# Propagating Schema Changes to Code: An Approach Based on a Unified Data Model

Alberto Hernández Chillón*, Jesús García Molina, José Ramón Hoyos and María José Ortín

*Faculty of Computer Science, University of Murcia, Murcia, Spain.*

Abstract
Schema evolution is a crucial activity when managing databases in order to satisfy new requirements and improve data and code quality. Therefore, a great research effort has been devoted to automate this activity for relational databases, and agile database development is the most recent innovation. With the emergence of NoSQL systems, the attention has shifted to study schema evolution for these stores that are characterized by the absence of a declaration of schema, and the use of APIs instead of the usage of SQL-like languages. Moreover, the number of multi-model database systems and tools is continuously growing as polyglot persistence becomes the architecture of choice to support the requirements of modern applications. In this scenario, several works have presented approaches to support schema evolution for relational and NoSQL systems. In previous work, we presented the U-Schema unified metamodel to represent schemas for relational systems and the four most widely used NoSQL paradigms. For U-Schema, we defined a taxonomy of schema change operations, and the Orion language to specify scripts defining sets of schema changes based on operations of the taxonomy. The Orion engine is then able to automatically update schema and data from Orion scripts. In this paper, we present an ongoing work aimed at adding schema and code co-evolution to Orion. The novelty of our proposal is to define schema changes in a platform-independent way, and then automatically generate static code analysis queries to find the statements that need to be updated. These queries are obtained by using model to text transformations, and they return information about the location of the code fragments to be updated, and the modifications that need to be applied so the code conforms to the new schema.

**Keywords**
NoSQL databases, Schema evolution, Taxonomy of changes, Code update

## 1. Introduction

In the industrial and academic database community, the idea of "one size does not fit all" has been gaining acceptance and *polyglot persistence* is the scenario foreseen for the future [1, 2]: relational database systems will be still predominant but they will coexist with other kinds of systems, such as NoSQL and NewSQL. In fact, the first eight databases in the DB-engines ranking [1] are multi-model, and database tooling provide support for database systems of different data model paradigms.

When several data models are popular and widely used, the definition of a unified or generic metamodel is a well-known approach to reduce the effort of implementing multi-model database utilities and tools: the generic metamodel is able to represent schemas of all the involved data models. Some examples of such generic metamodels are DB-Main [3] and more recently TyphonML [4] and U-Schema [5]. In our research group, we created U-Schema to provide a unified representation for relational and the four most used NoSQL data models: columnar, document, key-value, and graph [2]. We have used U-Schema to design and implement the Orion approach for evolving databases [6, 7] and the SkiQL language aimed to query schemas [8]. Here, will focus on the schema evolution based on the Orion language.

Schema evolution is a crucial task in database applications: a co-evolution of data and code is required when schema changes are applied, as illustrated in Figure 1. Therefore, a great research effort has been devoted to deal with this problem in the scope of relational databases [9, 10]. Since the appearance of NoSQL systems, a great attention has also been paid to the evolution of these systems [11, 12], which have two significant differences regarding the relational systems: most of them do not require the declaration of schemas and queries are based on APIs instead of using a SQL-like language. Moreover, some approaches have also addressed the polyglot persistence scenario [13, 14].

In [6, 7], a taxonomy of schema changes for U-Schema is proposed, and the Orion language is defined to express schema change scripts. Co-evolution of schema and data is automatically applied by the Orion engine. In this paper, we present an ongoing work intended to add co-evolution of schema and code to Orion. Developers will be assisted in two ways: depending on the kind of schema change, code will be automatically updated or

*Published in the Workshop Proceedings of the EDBT/ICDT 2023 Joint Conference (March 28-March 31, 2023, Ioannina, Greece)*

*Corresponding author.
alberto.hernandez1@um.es (A. Hernández Chillón); jmolina@um.es (J. García Molina); jose.hoyos@um.es (J. R. Hoyos); mjortin@um.es (M. J. Ortín)
0000-0002-1154-9192 (A. Hernández Chillón); 0000-0003-4685-6659 (J. García Molina); 0000-0002-0022-4771 (J. R. Hoyos); 0000-0002-5507-5566 (M. J. Ortín)

[1]https://db-engines.com/en/ranking.

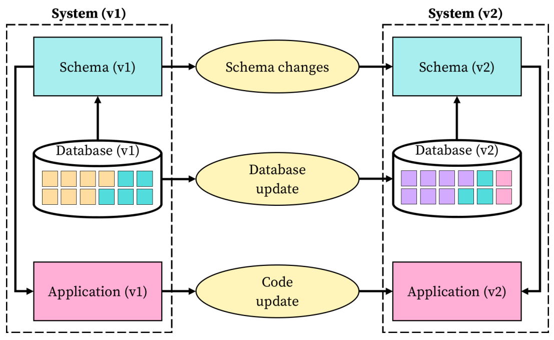

**Figure 1:** Data and code co-evolution when the schema changes. (Extracted from [7].)

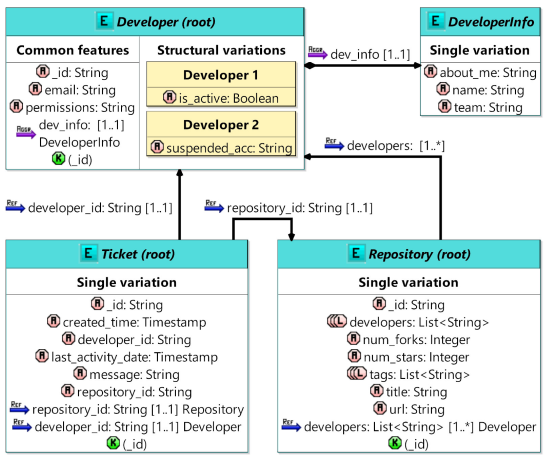

**Figure 2:** The *SoftwareDev* schema running example.

either messages carrying information about the changes to be applied manually will be shown to developers, in a way similar to those described in [13]. Here, we will present the current state of our work. The novelty of our proposal is to define schema changes in a platform-independent way, and then automatically generate static code analysis queries to find the statements to be updated. These queries are obtained from a Orion script through model to text transformations. We have validated generated queries valid for MongoDB APIs and mappers for the JavaScript language. Our work is described in Section 3. Prior to this explanation we briefly introduce the Orion approach, and we conclude by commenting the related work and exposing some ideas about the tasks to be done to complete the approach.

## 2. U-Schema and the Orion Language

U-Schema is a unified metamodel that integrates the NoSQL and relational data models. It includes the elements traditionally used to create logical schemas, as those that are part of the Entity-Relationship (ER) model [15]. However, these elements are structured differently because their semantics are not exactly the same, and additional concepts are considered (e.g. structural variation of types), as briefly explained below. A detailed description of U-Schema and the bidirectional mappings between this unified metamodel and each of the individual data models can be found in [5].

A U-Schema model (i.e., a logical schema) consists of a set of schema types that can be *entity types*, used to represent domain entities in any database, and *relationship types* to represent relationships between nodes in graph databases. This illustrates that not all U-Schema concepts are present in all the data models that it integrates.

An entity type is formed by a set of features that can be structural or logical. The former are simple and multi-valued attributes and aggregation relationships, and the latter refer to relations between entities, namely keys and reference relationships. Entity types can be *root* or *non-root* depending on if their objects are the root of an aggregation hierarchy or are embedded in other objects.

Figure 2 shows a U-Schema model example named *SoftwareDev*, which will be used as a running example. We will suppose that this schema corresponds to a document database, in particular MongoDB, and therefore it has no relationship types. The schema is shown by using the graphical notation described in [16], whose semantics will be easily understood from the explanation of the schema. There are three root entity types in *SoftwareDev* schema: `Developer`, `Ticket`, and `Repository`, and there is one non-root entity type (`DeveloperInfo`) that is aggregated by `Developer`.

U-Schema allows structural variations of schema types to be represented. Thus, a schema type is formed by a set of variations. In the schema example, `Developer` has two structural variations that share a common set of features (e.g., `email`, `permissions`, and `dev_info`) but differ in two features: The first variation stores a feature named `is_active`, and the second one has `suspended` instead. The remaining entity types have a single variation with a set of features.

In the graphical notation, features are specified by means of a name and a type, but aggregations and references are also shown by means of the typical arrows used in UML class diagrams. In the example, `Repository.title` is an attribute, `Developer.dev_info` aggregates an instance of `DeveloperInfo`, `Ticket.repository_id` and `Ticket.developer_id` hold references to the corresponding entity types, and `Repository.developers` references the set of developers involved in a repository.

It is convenient to note that references and relationship types are different concepts. The former represent foreign keys of the relational model or links between database objects, and they cannot have features, and the latter represent connections between nodes in a graph database, which can have features.

Typically, schema evolution approaches define a set of *schema change operations* (SCOs) that can be applied on a particular data model, and classify these changes in a taxonomy depending on the element they affect [17, 13, 14]. In [6, 7], we defined a taxonomy for U-Schema, which classifies SCOs in several categories that correspond to the U-Schema elements: *Schema type*, *Structural Variation*, *Feature* (grouping common operations for each kind of feature), *Attribute*, emphReference, and *Aggregate*, as can be seen in the first column of Table 1, where some of the most relevant operations are shown.

Once the taxonomy was defined, we developed the Orion textual domain-specific language (DSL) and the Orion engine to automate the schema change evolution. A metamodeling-based technique was applied to build Orion: a metamodel defines its abstract syntax, and model transformations implement its semantics (i.e., its engine) [7]. The Orion engine first injects the input Orion script into a Orion model that conforms to the metamodel, then the sequence of SCOs are executed in order to update schema and data. Figure 3 shows the components of the engine: (i) A *Schema Updater* whose input are the U-Schema and Orion models and generates the updated schema, and (ii) a *Data Updater* for each supported database system, which automatically generates the data updating code from the input Orion model. Up until now we provide data updaters for three of the most commonly used NoSQL stores: MongoDB (document), Cassandra (columnar), and Neo4j (graph).

An example of an Orion script is shown in Figure 4, which is intended to apply a refactoring to the *SoftwareDev* schema. First, an unnecessary feature is deleted and two more features are renamed. Then, a COPY operation is applied to copy a field of the `Ticket` entity into its corresponding `Repository`, by using a `JOIN` condition. Next, two feature adding operations (ADD ATTR and ADD AGGR) are used to add a new attribute and a new aggregation, and a CAST operation is applied to change the type of the `Developer.suspended_acc` attribute. The script example ends with two operations on entities: `Ticket` is renamed to texttt Active_Ticket, and a new entity is added to store tickets already resolved.

Given this script, the Orion engine provides a new version of the *SoftwareDev* schema (*SoftwareDev:2*), and generates code scripts to update the MongoDB database. We will present now the current status of our work intended to extend the Orion engine with capabilities to detect code affected by the Orion script, and provide information to developers about the changes to be made.

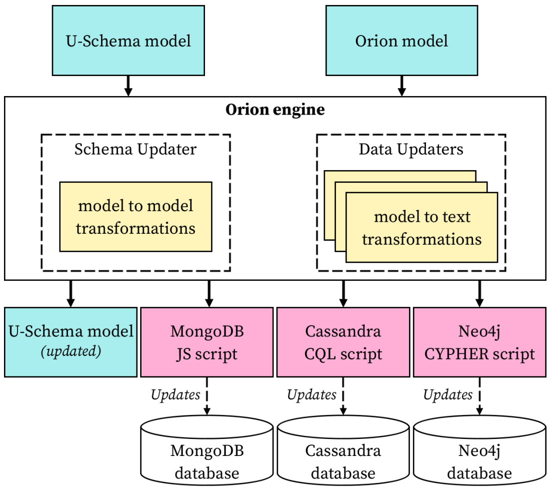

**Figure 3:** The Orion Engine handles schema and data updates.

```
SoftwareDev_ops operations

USING SoftwareDev:1

DELETE Developer::permissions
RENAME *::num_forks TO rank_forks
RENAME *::num_stars TO rank_stars

COPY Ticket::last_activity_date
  TO Repository::last_ticket
  WHERE Ticket.repository_id = Repository._id
ADD ATTR Repository::subscribers: Number
ADD AGGR Developer::dev_location:
  { city: String, country: String }&
  AS DeveloperLocation
CAST ATTR Developer::suspended_acc TO Boolean

RENAME ENTITY Ticket TO Active_Ticket
ADD ENTITY Archived_Ticket: { +_id: String,
  archived_date: Timestamp, message: String }
```

**Figure 4:** Orion script applied to the *SoftwareDev* schema.

## 3. Detection of Code Affected by Schema Changes

In this section, we describe the proposed approach in detail. An overview of the designed and implemented strategy is firstly presented. Secondly, the mappings between each Orion operation and the updates to be performed are discussed. Thirdly, some details about the code analysis based on CodeQL queries are commented, and finally the obtained output for the schema of the running example is shown.

## 3.1. Overview of the Code Analysis Strategy

As noted in [13], depending on the semantics of the schema change operations, four different situations can be distinguished for the application code accessing databases: (i) Code may need to be modified according to the new schema, e.g., when features or entity types are renamed; (ii) code may become invalid, e.g., when entity types or features are removed; (iii) code is syntactically correct but should be modified according to the new schema to return correct values, e.g., when a casting is applied on an attribute; and (iv) code does not need to be changed since it is not directly affected, e.g., when adding a new entity type.

An analysis of the application code involves finding the statements affected by each SCO in an Orion script (e.g., renaming the `Ticket` entity type to `Active_Ticket`). Once one or more affected statements are identified, they could be modified to be adapted to the new schema, or either a message could be generated and shown to developers.

In order to perform such code analysis and resulting actions, a mapping should be established between each SCO of the taxonomy and its impact on the code, based on: (i) Identifying the statements that need to be updated, and (ii) providing a suitable solution according to the new schema. Table 1 shows the mappings for the currently addressed Orion SCOs. These operations cover the four categories mentioned above. In the table, the first column indicates the Orion SCO, and the second column specifies the actions to be applied on the code, which are expressed in an abstract way and independent of any technology. The third and fourth columns are specific to each programming language, in this case JavaScript, and they indicate which code should be inspected to detect warnings and errors, as commented later in this section.

Similarly to [13], our aim is to assist developers to update code with two kinds of facilities: (i) Automating all possible code changes, and (ii) generating log files that carry information about the automatically applied modification, and the changes that should be manually introduced by developers. Here, we will focus on the automatic generation of notification messages to provide developers with such information.

We have started to build the *Code Updater* component of the Orion engine by implementing a 3-step strategy that uses the CodeQL code analysis engine [18] to discover the code that is candidate to be changed. This strategy is outlined in Figure 5 and commented below.

As a preliminary step, the code repository is converted into a *CodeQL database* that can be accessed by CodeQL queries. In the first step, a model-to-text (*m2t*) transformation is executed to automatically generate the CodeQL queries that correspond to the input Orion model. This

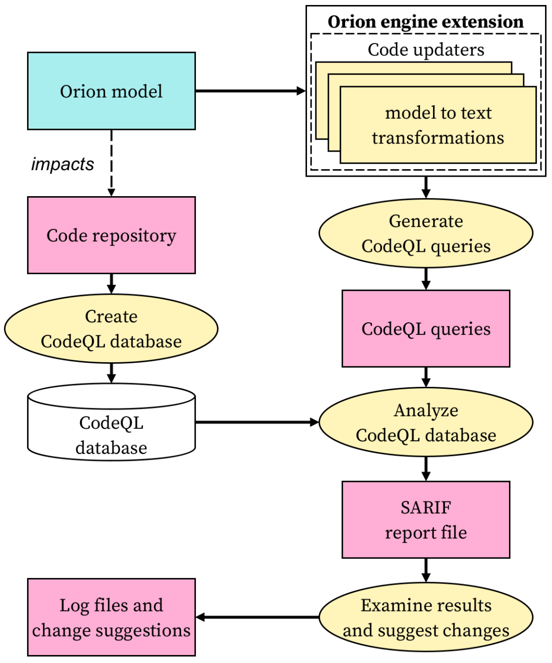

**Figure 5:** The Orion engine extension to handle code update.

transformation consists of a set of rules that implement a mapping between each Orion SCO and CodeQL queries able to find the location of the application code to be updated. Table 1 shows such mappings in the third and fourth columns. For each Orion operation up to two CodeQL queries are generated depending on whether the code has become invalid and needs to be updated according to the new schema (*error queries*), or the code should be modified, but it continues to be valid (*warning queries*).

In the second step, the generated queries are applied to the previously created *CodeQL database*. As a result of this application, a log file in SARIF format [2] is obtained. This file contains information of the analysis process, the applied queries, the examined files and every match found in those files by the queries, along with useful information to locate the code statements to be changed (e.g., the file where they are located, the line number, and a custom message).

Because the log file is rather complex and difficult to navigate, the third step performs a parsing of the log file and generates a digest useful for the developer where, for each Orion SCO in the input script, a set of messages show the statements that need to be changed along with their location, and which changes should be applied, e.g., renaming a variable to a new name or deleting an assignment.

[2]https://sarifweb.azurewebsites.net/.

**Table 1**
Mapping between Orion SCOs, actions to be taken on code and JavaScript statements to query with CodeQL

| Orion SCO | Actions on code | CodeQL JavaScript warning | CodeQL JavaScript error |
|---|---|---|---|
| **Entity Type Operations** | | | |
| **Add** $E$ | Detect and warn access to all collections. | On MongoDB methods that retrieve all collections | — |
| **Delete** $E$ | Detect and delete access and references to $E$. | On assignments with the literal *E.name*. | On access to $E$ by property, by method or by Mongoose schema. |
| **Rename** $E$ | Detect access and references to $E$. Update *E.name* with new name. | On assignments with the literal *E.name*. | On access to $E$ by property, by method or by Mongoose schema. |
| **Feature Operations (Attribute, Reference and Aggregate)** | | | |
| **Delete** $E.f$ | Detect and delete access and references to $E.f$. | — | On MongoDB methods involving $E.f$ and on the $E$ Mongoose schema. |
| **Rename** $E.f$ | Detect access and references to $E.f$. Update $E.f.name$ with new name. | — | On MongoDB methods involving $E.f$ and on the $E$ Mongoose schema. |
| **Copy** $E_1.f$ **to** $E_2.f$ | Detect and update the $E_2$ schema by adding $f$ to it. | On $E_2$ Mongoose schema. | — |
| **Move** $E_1.f$ **to** $E_2.f$ | Detect and delete access and references to $E_1.f$, and update the $E_2$ schema by adding $f$. | On $E_2$ Mongoose schema. | On MongoDB methods involving $E_1.f$ and on the $E_1$ Mongoose schema. |
| **Attribute Operations** | | | |
| **Add** *E.attr* | Detect and update the $E$ schema by adding *attr* to it. | On $E$ Mongoose schema. | — |
| **Cast** *E.attr* | Detect access and references to *E.attr*. Update *E.attr.type* to new type. | — | On MongoDB methods involving *E.attr* and on the $E$ Mongoose schema. |
| **Reference Operations** | | | |
| **Add** $E.ref$ | Detect and update the $E$ schema by adding $ref$ to it. | On $E$ Mongoose schema. | — |
| **Morph** $E.ref$ | Detect access and references to $E.ref$. Update $E.ref$ for an aggregate. | — | On MongoDB methods involving $E.ref$ and on the $E$ Mongoose schema. |
| **Aggregate Operations** | | | |
| **Add** *E.aggr* | Detect and update the $E$ schema by adding *aggr* to it. | On $E$ Mongoose schema. | — |
| **Morph** *E.aggr* | Detect access and references to *E.aggr*. Update *E.aggr* for a reference. | — | On MongoDB methods involving *E.aggr* and on the $E$ Mongoose schema. |

## 3.2. Analyzing the Application Code

For a first validation of our approach, we have chosen MongoDB as our target database and JavaScript as language of the database code. According to the DB-Engines ranking, MongoDB is the fifth most popular database (the first of the NoSQL stores), and the driver for Node.js JavaScript frameworks is one of the most used to access MongoDB databases in Web applications. Finally, we have also considered the Mongoose object-document mapper (ODM) [19, 20], which is the most widely used ODM for MongoDB in JavaScript.

In MongoDB, data on entity types are stored in collections of JSON-like semi-structured documents [21]. Therefore, a document is formed by a set of name-value pairs of features, and values can be primitives (e.g., *Number* and *String*), another nested document, or arrays.

Figure 6 shows an excerpt of JavaScript code accessing a MongoDB database for the running example. First, the `Developer` collection is retrieved by using a method call provided by the MongoDB API, and a query is performed that involves retrieving a single developer and using the `permissions` field. Then a variable is created with the `"Ticket"` value, and another method call retrieves the `Ticket` collection by using this variable, and applies a `deleteMany` operation with another query on it.

```
1  import { MongoClient } from "mongodb";
2  const uri = "<connection string uri>";
3  const client = new MongoClient(uri);
4
5  async function run() {
6   try {
7    await client.connect();
8    database = client.db("software_dev_sample");
9
10   // Try to access Developer.permissions
11   developers = database.collection("Developer");
12   result1 = await developers.findOne(
13     {_id: "13"},
14     {projection: {"permissions": 1}});
15
16   // Query the Ticket entity type
17   collection_name = "Ticket";
18   tickets= database.collection(collection_name);
19   query = { message: { $regex: "error" } };
20   result2 = await tickets.deleteMany(query);
21   console.log(result2.deletedCount);
22  } finally {
23   await client.close();
24  }
25 }
26 run().catch(console.dir);
```

**Figure 6:** JavaScript excerpt querying a MongoDB database.

As evidenced in this example, the code analysis should not only consider just database operations, but also any other statements such as assignments and variables that can carry values referring to schema elements (e.g., entity type names and feature names), and differentiate between these literal values and function or variable names that might have the same name. Moreover, the analysis needs to take into account that in the case of feature names, it is necessary to identify not only the feature, but also if it belongs to the entity indicated in the SCO.

On the other hand, Figure 7 shows a Mongoose schema for the `Repository` entity type of the running example. A Mongoose schema is composed of a set of fields with a name and a specific type, such as the `num_stars` field of type `Number`, or the `developers` field, which references the `Developer` entity type stored as a list of Strings.

In Table 1, the third and fourth columns shows which

```
import mongoose from "mongoose";
const RepositorySchema = new mongoose.Schema({
  _id:        {type: String, required: true},
  developers: {type: [String],
               ref: "Developer", required:true},
  num_forks:  {type: Number, required: true},
  num_stars:  {type: Number, required: true},
  tags:       {type: [String], required: true},
  title:      {type: String, required: true},
  url:        {type: String, required: true}
});
export default mongoose.model("Repository",
     RepositorySchema);
```

**Figure 7:** Mongoose schema representing the `Repository` entity type.

code statements are involved when applying a Orion SCO. The third column where a *warning* should be thrown, pieces of code which are syntactically correct but might be semantically incorrect. The fourth column, on the other hand, indicates *errors*, those updates that are mandatory to fix code that is now invalid.

According to these mappings, when an SCO that modifies an entity type is applied (e.g., a `DELETE` or `RENAME` operation), it is necessary to look for statements that access and return the corresponding collection, since those statements are now invalid. In JavaScript, it is common to use a `MongoClient` object to obtain a MongoDB `database` object by using the `.db()` method. Then, obtaining a specific collection from such `database` object can be achieved in several ways, classified in two kinds: (i) By *property* (e.g., `database.E` or `database[E]`, being `E` the name of the collection), and (ii) by *method* (e.g., `database.collection("E")` or `database.getCollection("E")`), as shown in Figure 6, lines 11 and 18. These statements have in common that the literal name of the entity is used at some point (in the examples, *E*), so when an SCO that modifies an entity type is applied, this kind of accesses need to be updated. Moreover, instead of directly using the literal "`E`" developers can use variables that hold such value, so we also need to look for variables that were assigned that value. Finally, the Mongoose schema corresponding to `E` must also be updated.

In the example, the `RENAME Ticket TO Active_Ticket` operation will cause the statement in line 18 to not be valid, and an *error* message should be triggered by our *Code Updater*. Also, a *warning* should be issued for the assignment in line 17 because it will not return the desired results, although such an assignment is still valid.

Detecting code errors due to an SCO that modifies a feature of an entity type requires of a different strategy.

To analyze the impact of an operation such as DELETE Developer::permissions, it is necessary to take into account that only references to the permissions feature belonging to Developer should be updated, and that other entity types might hold a feature with the same name. Therefore, queries for these operations are organized as follows. MongoDB API operations are first searched, such as findOne(), insertOne(), or deleteMany(). Then, the process checks if the operation is being applied over the specified entity type (e.g., an object storing the Developer collection). Finally, the code is analyzed to check if the field is referenced in some of the arguments of the method call, which use to be expressions or JSON objects containing the feature name. In our example, the statement in line 12 should be updated to reflect that the permissions field referenced by projection has been deleted.

Operations that modify features also impact Mongoose schemas. For example, the Orion operation RENAME *::num_forks TO rank_forks would require updating the Repository schema shown in Figure 7, where the num_forks field should be modified to reflect the RENAME operation.

Finally, we have to consider SCOs adding new features to existing entity types, such as COPY or ADD attr. These SCOs do not turn invalid code, and therefore do not issue any *errors*, but we provide information for developers by issuing *warnings* on Mongoose schemas indicating that the developer should add the new features.

## 3.3. CodeQL Query Implementation

Once we defined which code statements throw warnings and errors for each considered SCO, we used CodeQL [18] to implement queries that return those precise statements from a code repository. CodeQL is a code analysis engine developed by GitHub. It generates a database representation of a code base, so the code can be treated as data. Code patterns are modeled as CodeQL queries that can be executed to generate a result set that includes those lines of code that matches the pattern. We have chosen CodeQL because it is a semantic code analysis engine, it supports a wide variety of programming languages, and it can be integrated with GitHub repositories to analyze the code they contain.

A CodeQL query includes three clauses, as illustrated in the example of Figure 8: (i) FROM defines the kind of element being queried (in this example, Expr, a class used to get JavaScript expressions), (ii) WHERE is used to filter only the elements considered in the first clause that also meet certain criteria, and (iii) SELECT to return the elements that were filtered and a custom message.

Some Orion SCOs are mapped to similar CodeQL queries. For example, DELETE and RENAME an entity type require updating the same statements. The difference between these operations comes on the solution to be applied for the update: For DELETE, the involved statements should be removed as the collection being deleted no longer exists, while for the RENAME operation the involved statements should be updated to reference the new name. The same case happens for ADD Attribute, ADD Reference, and ADD Aggregate: In JavaScript and MongoDB these operations do not turn any code invalid. Instead, queries for these operations look for Mongoose schema declarations and suggest updating such schemas by adding the new fields.

These similarities between SCOs ease the process of generating CodeQL queries. For this purpose, we created a library of CodeQL functions which allows simpler queries, as illustrated in the query of Figure 8: A CodeQL *error* query generated from the RENAME ENTITY Ticket TO Active_Ticket. Here, the query looks for expressions that meet one of the following conditions on the Ticket collection: (i) It is accessed as a property (e.g., database.Ticket), (ii) it is accessed by using a method that retrieves a collection (e.g., getCollection), and (iii) there is a Mongoose export statement that generates a Ticket schema. These conditions are implemented with three functions of the library: checkPropAccess, checkExprIsCollectionMethod, and checkIsMongooseExport, and they can be used for other SCOs. Finally, a message stating how the code needs to be updated is shown: "*Ticket: Entity renamed to "Active_Ticket"*.".

```
IMPORT javascript
IMPORT utils

FROM Expr expr
WHERE
 checkPropAccess(expr, "Ticket")
 OR
 checkExprIsCollectionMethod(expr, "Ticket")
 OR
 checkIsMongooseExport(expr, "Ticket")
SELECT expr,
  "Ticket: Entity renamed to \"Active_Ticket\"."
```

**Figure 8:** CodeQL *error* query generated for the RENAME ENTITY Ticket TO Active_Ticket operation.

The same principle is followed for generating a CodeQL query for the operation DELETE Developer::permissions, shown in Figure 9. Here, instead of single generic expressions, we look for MethodCallExpr, a CodeQL class used to get expressions in which a method is invoked. According to what was stated in the previous subsection, we need to update expressions that meet one of the following conditions: (i) A MongoDB method is invoked over a specific entity (`Developer`) and the `permissions` field shows as one of its arguments (either directly or embedded as a value in a JSON object), or (ii) a Mongoose schema for `Developer` is detected, and it defines the `permissions` field in it. In the query, the use of other functions of the library can be observed, such as (`checkIsMDBDataMethod` and `checkFeatIsInObject`.

```
IMPORT javascript
IMPORT utils

FROM MethodCallExpr method
WHERE (
 checkIsMDBDataMethod(method.getMethodName())
 AND (
  checkPropAccess(
    method.getReceiver(),"Developer")
  OR checkExprIsCollectionMethod(
    method.getReceiver(),"Developer")
  OR checkExprIsCollectionObject(
    method.getReceiver(),"Developer")
 )
 AND (
  checkFeatIsInObject(
    method.getAnArgument(),"permissions")
  OR checkFeatIsInVarRef(
      method.getAnArgument(),"permissions")
  OR checkFeatIsInArray(
      method.getAnArgument(),"permissions")
  OR checkExprEvalsToString(
    method.getAnArgument(),"permissions")
 )
)
OR (
 checkIsMongooseExport(method, "Developer")
 AND (
  checkFeatIsInMongooseSchema(
    method.getArgument(1),"permissions")
  OR checkFeatIsInMongooseSchemaVarRef(
    method.getArgument(1),"permissions")
 )
)
SELECT method,
  "Developer.permissions: This feature has been
       deleted."
```

**Figure 9:** CodeQL *error* query generated for the `DELETE Developer::permissions` operation.

Please note that all CodeQL queries generated by the described process are able to match expressions that show specific string values (e.g., a feature name being modified) and also solve variables that are found on candidate expressions, recursively, until those variables are evaluated to a string value.

## 3.4. CodeQL Query Output

Once the CodeQL queries have been generated, developers can use the CodeQL engine to analyze folders with code files and apply the supplied queries, as shown in Figure 5. The engine outputs a SARIF file containing the results of the analysis. This file stores information such as the location of the analyzed files and the queries provided, and a section for each match found on those files. As shown in Figure 10, for each found match the file stores the rule that found it (*'softwaredev-1/000-op-delete-feature-developer-permissions-error'*), a custom message informing of the event (*'Developer.permissions: This feature has been deleted.'*), and the location of the match divided into two blocks: (i) the file in which it was found (*'models/Developer.js' in this example*), and (ii) the line and column inside the file content.

```
{
 "ruleId": "softwaredev-1/000-op-delete-feature-
      developer-permissions-error",
 "ruleIndex": 0,
 "rule": { "id": "...", "index": 0 },
 "message": {
  "text": "Developer.permissions: This feature has
       been deleted."
 },
 "locations": [{
  "physicalLocation": {
   "artifactLocation": {
    "uri": "models/Developer.js",
    "uriBaseId": "%SRCROOT%",
    "index": 0 },
   "region": {
    "startLine": 20,
    "startColumn": 16,
    "endColumn": 60
   }
  }
 }],
 "partialFingerprints": {
  "primaryLocationLineHash": "f31ed949860b8f:1",
  "primaryLocationStartColumnFingerprint": "15"
 }
}
```

**Figure 10:** A match found by CodeQL stored in SARIF format.

This file is then processed to produce a more readable output that helps developers to better understand which statements have been affected by SCOs and how to fix them. In our case, we opted for a format in which for each match entry a line such as *<file_path>(<line_number>): <message>* is generated.

## 4. Related Work

In this section, we will contrast our proposal with some of the most relevant works to co-evolve schema and code.

In [10], Carlo Curino et al. present the PRISM++ tool to automate the relational schema evolution. A set of Schema Modification Operators (SMOs) is defined to express schema changes in form of atomic operations, as well as a set of Integrity Constraint Modification Operators (ICMOs). For each SMO, a set of instructions are generated to change the relational schema, update the stored data, and rewrite queries under the given database constraints. Existing SQL queries are adapted by applying rewritten techniques and creating views.

While PRISM++ only tackles the schema evolution for relational systems, DB-Main is a database engineering environment built for relational databases and other data models prior to the emergence of NoSQL stores [22]. In addition to support schema evolution, DB-Main integrates utilities for reverse engineering, re-engineering, and maintenance. DB-Main is composed of three basic pillars: (i) The GER generic metamodel defined by extending ER, (ii) a transformation-based engineering process, and (iii) the maintenance of a history of schema changes.

Query rewriting for NoSQL databases, in particular MongoDB, has been addressed in [12] and [13]. In [13], Jerome Fink et al. describe a strategy to adapt queries to schema changes for polystores. This work is part of the Typhon project intended to offer a solution for designing polystores that can be constituted by relational and NoSQL databases. In Typhon, a family of languages was created for defining conceptual schemas (TyphonML), changing schemas, and querying databases (TyphonQL). The strategy proposed to adapt queries is the following. When TyphonQL queries are issued on a polystore whose schema has changed, they are rewritten according to the new schema. Query rewriting depends on the change operations applied, and four actionscan be applied on queries: (i) no modification is needed, (ii) query is modified to be valid, (iii) query is modified but a warning indicates that the result could be incorrect, (iv) and the query cannot be adapted. The last two actions produce messages to assist developers to change the code. A pattern matching scheme is used to map each triple <SCO, schema, query> to a handler function that produces the query adaptation.

Darwin is a schema evolution platform for NoSQL databases, which supports schema history extraction, several data update strategies, and query rewriting [12]. When lazy data migration is applied, several versions of the same entity can coexist in the database, and existing queries must be rewritten to be able to access all the schema versions. To achieve this, a schema history graph is calculated in form of a set of evolution operations that specifies the sequence of schema changes. Then, a query rewriting algorithm is performed which consider all the versions of the schema. Query rewriting applies forward and backward query rewriting (i.e., applies SCOs and their reverse equivalents to queries) to generate different sub-queries (one for each schema version), execute these sub-queries and union the obtained results.

Our work differs from previous ones in the following ways. PRISM++ is probably the most influential work related to the automation of schema evolution, but it is focused on relational databases. DB-Main is based on a generic metamodel and schema transformations, but NoSQL data models are not considered, and code updating is mainly focused on SQL queries. Also, GER and U-Schema are clearly different, and a taxonomy of changes was not defined. Regarding Typhon, U-Schema has a richer semantic than TyphonML which allows the structural variation and graph data models to be represented, among other differences, as discussed in [5]. Also, we are addressing different APIs instead of rewriting code of a query language specific to the Typhon platform, and we are searching code patterns in the use of APIs in order to reduce the effort. Finally, when compared to Darwin, this tool focuses on query rewriting [23], and our approach is based on a more complex unified metamodel than the data model underlying Darwin, which relies on a common interface used to access different stores. This also entails a richer taxonomy of operations.

## 5. Conclusions and Future Work

While the first version of the Orion engine supported the co-evolution of schema and data, the strategy here presented is a first step to assist developers to update code when schemas are updated. To this moment, the effort is being focused on the mapping of each SCO to the code patterns to be detected, the CodeQL queries to be generated, the changes to be applied, and the information to be provided to developers. As each mapping is elicited for a SCO, the corresponding rules are implemented and added to the model to text transformation and the analysis of the query result is extended.

As of future work, we consider the following: (i) Implementing the remaining taxonomy operations for MongoDB and JavaScript, (ii) expanding this implementation to additional databases to cover all the considered data models, (iii) integrating our approach with CI systems, and (iv) extending our current output and adding the possibility to apply changes directly to code by applying the generic rewrite approach shown in [24], where a generic metamodel was created to represent code.

The running example models as well as the generated queries used to illustrate our approach are available in a public GitHub repository.[3]

[3]https://github.com/modelum/code-scan.

## Acknowledgments

Work funded by the Spanish Ministry of Science and Innovation (project grant: PID2020-117391GB-I00).

## References

[1] M. Stonebraker, The case for polystores, ACM Sigmod Blog (2015). URL: https://wp.sigmod.org/?p=1629.

[2] P. Sadalage, M. Fowler, NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence, Addison-Wesley, 2012.

[3] J. Hainaut, et al, Database Evolution: the DB-Main Approach, in: Entity-Relationship Approach - ER'94, 13th Int. Conf. on the Entity-Relationship Approach, volume 881, Springer, 1994, pp. 112–131.

[4] Typhon Project, Hybrid Polystore Modelling Language (Final Version), Technical Report, University of L'Aquila, 2018. URL: https://4d97e142-6f1b-4bbd-9bbb-577958797a89.filesusr.com/ugd/d3bb5c_3394b40f9cb54bcbb873f2c4ea1f2298.pdf.

[5] C. J. F. Candel, D. S. Ruiz, J. J. G. Molina, A unified metamodel for nosql and relational databases, Information Systems 104 (2022) 101898. doi:10.1016/j.is.2021.101898.

[6] A. Hernández Chillón, D. Sevilla Ruiz, J. Garcia-Molina, Towards a Taxonomy of Schema Changes for NoSQL Databases: The Orion Language, in: Conceptual Modeling - ER 2021 40th Int. Conf. on Conceptual Modeling, St.John's, NL, Canada, volume 13011, 2021, pp. 176–185. doi:10.1007/978-3-030-89022-3_15.

[7] A. Hernández Chillon, M. Klettke, D. Sevilla Ruiz, J. Garcia-Molina, A Taxonomy of Schema Changes for NoSQL Databases, CoRR abs/2205.11660 (2022). URL: https://arxiv.org/abs/2205.11660.

[8] C. J. F. Candel, J. J. G. Molina, D. S. Ruiz, Skiql: A unified schema query language, CoRR abs/2204.06670 (2022). URL: https://doi.org/10.48550/arXiv.2204.06670. doi:10.48550/arXiv.2204.06670.

[9] J. Hainaut, The transformational approach to database engineering, in: Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers, 2005, pp. 95–143. URL: https://doi.org/10.1007/11877028_4. doi:10.1007/11877028_4.

[10] C. Curino, H. J. Moon, A. Deutsch, C. Zaniolo, Automating the database schema evolution process, in: The VLDB Journal, volume 22, 2013, pp. 73–98.

[11] L. Meurice, A. Cleve, Supporting schema evolution in schema-less nosql data stores, in: IEEE 24th Int. Conf. on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017, 2017, pp. 457–461.

[12] U. Störl, M. Klettke, Darwin: A Data Platform for Schema Evolution Management and Data Migration, in: DataPlat 2022: 1st International Workshop on Data Platform Design, Management and Optimization, 2022.

[13] J. Fink, M. Gobert, A. Cleve, Adapting Queries to Database Schema Changes in Hybrid Polystores, in: 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020, IEEE, 2020, pp. 127–131. URL: https://doi.org/10.1109/SCAM51674.2020.00019. doi:10.1109/SCAM51674.2020.00019.

[14] I. Holubová, M. Vavrek, S. Scherzinger, Evolution management in multi-model databases, Data & Knowledge Engineering 136 (2021) 101932. doi:10.1016/j.datak.2021.101932.

[15] P. P.-S. Chen, The Entity-Relationship Model: Toward a Unified View of Data, ACM Transactions on Database Systems 1 (1976) 9–36.

[16] A. Hernández, S. Feliciano, D. Sevilla Ruiz, J. García Molina, Exploring the Visualization of Schemas for Aggregate-Oriented NoSQL Databases, in: ER Forum 2017, 36th Int. Conf. on Conceptual Modeling (ER), 2017, pp. 72–85.

[17] J. Banerjee, W. Kim, H.-J. Kim, H. F. Korth, Semantics and Implementation of Schema Evolution in OO Databases, SIGMOD Rec. 16 (1987) 311–322.

[18] Codeql web page, 2023. URL: https://codeql.github.com/, accessed February 2023.

[19] Mongoose Web Page, 2017. URL: http://mongoosejs.com, accessed February 2023.

[20] S. bin Uzayr, N. Cloud, T. Ambler, Mongoose. In JavaScript Frameworks for Modern Web Development: The Essential Frameworks, Libraries, and Tools to Learn Right Now, Apress, 2019. doi:10.1007/978-1-4842-4995-6_9.

[21] K. Chodorow, M. Dirolf, MongoDB - The Definitive Guide: Powerful and Scalable Data Storage., O'Reilly, 2010.

[22] J.-M. Hick, J.-L. Hainaut, Strategy for database application evolution: The DB-MAIN approach, in: International Conference on Conceptual Modeling, Springer, 2003, pp. 291–306.

[23] M. Klettke, U. Störl, M. Shenavai, S. Scherzinger, NoSQL Schema Evolution and Big Data Migration at Scale, in: IEEE International Conference on Big Data, IEEE Computer Society, 2016.

[24] C. J. F. Candel, A Unified Data Metamodel for Relational and NoSQL databases: Schema Extraction and Query, Ph.D. thesis, Faculty of Informatics, University of Murcia, Murcia, Spain, 2022.