversion of UML models into NoSQL databases. While NoSQL is more flexible and scalable, it usually needs to manage large quantities of sensitive data. In such situations, ensuring data privacy and security must adhere to best practices, such as:

- Access Control Mechanisms: Role-based access control (RBAC) to prevent unauthorized access to data.
- Encryption: Data at rest and in transit is encrypted to mitigate breach risks.
- Data usage and reporting features: provide audit trails, logging access to and modifications of information to trace how it is being used and identify unusual or threatening behavior
- Regulatory Compliance: Complying with regulations such as GDPR and HIPAA to protect data.

Integrating security considerations into the transformation process allows MDA to contribute to more resilient NoSQL database architectures. This allows MDA to facilitate stronger NoSQL database architectures by ensuring data integrity, compliance, and security best practices.

1.2. Big Data Concepts

Big Data is defined by three fundamental characteristics: Volume, Variety, and Velocity. Volume refers to the massive amount of data generated, stored, and processed, often reaching petabyte or even exabyte scales. Variety highlights the diverse nature of data formats, encompassing structured data from traditional databases, semi-structured data such as JSON and XML, and unstructured data like images, videos, and social media content.

Velocity represents the speed at which data is produced and ingested, often requiring real-time or near-real-time processing to extract meaningful insights. These dimensions collectively present significant data storage, management, and analysis challenges. Designing NoSQL database structures that can efficiently handle high-volume, heterogeneous, and rapidly changing data streams is essential for modern applications, particularly in domains such as cloud computing, artificial intelligence, and IoT, where real-time data processing is critical.

2. Research Background

2.1. Model-Driven Architecture (MDA)

Model-Driven Architecture (MDA): MDA is a software design approach by the Object Management Group (OMG) that uses high-level models to lead the software development process. The general intention of MDA is to decouple the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. MDA allows for automatic transformations between various models, greatly minimizing manual coding and errors.

2.2. Key Concepts of MDA

The MDA approach is based on the concept that models should be the essential product Within the software development method. It provides a process in which the systems' design, analysis, and implementation are based on abstract representations or models, which can be further refined and transformed in different ways throughout the development lifecycle. MDA is centered on the idea of models and metamodels. Models abstract important aspects of the system, showing its structure and behavior in a formalized way. Whatever the technology, whatever the platform, it is a very high level of abstraction, just fifty peas. Metamodels describe the language and rules that govern the creation of these models. Prioritization techniques offer a uniform structure to ensure consistency and standardization between models.

A MDA also distinguishes levels of abstraction, namely Program Independent Model (CIM), Program Independent Model (PIM), and Program Specific Model (PSM). It focuses on the business context and the business requirements without getting into the technical implementation. The PIM defines the system's functionality and behavior without any platform-specific details. Compared with the PIM, the PSM adapts the platform-independent model (PIM) together with all the technical information that a system requires in the implementation platform. Organizing them at various levels of abstraction helps decompose the domain and build more manageable and scalable systems. In short, v. MDA enables a systematic, standardized software engineering approach, which can make it more agile and robust [12].

2.3. Transformation Types of MDA

Data is modeled using the System Modeling Language (SysML) and the Unified Modeling Language (UML) languages for transformations that belong to the Model-Driven Architecture (MDA) categories. There are two major types of transformation: M2M (Model-to-Model) and M2T (Model-to-Text). Example-based transformations: Model-to-Model (M2M) transformations are used to convert from one model to another, which may be at the same level of abstraction or between different levels. This is imperative in transforming abstract models (Platform Independent Models (PIMs)) into more detailed low-level Platform Specific Model

(PSM)s, or, into PIMs with other focus. Concrete M2M transformation consists of mapping rules that indicate how elements from the source model relate to the target model. These rules can be expressed using transformation languages (for example, QVT (Query/View/Transformation) or ATL (ATLAS Transformation Language)) that offer formal ways to define, run and automate the transformation of models.

Model-to-Text (M2T) transformations produce textual artifacts from models. They can be source code, configuration files, documentation and other text files needed for the system to be implemented. M2T transformations are crucial for covering the distance between high-level models and real systems. Acceleo (<http://www.eclipse.org/acceleo/>) and Xpand (<http://www.eclipse.org/Xpand/>) are examples of such tools providing M2T (M2T, Model To Text) transformations through a template-based approach where certain templates explain how certain elements in the model are translated into concrete textual/formatted representation of the model. This functionality accelerates the software development process while providing uniformity and precision in the generated artifacts. M2M and M2T transformations support high-level abstractions and facilitate the generation of lower-level and executable pieces of a larger system, likewise enhancing the flexibility, scalability, and efficiency of the development process [12].

2.4. Challenges in Transforming UML Models to NoSQL Schemas

There are several important challenges when transforming UML models into No SQL schemas, mainly because UML has a much more structured model, while No SQL databases can be schema-less. One main challenge is that UML models (for example, associations, inheritance hierarchies, constraints) have a well-defined hierarchy and relationship that needs to be kept in the NoSQL data model, where relationships are often lost and denormalized. It's

essential to preserve its integrity and consistency in this process, given that such a mapping, particularly one that requires complex UML model constraints and relationships, must be a comprehensive and faithful representation of the UML model. Scalability and flexibility are another challenge; NoSQL databases are purpose-built for large-scale, unstructured data storage, so the process of converting structured UML models to efficient NoSQL schemas that provide horizontal scalability adds complexity to the mapping process. Moreover, there is a lack of comprehensive tools and frameworks capable of automating this transformation process, which entails complex transformation rules and algorithms and the need for numerous manual adjustments, making it time-consuming and prone to errors. It further complicated the process, which now includes provisioning, provisioning time and validity of the generated schemas and sample data correctness, which requires additional testing and verification of whether they meet functional and specification requirements. Such challenges underline the importance of new methodologies and tools to aid precise, dynamic and scalable translations from UML models to NoSQL schemas [13].

3. Methodology

We follow a Model-Driven Architecture (MDA)-based development process that expresses the development effort in terms of high-level models. With this, one can convert UML models into a NoSQL database schema written in JavaScript. Following MDA principles and transformations, this methodology includes several steps: preparing the technical environment where some technical prerequisites were made available in order to connect to MongoDB securely, UML models parsed into JSON objects and then undergoing Model-to-Model (M2M) transformations, Model-to-Text (M2T) transformations, and finally, validating the schemas (M2T) using MongoDB Atlas. The overall approach steps are shown in the diagram below:

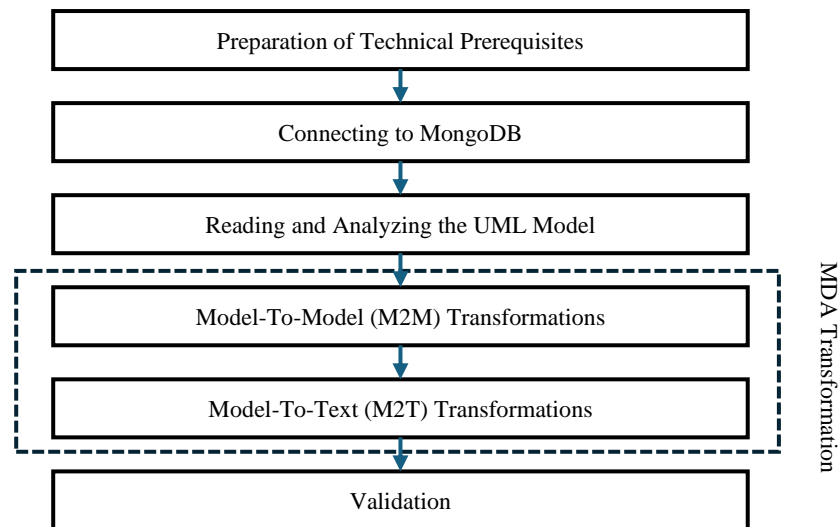


Fig. 1 Diagram illustrates the steps in our approach

3.1. Preparation of Technical Prerequisites

First, it imports modules: fs - a file system module that provides a standard API for interacting with the local computer's file system; xml2js - it provides a way to parse XML, which seems to be in Ecore XML format; mongoose - a library, that connects and interacts with MongoDB; and lastly, the Schema function from mongoose, which is used to define the structure of documents in a MongoDB collection. The code for this step looks like this

```
const fs = require('fs');
const xml2js = require('xml2js');
const mongoose = require('mongoose');
const { Schema } = mongoose;
```

Fig. 2 Importing necessary modules for UML to NoSQL transformation

3.2. Preparation of Technical Prerequisites

This step connects you to a MongoDB Atlas cluster using a MongoDB connection string. But when you go inside the credentials, you will see a URI used to connect to the MongoDB database. This way, the transformed schemas will be alloyed with the instance using the string provided. The code for this step looks like:

```
// MongoDB Atlas connection string
(replace <username>, <password> and
<cluster> with your MongoDB Atlas
credentials)
const uri =
'mongodb+srv://<username>:<password>@<clus
ter>/?retryWrites=true&w=majority'
```

Fig. 3 MongoDB atlas connection string

3.3. MDA Method

Model Driven Architecture: MDA is a software design approach centered on creating models from different viewpoints, for instance, an idealized target model, an achievement model, and different additional levels of views. MDA defines system functionality separately from the implementation specification on a given platform. There are three main levels of abstraction in the process:

```
// Define MongoDB schema definitions based on UML classes from Ecore model
const generateSchemasFromEcore = (ecoreModel) => {
  const eClasses = ecoreModel['ecore:EPackage']['eClassifiers'];
  const schemas = {};
  eClasses.forEach(eClass => {
    const className = eClass['$']['name'];
    const schemaDefinition = {};
    eClass['eStructuralFeatures'].forEach(feature => {
      const featureName = feature['$']['name'];
      const eType = feature['$']['eType'];
      // Map UML attributes to MongoDB schema fields
      if (feature['$']['xsi:type'] === 'ecore:EAttribute') {
```

3.3.1. Reading and Analyzing the UML Model

Reading and analyzing the UML model from the Ecore XML file. First, we read the Ecore XML file where the UML model is stored with the file system module (fs). After reading the file, we then parse it through the xml to json parser (xml2js) to allow us to turn the XML data into usable JavaScript objects. This conversion is critical because it translates the structured form of the UML model into a format a JavaScript machine can work with. The produced JSON objects maintain the hierarchy and relations specified in the UML model and allow further transformations required to generate NoSQL schemas. Transformations to RTL or even gate-level implementations are based on these objects, and the process ensures that every important object in the UML model remains intact and translates properly to be ready to be used in later transformations.

3.3.2. Generating MongoDB Schemas from the Ecore Model

When the data has been converted to JSON, each of the provided UML classes will be traversed to pull out attributes and references. UML attributes translate to MongoDB schema fields, and references are turned into either sub-documents or ObjectId references. This transition verifies that the UML entities and relationships are appropriately mapped to MongoDB structures. NoSQL431 generated MongoDB schemas from UML models without losing the hierarchy and relationships between them. Thus, this transformation results in an implementation that adheres to the original UML model regarding structure and integrity constraints, enabling appropriate access and use of data within a NoSQL-based system. At this point, intermediate specific to NoSQL schemas produced in this step are ready for a final definition of MongoDB schemas and their deployment in MongoDB Atlas, guaranteeing a faithful and coherent representation of the conceptual models in the NoSQL database. The function generateSchemasFromEcore records each UML class and its features, correlating UML attributes to MongoDB schema properties and UML references to MongoDB subdocuments. We then provide a one-to-one mapping of each UML class to the MongoDB schema. This process ensures that the hierarchy structure and relationships specified in the UML model are accurately mapped to MongoDB schema definitions.

```

        schemaDefinition[featureName] = String; // For simplicity, assuming all
attributes are strings
    }
    // Map UML references (EReferences) to MongoDB subdocuments
    if (feature['$']['xsi:type'] === 'ecore:EReference') {
        const refType = eType.replace('#//', '');
        schemaDefinition[featureName] = [{ type: Schema.Types.ObjectId, ref: refType
    }];
    }
    });
    // Define MongoDB schema for each UML class
    schemas[className] = new Schema(schemaDefinition);
    });
    return schemas;
};

```

Fig. 4 Generating MongoDB schemas from UML models

3.3.3. Saving Sample Objects in MongoDB

After creating the MongoDB schemas, ensure the transformation boots up by persisting sample objects in MongoDB.

Mongoose Establish Connection through generated schemas with Mongoose models and creating objects from

these models. For instance, we create objects from Entity such as Student and Course with sample data so that they are saved in MongoDB. This ensures that the generated schemas are accurate and that the data can be stored and retrieved from storage efficiently. We use MongoDB Atlas to perform the final validation, thus ensuring that the transformed schemas are faithful to and correct for each of the original UML model's specifications.

```

// Function to save sample objects to MongoDB based on the generated schemas
const saveSampleObjectsToMongoDB = (schemas) => {
    mongoose.connect(uri).then(() => {
        // Define Mongoose models based on generated schemas
        const models = {};
        Object.keys(schemas).forEach(className => {
            models[className] = mongoose.model(className, schemas[className]);
        });
        // Sample data for objects to be saved in MongoDB
        const Student = models['Student'];
        const Course = models['Course'];
        const newStudent = new Student({
            studentId: 'S001', firstName: 'John', lastName: 'Doe',
            username: 'johndoe', password: '12345'
        });
        const newCourse = new Course({
            courseId: 'C001', title: 'Introduction to Programming',
            description: 'Learn basic programming concepts'
        });
        // Save the sample objects to MongoDB
        return Promise.all([
            newStudent.save(),
            newCourse.save()
        ]);
    })
    .then(() => {
        console.log('Sample objects saved to MongoDB');
    })
    .catch(err => {

```

```

        console.error('Error saving sample objects:', err);
    })
    .finally(() => {
        mongoose.disconnect();
    });
};

```

Fig. 5 Saving sample objects MongoDB

The following code executes the transformation, calling the other methods already defined. It parses and loads the UML model from the Ecore XML file, transforms it into a

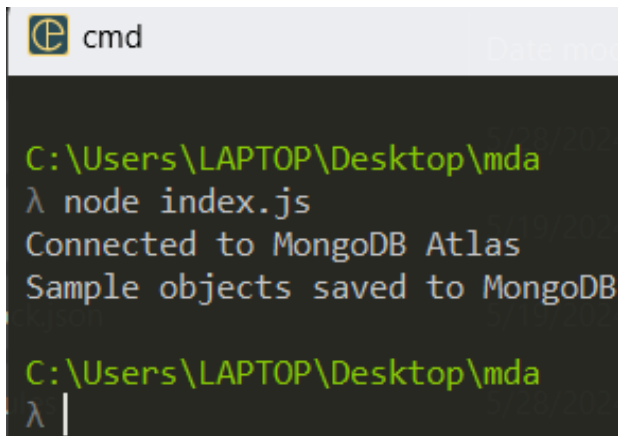
JavaScript object, generates MongoDB schema definitions from the UML classes, and inserts sample objects into MongoDB:

```

// Read and parse the UML model from the Ecore XML file
const xmlModelPath = 'model.xml';
const parser = new xml2js.Parser();
fs.readFile(xmlModelPath, 'utf-8', (err, data) => {
    if (err) {
        console.error('Error reading file:', err);
        return;
    }
    // Parse the XML data into a JavaScript object
    parser.parseString(data, (parseErr, result) => {
        if (parseErr) {
            console.error('Error parsing XML:', parseErr);
            return;
        }
        // Generate MongoDB schema definitions based on UML classes from Ecore model
        const schemas = generateSchemasFromEcore(result);
        // Save sample objects to MongoDB based on the generated schemas
        saveSampleObjectsToMongoDB(schemas);
    });
});

```

Fig. 6 The main code source to execute the transformation



```

cmd
C:\Users\LAPTOP\Desktop\mda
λ node index.js
Connected to MongoDB Atlas
Sample objects saved to MongoDB
C:\Users\LAPTOP\Desktop\mda
λ |

```

Fig. 7 Running the transformation script

The following figure illustrates the execution of the transformation script in the command line interface. The script connects to MongoDB Atlas and confirms that sample objects have been successfully saved to MongoDB. The console output shows the commands executed and the resulting messages, indicating a successful connection and data storage.

3.4. Handling Complex UML Models and Performance Evaluation

The solution part addresses the intricacies of correctly and effectively mapping UML models to MongoDB schemas. Mapping inheritance hierarchies, we transpose UML inheritance to MongoDB schemas using methods like document embedding for subclasses or referencing in polymorphic associations. To map multiplicities and associations, this solution maps different association types (one-to-one, one-to-many, many-to-many) using embedded documents or references with data consistency and integrity guarantees. We enforce UML-defined constraints, such as uniqueness and range constraints, by means of schema validation rules, custom validators, or application logic. Complex attribute types, like arrays, lists, and user-defined types, are translated into MongoDB-compatible types. To avoid performance issues due to deeply nested documents, we denormalize nested document hierarchies or employ flat structures with references. To measure the efficiency of our methodology, we metricized transformation time, error rate, schema correctness, development effort, scalability, and consistency. We compared the transformation time in the

automated approach with the manual approach and identified a significant amount of time savings for various sizes of models. The error rate, expressed as schema mismatches and data integrity violations, resulted in fewer human errors with automation. Schema fidelity was tested by comparing the schemas generated to expected outcomes, quantifying in attribute types, relationships, and constraints. The development effort was gauged via hours for hand vs. auto transformations, stressing efficiency gains. Scalability experiments illustrated the process's performance when UML model size and complexity increased, charted in a scalability graph. Consistency and reproducibility were verified by applying the same transformation to multiple copies of the same UML model and seeing consistent MongoDB schema output. These performance statistics all demonstrate the strength and efficacy of our automated transformation mechanism.

4. Results and Discussion

Our strategy for the transformation of UML models into NoSQL schemas was successful. The mapping of the UML models into JSON objects preserved the structure and relationships of the source model. The M2M and M2T transformations enabled us to generate correct and functional MongoDB schemas.

The validation against MongoDB Atlas ensured that the schemas generated were correct and that the sample objects were stored and retrieved correctly. This approach guarantees data consistency and flexibility in meeting Big Data demands. The following image shows what we should see after doing a query of courses collection in MongoDB Atlas:

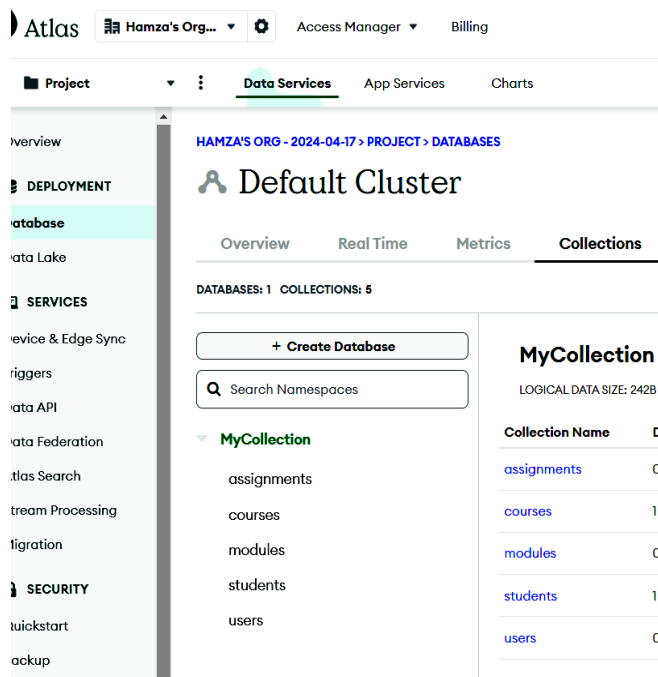


Fig. 8 MongoDB atlas collections overview

MyCollection.courses

STORAGE SIZE: 20KB LOGICAL DATA SIZE: 153B TOTAL DOCUMENTS: 1 INIT

Find Indexes Schema Anti-Patterns 0 Aggreg

Generate queries from natural language in Compass

Filter

Type a query: { field: 'value' }

QUERY RESULTS: 1-1 OF 1

```
{
  "_id": ObjectId('661ff3115ffa0420b89a29df'),
  "courseId": "C001",
  "title": "Introduction to Programming",
  "description": "Learn basic programming concepts",
  "modules": Array (empty),
  "__v": 0
}
```

Fig. 9 MongoDB atlas query result for course collection

It ensures that the Course object with data like courseId, title, and description was saved in the database and verifies that the data was transformed and stored as expected, and the UML model's details regarding the objects' structure and content match the outcome.

5. Conclusion

Against the backdrop of Big Data evolution, traditional relational database systems show their limitation in scalability and flexibility. To resolve this issue, we have designed a novel solution for transforming UML models to NoSQL schemas using JavaScript, i.e., for MongoDB. We start the process by reading and parsing UML models from Ecore XML files and then their mapping to usable JSON objects. Secondly, Model-to-Model (M2M) transformations are employed to generate intermediate NoSQL schemas, which are then converted into MongoDB schemas using Model-to-Text (M2T) transformations.

Finally, we validate these schemas against MongoDB Atlas so that test objects are stored and retrieved correctly. The result shows that the provided method effectively preserves the structure and relationships defined in the UML models without sacrificing the scalability and flexibility requirements of Big Data environments.

In the future, this transformation framework will be generalized to accommodate other structures of NoSQL databases like document stores, column-family stores, and graph databases. In addition, future work shall focus on improving transformation efficiency, automation, and semantic validation mechanisms to strengthen these proposals across several real-world scenarios.

References

- [1] Fatma Abdelhadi, Amal Ait Brahim, and Gilles Zurfluh, "Applying a Model-Driven Approach for UML/OCL Constraints: Application to NoSQL Databases," *On the Move to Meaningful Internet Systems: OTM Conferences*, Rhodes, Greece, pp. 646-660, 2019. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [2] Fatma Abdelhedi et al., "MDA-Based Approach for NoSQL Databases Modelling," *Big Data Analytics and Knowledge Discovery*, pp. 88-102, 2017. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [3] Fatma Abdelhedi, Amal Ait Brahim, and Gilles Zurfluh, "Towards an Automatic Approach for Implementing UML/OCL Models on NoSQL Systems," [[Google Scholar](#)]
- [4] Jean Bézivin et al., "Applying MDA Approach for Web Service Platform," *Proceedings. Eighth IEEE International Enterprise Distributed Object Computing Conference*, 2004. *EDOC 2004*, Monterey, CA, USA, pp. 58-70, 2004. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [5] M'hamed Rahmouni, Chaymae Talbi, and Soumia Ziti, "Model-Driven Architecture: Generating Models from Symphony Framework," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 30, no. 3, pp. 1659-1668, 2023. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [6] M'hamed Rahmouni, and Samir Mbarki, "Model-Driven Generation of MVC2 Web Applications: From Models to Code," *International Journal of Engineering and Applied Computer Science*, vol. 2, no. 7, pp. 217-231, 2017. [[Google Scholar](#)]
- [7] Moneeb Abbas et al., "A Model-Driven Framework for Security Labs Using Blockchain Methodology," *2021 IEEE International Systems Conference*, Vancouver, BC, Canada, pp. 1-7, 2021. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [8] Nour Moadad, Issam Damaj, and Islam El Kabani, "A Generic MDA-IoT Architecture for Connected Vehicles in Smart Cities," *2022 IEEE Global Conference on Artificial Intelligence and Internet of Things*, Alamein New City, Egypt, pp. 122-129, 2022. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [9] Mantas Jurgelaitis et al., "Smart Contract Code Generation from Platform Specific Model for Hyperledger Go," *World Conference on Information Systems and Technologies*, Terceira Island, Portugal, pp. 63-73, 2021. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [10] Mohammad Ali Kadampur, and Sulaiman Al Riyace, "Skin Cancer Detection: Applying a Deep Learning Based Model Driven Architecture in the Cloud for Classifying Dermal Cell Images," *Informatics in Medicine Unlocked*, vol. 18, 2020. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]
- [11] Hanane Benouda et al., "Modeling and Code Generation of Android Applications Using Acceleo," *International Journal of Software Engineering and its Applications*, vol. 10, no. 3, pp. 83-94, 2016. [[Google Scholar](#)] [[Publisher Link](#)]
- [12] Model Driven Architecture (MDA), The Architecture of Choice for a Changing World, Object Management Group (OMG), 2003. [Online]. Available: <https://www.omg.org/mda/>
- [13] Aaron Schram, and Kenneth M. Anderson, "MySQL to NoSQL: Data Modeling Challenges in Supporting Scalability," *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, Tucson Arizona, USA, pp. 191-202, 2012. [[CrossRef](#)] [[Google Scholar](#)] [[Publisher Link](#)]