Model-driven Development of OData Services: An Application to Relational Databases

Hamza Ed-douibi
UOC
Barcelona, Spain
hed-douibi@uoc.edu

Javier Luis Cánovas Izquierdo
UOC
Barcelona, Spain
jcanovasi@uoc.edu

Jordi Cabot
ICREA - UOC
Barcelona, Spain
jordi.cabot@icrea.cat

Abstract—**Open Data Protocol (OData) is a protocol to facilitate the publication and consumption of queryable and interoperable data-driven online services. OData is based on the use of RESTful APIs derived from a data model plus a URL-based query language to identify and filter the data described in such model. Due to its maturity and ease of use for end-users and client applications, OData has become the natural choice to publish datasets online. Still, creating OData services is a tedious and time-consuming task, since data providers should (1) represent their data models in OData format, (2) implement the business logic to transform OData requests to SQL statements (or the target storage technology of choice), and (3) de/serialize the exchanged messages conforming to the OData protocol. This paper presents a model-based approach aimed at (semi)automating all these steps. From an initial UML class diagram, we derive all the artifacts required to have an OData service up and running on top of a relational database conforming to the model definition. A prototypical implementation of the approach is provided.**

Index Terms—**OData, Web Service, MDE, MDA, UML**

I. Introduction

Open Data Protocol (OData)[1] is a data access protocol to create Web services with query and update capabilities in a simple and standard way, thus allowing developers to easily expose and access information from a variety of data sources such as relational databases, file systems and content management systems. In the last years, OData has evolved to become the natural choice for creating data-centric Web services, specially for Open Data initiatives aiming at facilitating the access to information using Web services rather than Resource Description Framework (RDF). As a result, many service providers have integrated OData in their solutions (e.g., SAP, IBM WebSphere or JBoss Data Virtualization). The current version of OData (version 4.0) has been approved as an OASIS standard [1].

OData enables the creation of data-centric Web services, where URL-accessible resources are defined according to an Entity Data Model (EDM) and can be queried by Web clients using standard HTTP messages. EDM, which borrows some concepts from the Entity Relationship (ER) model, defines an abstract conceptual model of the data exposed by the OData service [2]. Thus, relational databases are usually the most common storage solution for OData services. The protocol also defines a URL-based query language sharing some similarities with SQL that facilitates clients to query the data described by the EDM [3].

While consuming an OData service is easy, the task of creating OData services is tedious and time-consuming, specially for relational databases where extra effort is needed to align the service with the database query capabilities. Developers should first represent their data models according to the EDM format and then provide support to query and update the data by implementing the business logic to resolve URLs using the OData query language and transforming such queries into SQL statements. Furthermore, a de/serialization mechanism is required to exchange messages with clients conforming to the OData protocol (using OData JSON and Atom formats).

There are some Software Development Kits (SDKs) for developing OData applications (e.g., *RESTier*[2], *Apache Olingo*[3], *SDL OData Frameworks*[4]) and commercial tools for exposing OData services from already existing data sources (e.g., *Cloud Drivers*[5], *OData server*[6], *Skyvia Connect*[7]), but they still require advanced knowledge about OData to implement the business logic of the service, and provide limited support for the OData specification, respectively. Other tools such as *simple-odata-server*[8] and *JayDATA*[9] allow generating a basic OData server from both an entity model expressed in OData format and the corresponding database, but they only cover a subset of the OData protocol.

Model-Driven Engineering (MDE) is a methodology that focuses on using models to raise the level of abstraction and automation in software development [4]. MDE relies on models and model transformations for the specification and generation of software applications, thus hiding the complexity of the target technology. Even though many MDE approaches target the generation of Web applications in general [5–7], to the best of our knowledge, there is no current support for the specification and generation of OData services.

[1]http://www.odata.org/

[2]https://github.com/OData/RESTier
[3]https://olingo.apache.org/
[4]https://github.com/sdl/odata
[5]http://www.cdata.com/odata/
[6]https://rwad-tech.com/
[7]https://skyvia.com/connect/
[8]https://github.com/pofider/node-simple-odata-server
[9]https://github.com/jaystack/jaydata

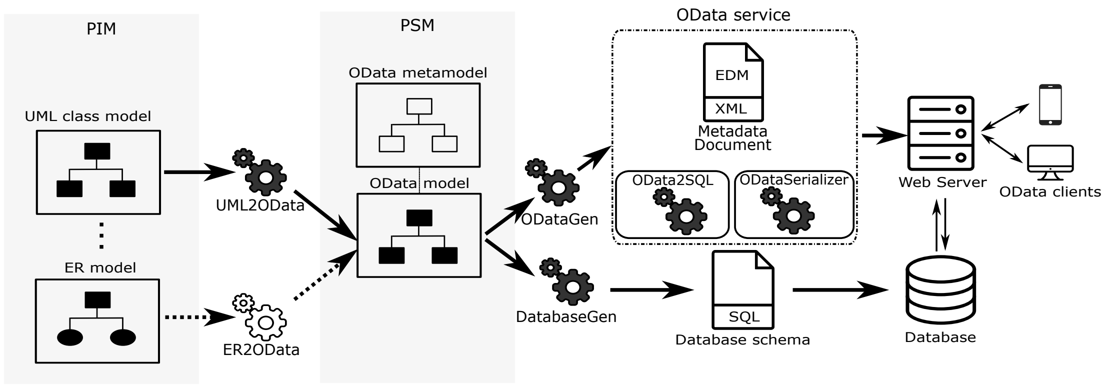


Fig. 1. Overview of the approach.

In this paper, we propose to combine the benefits of MDE and OData by providing a model-based approach to (semi)automate the generation of ready-to-deploy OData services. From an initial Unified Modeling Language (UML) class diagram, we derive all the artifacts required to have an OData service up and running on top of a relational database conforming to the model definition, including the transformation of OData requests to SQL queries and complying with OData protocol. Our approach relies on an OData pivot metamodel, which is used to represent and generate OData services; thus allowing us to leverage on the plethora of existing modeling tools and therefore enabling our approach to use other input models or target technologies.

The rest of this paper is structured as follows. Section II describes our approach and Section III shows the running example used along the paper. Section IV presents OData metamodel and how we derive its instances from UML models. Sections V and VI show the database schema generation process and the OData service generation process, respectively. Section VII describes the tool support. Section VIII presents the related work. Finally, Section IX concludes the paper and presents the future work.

II. Approach

We propose a model-driven approach where OData models drive the generation of OData services using a relational database as storage solution. These OData models could be specified directly but typically they will be derived from an input UML or ER model describing the domain. Figure 1 shows an overview of our approach.

On the left-hand side of Figure 1, and following the Model-Driven Architecture (MDA) terminology of the Object Management Group (OMG), we have the UML and ER models at the Platform-Independent Model (PIM) level while the OData metamodel would belong to the Platform-Specific Model (PSM) level as a refinement of the previous one. The mapping between the PIM and PSM level is rather straightforward, as we will show. In this paper we will focus on the UML to OData path (see *UML2OData* transformation) but a similar approach could be used for ER models (see *ER2OData* transformation).

On the right-hand side of Figure 1, we see how OData models are used to generate: (1) an OData service wrapped in a Web application to be deployed in a server (see *ODataGen* transformation); and (2) the corresponding database schema to initialize the database (see *DatabaseGen* transformation). The OData service includes: (a) the OData metadata document, which defines the Entity Data Model (EDM) for the data exposed by the service [8]; (b) the logic to transform OData requests into SQL statements according to the query language defined by OData protocol [3] (see *OData2SQL* component); and (c) an OData serializer, which defines the serialization mechanism according to OData JSON format [9] and OData Atom format [10] (see *ODataSerializer* component).

OData defines three levels of conformance for an OData Service, namely: minimal, intermediate and advanced (cf. OData protocol [1], Section 13). Each level defines a set of requirements and recommendations that a service should fulfill in order to conform to this level. The OData service generated with our approach fully conforms to the OData Intermediate Level and partially to the OData Advanced Conformance Level, as we will present later.

The elements generated in each step could be customized by the user in order to either include other details not captured in our generation process or remove extra generated elements. For instance, an OData model generated from a PIM model could be enriched by adding other OData elements (e.g., OData annotations) or remove unwanted generated elements (e.g., an OData entity type generated from an unwanted UML class). Also, the generated application could be refined in order to customize the OData service or integrate other web functionalities not related to OData such as authentication.

III. Running Example

To illustrate our approach, we use as running example the UML class diagram shown in Figure 2 representing a data model to manage online stores. This example is inspired by the official reference example of OData[10]. The model includes two classes, namely: *Product*, which represents products; and *Supplier*, which represents the supplier of a product. The

[10]http://services.odata.org/V4/OData/OData.svc/$metadata

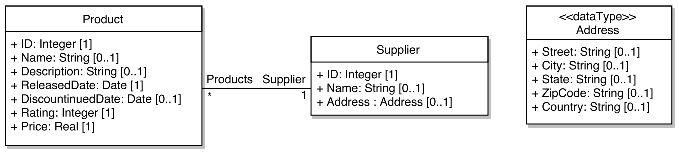


Fig. 2. UML model of the running example.

address of a supplier is defined using the data type *Address*. The bidirectional association between products and suppliers allows navigating from a product to a supplier (the association end *supplier*), and from a supplier to a list of products (the association end *products*).

Given this model, our approach generates a ready-to-deploy OData Service exposing the OData metadata document representing the data model and serving client requests for both querying and update data, which requires transforming OData requests to SQL statements and presenting the data according to OData protocol (i.e., OData JSON [9] and OData Atom [10] formats).

The OData metadata document is expressed using the Conceptual Schema Definition Language (CSDL) [8]. Listing 1 shows an excerpt of the generated metadata document for the data model shown in Figure 2. The `Schema` element describes the entity model exposed by the OData Web service and includes the entity types `Product` and `Supplier`, and the complex type `Address`. Each type includes properties and navigation properties to describe attributes and relationships, respectively. The `Schema` element includes also an `EntityContainer` element defining the entity sets exposed by the service and therefore the entities that can be accessed. Web clients use this document to understand how to query and interact with the service. For instance, the request `GET http://host/service/Products?$filter=Price le 2.6` including the `$filter` option, should retrieve the list of *products* having the price less or equals to 2.6. Listing 2 shows the result of this request in OData JSON format.

Next sections will describe how our approach can be used to go from the original model to the deployed OData services following a model-driven approach.

IV. Specification of OData services

This section describes the specification of OData services including (1) the OData metamodel and (2) the creation of models conforming to this metamodel from UML models.

A. The OData Metamodel

There are two primary ways to formalize domain-specific knowledge, either by refining/extending an existing modeling language (e.g., using a UML profile), or creating a metamodel describing a domain-specific language [11]. In a previous work we proposed a UML profile for OData [2], while for the context of this paper we propose a metamodel for OData to ease the definition of our MDA-based approach and implementation of the involved transformations. The OData metamodel is aligned with the OData CSDL specification [8],

Listing 1. A simple OData Metadata Documents for the products service.

```
<edmx:Edmx xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx" Version="4.0">
  <edmx:DataServices>
    <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="com.example.ODataDemo" Alias="ODataDemo">
      <EntityType Name="Product">
        <Key><PropertyRef Name="ID"/></Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Name" Type="Edm.String"/>
        <Property Name="Description" Type="Edm.String"/>
        <Property Name="ReleasedDate" Type="Edm.DateTimeOffset" Nullable="false"/>
        <Property Name="DiscontinuedDate" Type="Edm.DateTimeOffset"/>
        <Property Name="Rating" Type="Edm.Int16" Nullable="false"/>
        <Property Name="Price" Type="Edm.Double" Nullable="false"/>
        <NavigationProperty Name="Supplier" Type="ODataDemo.Supplier" Partner="Products"/>
      </EntityType>
      <EntityType Name="Supplier">
        <Key><PropertyRef Name="ID"/></Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Name" Type="Edm.String"/>
        <Property Name="Address" Type="ODataDemo.Address"/>
        <NavigationProperty Name="Products" Type="Collection(ODataDemo.Product)" Partner="Supplier" />
      </EntityType>
      <ComplexType Name="Address">
      <Property Name="Street" Type="Edm.String"/>...
      </EntityType>
      <EntityContainer Name="DemoService">
        <EntitySet Name="Products" EntityType="ODataDemo.Product">
          <NavigationPropertyBinding Path="Supplier" Target="Suppliers"/>
        </EntitySet>
        <EntitySet Name="Suppliers" EntityType="ODataDemo.Supplier">
          <NavigationPropertyBinding Path="Products" Target="Products"/>
        </EntitySet>
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

which defines the main concepts to be exposed by any OData service, thus facilitating later the generation of OData metadata documents (as we will describe in Section VI-A).

Figure 3 shows an excerpt of the OData metamodel. The top part of the metamodel comprises the `ODService` element, which includes a set of schemas (i.e., `schemas` reference). One or more schemas define the data model of an OData service. A schema is represented by the `ODSchema` element which includes the namespace of the schema (e.g., `com.example.ODataDemo`) and an alias for the schema namespace (e.g., `ODataDemo`). It includes also references to the data structures defined by the schema which comprise enumerations (i.e., `enumTypes` reference), complex types (i.e., `complexTypes` reference) and entity types (i.e., `entityTypes` reference). All data structures in the metamodel are subtypes of the `ODType` element which describes the structure of an abstract data type.

An enumeration is represented by the `ODEnumType` ele-

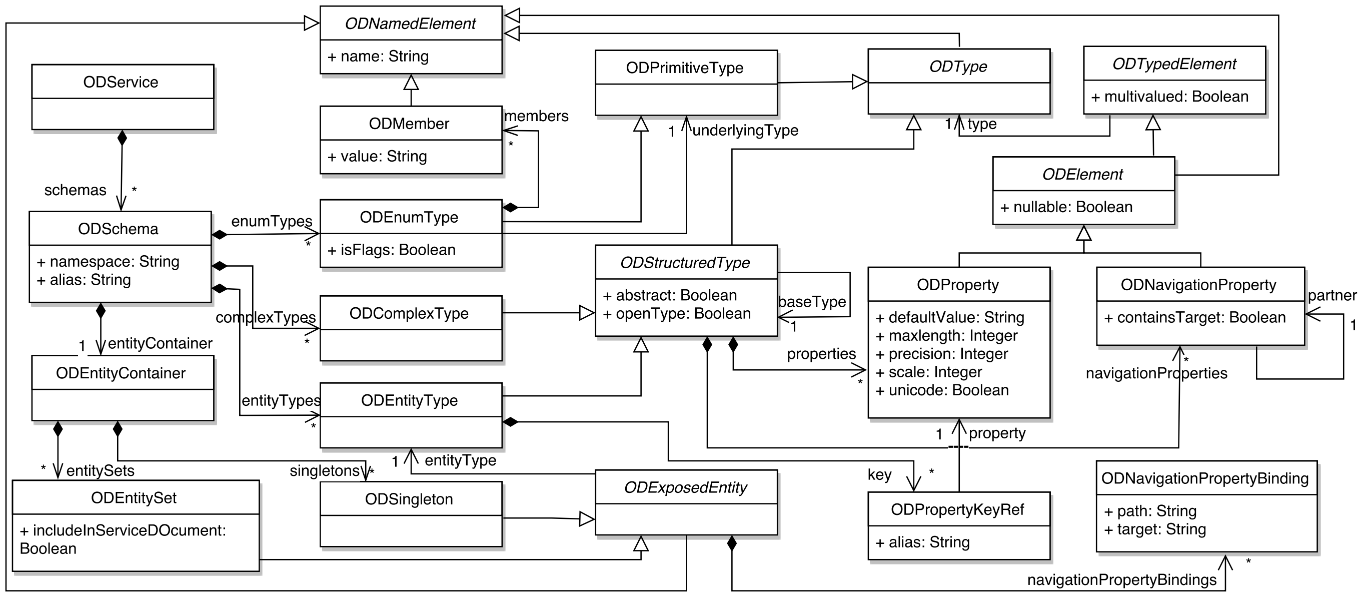


Fig. 3. An excerpt of OData metamodel.

Listing 2. An example of collection of *products* in OData JSON format.

```
{
    "@odata.context": "$metadata#Products",
    "value": [
        {
            "ID": 1,
            "Name": "Milk",
            "Description": "Fresh milk",
            "ReleasedDate": "1992-01-01",
            "DiscountinuedDate": null,
            "Rating": 4,
            "Price": 2.40
        },
        {
            "ID": 2,
            ...
            "Price": 2.25
        }
    ]
}
```

ment which is a subtype of the `ODPrimitiveType` element and includes a name (inherited from `ODNamedElement`), an attribute indicating whether the enumeration allows multi-selection (i.e., `isFlags` attribute), an underlying primitive type (i.e., `underlyingType` reference), and a list of members (i.e., `members` reference). The element `ODMember` defines the options for the enumeration type and includes a name (inherited from `ODNamedElement`) and a value.

Entity types are named structured types with a key, while complex types are keyless named structured types. Entity types and complex types are represented by the `ODEntityType` and `ODComplexType` elements, respectively. Both elements are subtypes of the `ODStructuredType` abstract element which represents a structured type. The `ODStructuredType` element is a subtype of the `ODType` element and includes a name (inherited from `ODNamedElement`), an attribute indicating whether the structural type cannot be instantiated (i.e., `abstract` attribute), and an attribute indicating whether undeclared properties are allowed (i.e., `openType` property[11]). A structured type is composed of structural properties (i.e., `properties` reference) and navigation properties (i.e., `navigationProperties` reference) and may define a base type (i.e., `baseType` reference). Additionally, the `ODEntityType` element includes a key (i.e., `key` reference) which indicates the properties identifying an entity (i.e., `property` reference of the element `ODPropertyKeyRef`) and an alias name for each property.

The `ODProperty` and `ODNavigationProperty` elements represent a structural property and a navigation property, respectively. While the `ODProperty` element defines an attribute of a structured type, the element `ODNavigationProperty` defines an association between two entity types. Both elements are subtypes of the `ODElement` abstract element which defines the common features of structural properties. This element includes a name (inherited from `ODNamedElement`), an attribute indicating whether the element can be null (i.e., `nullable` property), a type (i.e., `type` reference inherited from `ODTypedElement`), and a cardinality (i.e., the `multivalued` property inherited from `ODTypedElement`). Additionally, the `ODProperty` element includes several attributes to provide additional constraints about the value of the structural property (e.g., `maxLength` and `precision` properties). The `ODNavigationProperty` element, on the other hand, includes a containment attribute (i.e., `containsTarget` reference) and an opposite navigation property (i.e., `partner` reference).

[11]Open types entities allows clients to persist additional undeclared properties.

TABLE I
UML TO ODATA MODEL TRANSFORMATION RULES.

REF.	SOURCE ELEMENTS	CONDITIONS	TARGET ELEMENTS	INITIALIZATION DETAILS

1	`c: Class`	-	`et: ODEntityType`
			`es: ODEntitySet`
				- `et.name = c.name`
				- **if** `c.abstract = true` **then** `et.abstract ← true`
				- **if** `c.generalizations` contains a class cc **then** `et.baseType ← t` where `t` is the corresponding `EntityType` of `cc`
				- `et.properties ←` *(cf. rules to transform attributes, rows 3 and 4)*
				- `et.navigationProperties ←` *(cf. rule to transform navigable association ends, i.e., row 5)*
				- `es.name ←` the plural form of `c.name`
				- `es.entityType ← et`
				- `es.navigationPropertyBindings ←` *(cf. rule to transform navigable association ends, i.e., row 5)*

2	`dt: DataType`	-	`ct: ODComplexType`	- `ct.name ← dt.name` - **if** `ct.abstract = true` **then** `dt.abstract ← true` - **if** `dt.generalizations` contains a data type `dd` **then** `ct.baseType ← t` where `t` is the corresponding `ODComplexType` of `dd` - `ct.properties ←` *(cf. rules to transform attributes, i.e., rows 3 and 4)*
3		`p` is a class attribute or a data type attribute	`op: ODProperty`	- `op.name ← p.name` - `op.type ← t` where `t` is the corresponding type of the attribute - **if** `p` is multivalued **then** `op.multivalued ← true`
4	`p: Property`	`p` is a class attribute marked as `ID`	`pk: ODPropertyKeyRef`	- `pk.property = op`

5		`p` is a navigable association end	`np: ODNavigationProperty`
			`npb: ODNavigationPropertyBinding`
				- `np.name ← p.name`
				- `np.type ← t` where `t` is the corresponding entity type of `p.type`
				- **if** `p.aggregation = Composite` **then** `np.containsTarget ← true`
				- **if** `p` is multivalued is **then** `np.multivalued ← true`
				- `npb.path ← p.name`
				- `npb.target ← t.name` where `t` is the corresponding entity set of `p.type`

6	`e: Enumeration`	-	`oe: ODEnumType`	- `oe.name ← e.name` - `oe.members ←` *(cf. rule to transform literals, i.e., row 7)*
7	`el: EnumerationLiteral`	-	`om: ODMember`	- `om.name ← el.name`

The `ODSchema` element includes also an entity container (i.e., `entityContainer` reference) defining the entity sets and singletons queryable and updatable by the service. The `ODEntityContainer` element defines an entity container and includes a set of entity sets (i.e., `entitySets` reference) and singletons (i.e., `singleton` reference). An entity set allows addressing a collection of entities, while a singleton allows addressing a single entity directly from the entity container. The two concepts are materialized by the `ODEntitySet` and `ODSingleton` elements, respectively. Both elements are subtypes of the `ODExposedEntity` abstract element which includes a reference to the target entity type (i.e., `entityType` reference), a name (inherited from `ODNamedElement`), and a set of navigation properties bindings (i.e., `navigationPropertyBindings` reference).

Apart from the elements presented in this section, the OData metamodel includes other OData concepts which could not be presented for the sake of simplicity. Thus, the metamodel includes elements to define annotations and vocabularies which provide an extension mechanism to add additional characteristics or capabilities of OData elements. The complete metamodel is available in our Github repository [12].

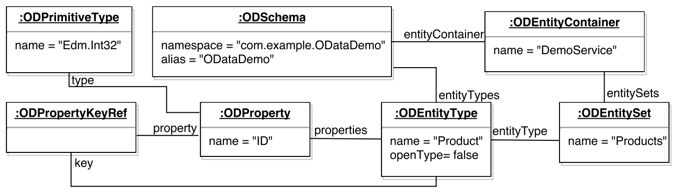


Fig. 4. An excerpt of the generated OData model for the running example.

B. *Mapping UML to OData Models*

OData models can be automatically derived from UML models by means of a model-to-model transformation (see *UML2OData* transformation in Figure 1). In this paper we rely on plain UML models to generate OData models, thus no knowledge of OData is to be required. A similar approach, relying on the UML profile for OData [2], could also be followed to drive a custom transformation for enriched UML models.

Table I shows a subset of the main transformation rules[12] from UML metamodel elements to OData metamodel

[12]The full transformation is available at our repository [12].

elements where the second column displays the source UML elements, third column shows the conditions to trigger the transformation, firth column shows the created/updated OData elements, and the last column shows the initialization values for the OData elements. Note that instances of the elements `ODEntityType`, `ODComplexType`, and `ODEnumType` are added to the element `ODSchema` once they are created, and likewise instances of `ODEntitySet` are added to the element `ODEntityContainer`. Furthermore, each instance of the elements `ODProperty`, `ODNavigationProperty`, `ODNavigationPropertyBinding` are added to its corresponding `ODEntityType`, `ODComplexType` element, and likewise each instance of `ODNavigationPropertyBinding` is added to its corresponding `ODEntitySet`. Figure 4 shows an excerpt of the generated OData model for the `Product` entity (with only the `ID` attribute) of our running example (see Figure 2).

V. Database Schema Generation

OData is designed to work on a variety of data stores. In particular, the protocol does not necessarily assume a relational data model. In this paper we defined an algorithm to generate a relational database schema from an OData data model, which we describe in this section (see *DatabaseGen* transformation in Figure 1). This algorithm is heavily inspired by the typical transformation rules to derive database schemas from UML models (e.g., GenMyModel[13], UMLtoX[14]) or ER models (e.g. see Fidalgo et al. [13], ER2SQL[15]).

Algorithm 1 illustrates the OData data model to database schema generation process. As can be seen, the algorithm takes as input an instance of `ODSchema` and returns a Data Definition Language (DDL) script representing the data model. The first part of the algorithm (i.e., from line 1 to line 41) iterates over the contained entity types and complex types then generates a `CREATE` command for each element not having a super type (i.e., `baseType = null`). The algorithm adds an extra column `id` for each complex type to define a primary key (cf. line 5) and an extra column `discriminator` for each complex or entity type having subtypes to identify the concrete type of the element (cf. line 8). Furthermore, for each single valued property or navigation property the algorithm generates a column statement (cf. line 12). Moreover, the algorithm generates a `CREATE` command for each multivalued property (cf. line 21) and many to many navigation property (cf. line 31). The second part of the algorithm (i.e., from line 42 to line 52) iterates over the navigation properties of each entity type and complex type then generates an `ALTER TABLE` command to declare a foreign key. The algorithm relies on the functions TABLENAME, COLUMNTYPE, COLUMNNAME, GETNULL and REFERENCE (see footnotes on Algorithm 1). Note that for the sake of simplicity, the algorithm assumes that the key of each entity type is represented by a single property. Listing 3 shows an excerpt of the DDL script to create the database schema corresponding to the running example after applying the algorithm.

Algorithm 1 DDL schema generation.

```
Input:
    s where s is an instance of ODSchema
Output:
    q where q is the DDL script of the database
 1: $S \leftarrow s.entityTypes \cup s.complexTypes$
 2: for $i = 1$ to $S.length$ do
 3:     if $S[i]$ does not have a super type then
 4:         $q \leftarrow q+$ "CREATE TABLE" + TABLENAME($S[i]$) + "("
 5:         if $S[i]$ is instance of ODComplexType then
 6:             $q \leftarrow q+$ "id INT not null,"
 7:         end if
 8:         if $S[i]$ is a super type then
 9:             $q \leftarrow q+$ "discriminator VARCHAR(255),"
10:         end if
11:         $P \leftarrow S[i].properties \cup S[i].navigationProperties$
12:         for $j = 1$ to $P.length$ do
13:             if $P[j]$ is single valued then
14:                 $q \leftarrow q+$ COLUMNNAME($P[j]$)
15:                 +COLUMNTYPE($P[j]$) + GETNULL($P[j]$) + ","
16:             end if
17:         end for
18:         $q \leftarrow q+$ "PRIMARY KEY (" + PRIMARY($S[i]$) + "));"
19:     end if
20:     for $j = 1$ to $S[i].properties.length$ do
21:         if $S[i].properties[j]$ is multivalued then
22:             $q \leftarrow q+$ "CREATE TABLE"
23:             +TABLENAME($S[i].properties[j]$)
24:             +"(id INT not null," + COLUMNNAME($S[i].properties[j]$)
25:             +COLUMNTYPE($S[i].properties[j]$) +","
26:             +TABLENAME($S[i]$) + "_id" + PRIMARYTYPE($S[i]$) +","
27:             +"PRIMARY KEY (id));"
28:         end if
29:     end for
30:     for $j = 1$ to $S[i].navigationProperties.length$ do
31:         if $S[i].navigationProperties[j]$ is many to many then
32:             $q \leftarrow q+$ "CREATE TABLE"
33:             +TABLENAME($S[i].navigationProperties[j]$) + "("
34:             +"id INT not null,"
35:             +TABLENAME($S[i]$) + "_id," + PRIMARYTYPE($S[i]$)+","
36:             +TABLENAME($S[i].navigationProperties[j].partner$)+"_id,"
37:             +PRIMARYTYPE($S[i].navigationProperties[j].partner$)+","
38:             +"PRIMARY KEY (id));"
39:         end if
40:     end for
41: end for
42: for $i = 1$ to $S.length$ do
43:     for $j = 1$ to $S[i].navigationProperties.length$ do
44:         if $S[i].navigationProperties[j]$ is single valued then
45:             $q \leftarrow q+$ "ALTER TABLE" + TABLENAME($S[i]$)
46:             +"ADD FOREIGN KEY"
47:             +COLUMNNAME($S[i].navigationProperties[j]$)
48:             +"REFERENCES" + REFERENCE($S[i].navigationProperties[j]$)
49:             +";"
50:         end if
51:     end for
52: end for
```

TABLENAME gets a table name according database recommendations representing the input element.

COLUMNTYPE gets a database primitive type representing the type of the input element.

COLUMNNAME gets a column name according database recommendations representing input element.

GETNULL generates `NOT NULL` statement if the input parameter is required.

PRIMARY gets the key column of the input element

PRIMARYTYPE gets the type of the key column of the input element

REFERENCE generates the target of `REFERENCES` statement of the input element.

VI. OData Service Generation

In this section we describe the generation process of OData services from OData models (see the *ODataGen* transforma-

[13]https://www.genmymodel.com/

[14]https://github.com/jcabot/UMLtoX

[15]http://er2sql.sourceforge.net/

TABLE II
EXAMPLE OF ODATA REQUESTS AND THE CORRESPONDING SQL STATEMENTS.

REF	HTTP METHOD	DESCRIPTION	RESOURCE PATH EXAMPLE	SQL QUERY
1	GET	Request a collection of entities	`GET http://host/service/Products`	`SELECT * FROM product p`
2		Request a single entity by ID	`GET http://host/service/Products(1)`	`SELECT * from product p WHERE p.id = 1`
3		Request an individual property	`GET http://host/service/Products(1)/Price`	`SELECT p.price from product p WHERE p.id = 1`
4		Request an entity collection by following a navigation from an entity to another related entity	`GET http://host/service/Suppliers(1)/Products`	`SELECT p.* from product p JOIN supplier s on p.supplier_id = s.id WHERE s.id = 1`
5	POST	Create a new entity	`POST http://host/service/Products { "ID": 1; "Name": "Milk", "Description": "Fresh milk",...}`	`INSERT INTO product (id, name, description, ...) VALUES (1, 'Milk', 'Fresh milk', ...)`
6	PATCH	Update an entity	`PATCH http://host/service/Products(1) { "Description": "Very fresh milk" }`	`UPDATE product SET description = 'Very fresh milk' WHERE id = 1`
7	PUT	Update a navigation property	`PUT http://host/service/Products(1)/Supplier/$ref { "@odata.id": "http://host/service/Suppliers(2)" }`	`UPDATE product SET supplier_id = 2 WHERE id = 1`
8	DELETE	Delete an entity	`DELETE http://host/service/Products(1)`	`DELETE FROM product WHERE id = 1`

Listing 3. A simple DDL file of the running example.

```
CREATE TABLE product (
    id INT not null,
    name VARCHAR(255),
    description VARCHAR(255),
    releaseddate DATE not null,
    discountinueddate DATE,
    rating INT not null,
    price DECIMAL (10,2) not null,
    supplier_id INT,
    PRIMARY KEY (id));
CREATE TABLE supplier (
    id INT not null,
    name VARCHAR(255)
    address_id INT,
    PRIMARY KEY (id));
CREATE TABLE address (
    id INT not null,
    street VARCHAR(255),...
    PRIMARY KEY (id));
ALTER TABLE product
ADD FOREIGN KEY (supplier_id) REFERENCES supplier(id);
ALTER TABLE supplier
ADD FOREIGN KEY (address_id) REFERENCES address(id);
```

tion in Figure 1). Our process includes the generation of (1) the metadata document, (2) the mapping between OData requests and SQL statements (see *OData2SQL* component in Figure 1), and (3) the de/serialization process (see *ODataSerializer* component in Figure 1).

A. OData Metadata Document Generation

This process transforms an OData model into an OData metadata document by means of a model-to-text transformation. This document helps clients discover the data schema exposed by the service and therefore build OData queries.

The generation process is nearly straightforward as our metamodel follows the OData CSDL specification and only special attention had to be paid when generating references among elements. Thus, the transformation iterates over OData model elements and generates their XML representation (e.g., `Schema` for the element `ODSchema`, `EntityContainer` for the `ODEntityContainer`, etc.) taking into account its specification (e.g., name, type, etc.). The resulting document is an XML file represented using the CSDL language [8] and can be retrieved by appending `$metadata` to the root URL of an OData service. An example of this file was previously shown in Listing 1.

B. OData Requests to SQL Statements Transformation

The OData specification defines standard rules to query data via HTTP `GET` requests and perform data modification actions via HTTP `POST`, `PUT`, `PATCH`, and `DELETE` requests. A URL of an OData request has three parts [3]: (1) the service root URL, which identifies the root of an OData service; (2) a target resource path, which identifies a resource to query or update (e.g., products, a single product, supplier of a product); and (3) a set of query options.

$$\underbrace{\texttt{http://host/service}}_{\text{service root URL}}/\underbrace{\texttt{Suppliers(1)/Products}}_{\text{target resource path}}?\underbrace{\texttt{\$top=2\&orderby=Name}}_{\text{query options}} \quad (1)$$

To transform OData requests to SQL statements we consider the HTTP method, which specifies if the request is either a query or a data modification action; the resource path, and the query options part. In the following we describe how these elements drive the SQL statement generation process, in particular, how we deal with the target resource paths, query options, and data modification actions.

1) Target resource path URL transformation: The target resource path in an OData request can address (1) a collection of entities, (2) a single entity, (3) and a property, which we will illustrate by means of examples. Table II shows a set of requests relying on our running example for all the CRUD operations. The second column shows the used HTTP methods. The third column explains the type of the request. The fourth and fifth columns show an example of the OData request including the resource path and request body, and the corresponding SQL query to the database, respectively.

Example 1 illustrates a request to access to a collection of entities (i.e., collection of products), which is transformed into a `SELECT` SQL statement for the corresponding table of the entity exposed by the addressed entity set. Example 5 also illustrates a request to add a new entity into the target collection of entities.

Example 2 shows how a single entity can be accessed by adding the entity key as path segment (i.e., the product of ID 1), which requires adding a `WHERE` clause to the `SELECT` statement. Likewise, examples 6 and 8 illustrate how to update and delete a single entity, respectively.

Example 3 shows how to access an entity property (i.e., the price of the product), which requires adding the corresponding column name to the `SELECT` statement.

Finally, example 4 illustrates a request to access to a collection of entities by navigating from an entity to another one (i.e., the collection of products of a specific supplier), which requires adding one or more `JOIN` clauses to the `SELECT` statement depending on the cardinalities of the navigation properties (i.e., one to many, many to many) and the hierarchy of the resource. Example 7 also illustrates how to update a navigation property.

To perform the transformation of the target resource path into the corresponding SQL statement we devised Algorithm 2. The algorithm takes as input the set of entity types referenced in the URL, the set of the properties used to navigate from one entity to the other in the path, a set of path segments to specify the key of a particular entity and the name of the final property to retrieve. Last two parameters are optional.

Algorithm 2 Resource path URL transformation.

```
Input:
    $E = \{E_1, E_2, ..., E_n\}$ where $E_i$ is the entity at the index $i$ and $n$ is the
    total number of entities
    $N = \{N_{1,2}, N_{2,3}, ..., N_{n-1,n}\}$ where $N_{i-1,i}$ is the navigation property
    from $E_{i-1}$ to $E_i$.
    $x = \{x_1, x_2, ..., x_n\}$ where $x_i$ is the key of the entity $E_i$ if presented in
    the URL or $\varnothing$ otherwise
    $p$ where $p$ corresponds to the name of a particular property to retrieve or $\varnothing$
Output:
    $q$ where $q$ is an SQL query representing the resource of the input
 1: for $i = 1$ to $n$ do
 2:     $T_i \leftarrow$ TABLENAME($E_i$)
 3:     $A_i \leftarrow$ TABLEALIAS($E_i$)
 4: end for
 5: if $p <> \varnothing$ then
 6:     $q \leftarrow$ +"SELECT" + $A_n$ +"." + COLUMNNAME($p$)
 7:     + "FROM" + $T_n$ + $A_n$
 8: else
 9:     $q \leftarrow$ + "SELECT" + $A_n$ + ".* FROM" + $T_n$ + $A_n$
10: end if
11: $i \leftarrow n$
12: while $i <> 1$ do
13:     if $N_{i-1,i}$ is many to one then
14:         Let $C_{i-1}^{i\,fk}$ be the column representing the foreign key of $T_i$ in $T_{i-1}$
    and $C_i^{pk}$ the primary key of $T_i$
15:         $q \leftarrow q$ + "JOIN" + $T_{i-1}$ + $A_{i-1}$ + "ON" + $A_{i-1}$ +"." + $C_{i-1}^{i\,fk}$
16:         + "=" + $A_i$ + "." + $C_i^{pk}$
17:     else
18:         if $N_{i-1,i}$ is one to many then
19:             Let $C_i^{i-1\,fk}$ be the column representing the foreign key of $T_{i-1}$
    in $T_i$ and $C_{i-1}^{pk}$ the primary key of $T_{i-1}$
20:             $q \leftarrow q$ + "JOIN" + $T_{i-1}$ + $A_{i-1}$ + "ON" + $A_i$ + "."
21:             + $C_i^{i-1\,fk}$ + "=" + $A_{i-1}$ + "." + $C_{i-1}^{pk}$
22:         else
23:             if $T_{i,i-1}$ is many to many then
24:                 Let $J_{i,i-1}$ be the association table between $T_i$ and $T_{i-1}$,
    $A_{i,i-1}$ the alias name $J_{i,i-1}$, $C_{i,i-1}^{i\,fk}$ the column representing the foreign
    key of $T_i$ in $J_{i,i-1}$, and $C_{i,i-1}^{i-1\,fk}$ the column representing the foreign key of
    $T_{i-1}$ in $J_{i,i-1}$
25:                 $q \leftarrow q$ + "JOIN" + $J_{i,i-1}$ $A_{i,i-1}$ + "ON" + $A_{i,i-1}$ +"."
26:                 + $C_{i,i-1}^{i\,fk}$ "=" + $A_i$ + "." + $C_i^{pk}$ "JOIN" + $T_{i-1}$ $A_{i-1}$
27:                 + "ON" + $A_{i,i-1}$" + "." + $C_{i,i-1}^{i-1\,fk}$ "=" + $A_{i-1}$ + "."
28:                 + $C_{i-1}^{pk}$
29:             end if
30:         end if
31:     end if
32:     $i \leftarrow i - 1$
33: end while
34: $q \leftarrow q +$ "WHERE"
35: for $i = 1$ to $n$ do
36:     $q \leftarrow q + A_e$ + "." + $C_e^{pk}$ + "=" + $x_i^e$
37:     if HASNEXT($x_i^e$) then
38:         $q \leftarrow q +$ "AND"
39:     end if
40: end for
```

TABLENAME gets a table name according to database recommendations representing the input element.

TABLEALIAS gets an alias name for the input element.

COLUMNNAME gets a column name according database recommendations representing input element.

HASNEXT returns `true` if there is a key in the queue and `false` otherwise.

The algorithm is divided into three parts. The first part (i.e., from line 1 to line 4) retrieves the database tables corresponding to each entity. The second part (i.e., from line 5 to line 33) iterates over the entities and constructs the `SELECT` statement and the `JOIN` clauses depending on the number and the type of the navigations (i.e., one-to-one, one-to-many, many-to-one, many-to-many). Finally, the third part (i.e., from line 34 to line 40) constructs the `WHERE` clause. The algorithm relies on the functions TABLENAME, TABLEALIAS, COLUMNNAME, and HASNEXT (see footnotes on Algorithm 2).

The first row of Table IV illustrates the execution of this algorithm for the URL shown on the left. This URL identifies the price of the product 1 belonging to the supplier 1. The corresponding input parameters are: $E = \{E_1 : Supplier, E_2 : Product\}$, $N = \{N_{1,2} : Products\}$, $x = \{x_1 : 1, x_2 : 3\}$, and $p = Price$. The resulting SQL query is shown on the right.

2) Query transformation: OData allows querying data via HTTP GET requests to the resources addressed by a resource path (as shown before). OData queries can include a set of options which are string parameters prefixed by a `$` symbol that control the amount and order of the returned data for a resource. Query options can be used to refine the result of an OData request and therefore are also considered in our transformation. Table III shows the query options provided by OData and the corresponding mapping rules to generate the SQL code. For each query option, the table provides a small description, the mapping rule with SQL, an example, and the corresponding SQL query.

OData also defines (1) logical and arithmetic operators (e.g., `eq` for equals or `add` for addition) to use with the `$filter` query option, (2) utility functions for string, date and time management (e.g., `concat`, `hour()` or `now()`), and (3) combining query options to create advanced queries. Our approach covers the operators and functions supported by MySQL database[16].

The second row of Table IV shows an example of query option mapping. On the left, the Table shows a URL example to retrieve 10 records after the position 5 ordered by name from the collection of products which have a name containing the character `i` and a price less or equals to `2.6`; while on the right, the Table lists the corresponding SQL query.

3) Data modification transformation: To perform data modification actions, OData relies on the HTTP `POST`, `PUT`, `PATCH` and `DELETE` requests. To support data modification actions, we generate a set of controllers which process the client requests and generate the corresponding SQL statements according to specification.

Table II shows the usage of HTTP methods in OData illustrated with examples. To create an entity, the client must send a `POST` request containing a valid representation of the new entity to the URL of a collection of entities (i.e., *EntitySet*, e.g., *Products*). This request is transformed to an `INSERT` statement (see example 5). To update an entity, the client must send a `PATCH` request containing a valid representation of the properties to update to the URL of a single entity (e.g., *Products(1)*) . This request is transformed to an `UPDATE` statement (see example 6). To update a navigation property, the client must send a `PUT` request containing the URL of the new related entity to the URL of the reference[17] of a single-valued navigation property (e.g., *Products(1)/Supplier/$ref*). This request is transformed to an `UPDATE` statement (see example 7). Finally to remove an entity, the client must send a `DELETE` request to the URL of an individual entity (e.g., *Products(1)*). This request is transformed to a `DELETE` statement. Next, we explain the serialization and deserialization mechanisms of the requests.

C. OData Serializer and Deserializer Generation

This process generates a serializer and a deserializer for OData objects supporting both the OData JSON [9] and Atom [10] formats.

The serializer applies a model-to-text transformation to the result of OData requests (i.e., entity collection, entity and property) in order to generate the textual representation according to OData format conventions. For instance, in the case of JSON, an entity collection is transformed to a JSON array holding the entities while an entity is represented by a JSON object containing a list of key/value pairs representing its properties. A similar process is followed for the Atom representation format. OData representation formats also support different levels (and content) of metadata (i.e., *full*, *minimal* or *none*) [9, 10], which can be configured in the header of any OData request. The generated serializer also takes into account this setting and generates the JSON or Atom representation accordingly. An example of a collection of *products* in OData JSON format was previously shown in Listing 2. As can be seen, apart from the properties of the entity, the JSON object also includes as metadata the annotation `odata.context` which indicates the root context URL of the payload.

The deserializer processes and parses the body of the OData requests `POST`, `PUT` and `PATCH` in order to generate the details of the `INSERT` and `UPDATE` SQL statements, accordingly. For instance, in the case of JSON (see the examples 5, 6 and 7 in Table II), each key and value in the JSON object are transformed to the corresponding field in the corresponding table and the value for such field, respectively, in the generated SQL statement. A similar process is followed for the Atom representation format.

VII. Tool Support

Our approach is available as a proof-of-concept plugin for the Eclipse platform [12]. The plugin extends the platform to provide contextual menus to obtain OData models from existing UML models and, given an OData model instance: (1) generate the metadata document conforming to OData specification; (2) generate the DDL of the database; and (3) generate an OData service based on a Maven-based project.

Our OData metamodel has been implemented using the Eclipse Modeling Framework (EMF). The UML to OData model transformation relies on UML2[18] which provides an EMF-based implementation of the UML 2.5 OMG metamodel; while the code generators are based on Acceleo[19], an implementation of the MOF Model-To-Text Transformation Language (MTL) specification from OMG, to define the logic and generate the different OData artifacts.

The generated Web application includes a properties file with the configuration of the database. The OData service implementation relies on Apache Olingo[20] to provide support for OData query language and serialization; and JOOQ[21] which provides a DSL to build SQL queries. The implementation includes controllers to analyze and deserialize the requests, transform them into SQL queries, execute the queries, and serialize the result to be sent back to the client.

Figure 5 shows a screenshot of the generated application for the running example. The figure includes the structure of

[16]More details can be found at https://github.com/SOM-Research/odata-generator.

[17]A reference is specified by adding `$ref` to the resource path of the navigation property

[18]https://wiki.eclipse.org/MDT/UML2

[19]https://www.eclipse.org/acceleo/

[20]http://olingo.apache.org

[21]https://www.jooq.org

TABLE III
ODATA SYSTEM QUERY OPTIONS AND THEIR CORRESPONDING SQL RULES.

OPTION	DESCRIPTION	SQL RULE	QUERY OPTION EXAMPLE	SQL EXAMPLE
`$filter`	Filter a collection of resources that are addressable by a request url	Add a `WHERE` clause including the corresponding operator to the filter expression	http://host/service/Products?$filter= Name eq 'Milk'	`SELECT * FROM product p WHERE p.name LIKE 'Milk'`
`$expand`	Include relative resource in line with retrieved resources	For each retrieved resource, create a `SELECT` statement to retrieve the relative resource	http://host/service/Suppliers(1) ?$expand=Products	`SELECT * FROM product p WHERE supplier_id = 1`
`$select`	Request a specific set of properties	Add the corresponding column names to the `SELECT` statement	http://host/service/Products?$select= Name,Price	`SELECT p.name, p.price FROM product p`
`$orderby`	Request resources in either ascending order using `asc` or descending order using `desc`	Add the `ORDER BY` clause to the `SELECT` statement	http://host/service/Products? $orderby='Name' desc	`SELECT * FROM product p ORDER BY p.name DESC`
`$top $skip`	`$top` requests the number of items to be included and `$skip` requests the number of items to be skipped	Add the supported clause by the database system (e.g., `LIMIT` for MySQL)	http://host/service/Products?$top= 10$skip=5	`SELECT * FROM product p LIMIT=5,10`
`$count`	Request a count of the resources	Add the `COUNT` function to the `SELECT` statement	http://host/service/Product?$count	`SELECT COUNT(*) FROM product`
`$search`	Request entities matching a free text search	Add a set of `LIKE` operators to the `WHERE` clause to much the search text with all the text columns of the corresponding table	http://host/service/Product?$search= 'Milk'	`SELECT * FROM product p WHERE p.name LIKE 'Milk' OR p.description LIKE 'Milk'`

TABLE IV
EXAMPLE OF ODATA REQUEST TO SQL MAPPING.

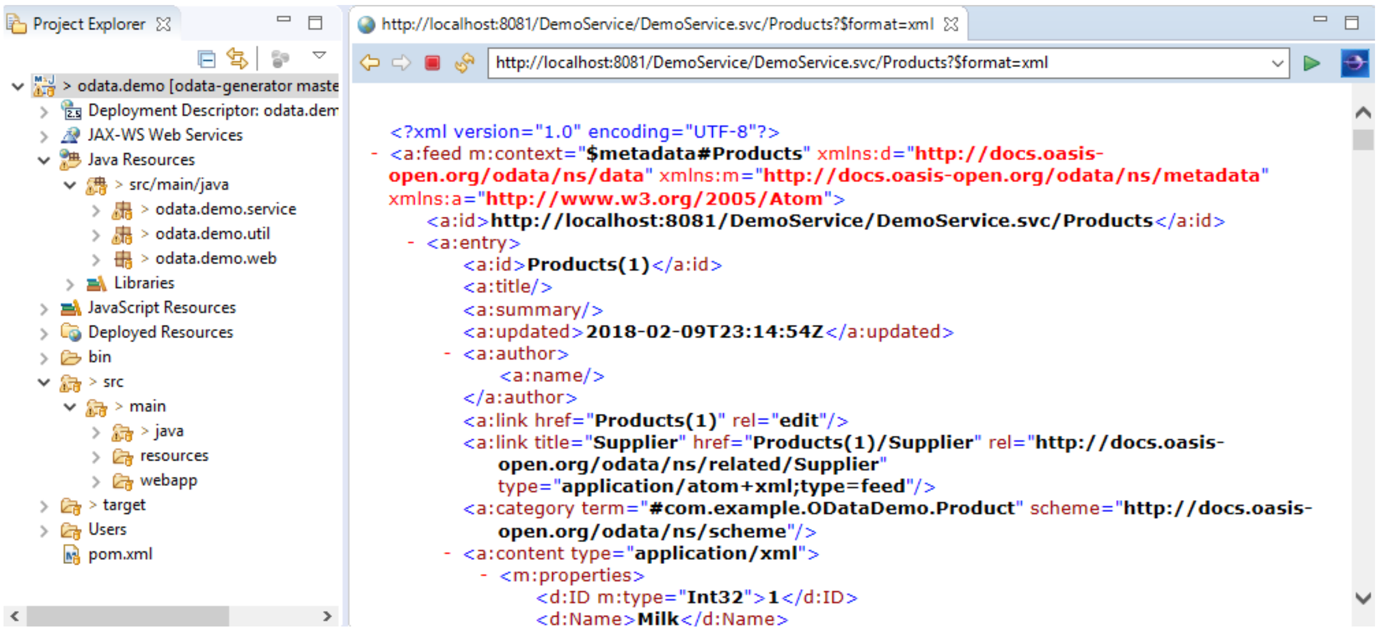
ODATA QUERY	SQL QUERY
`http://host/service/Suppliers(1)/Products(1)/Price`	`SELECT p.price`
	`FROM product p`
	`JOIN supplier s ON p.supplier_id = s.id`
	`WHERE p.id = 1 AND s.id = 1`
`http://host/service/Products?`	`SELECT p.name, p.price`
`$select=Name,Price`	`FROM product p`
`&$filter=contains(Name,'i') and Price le 2.6`	`WHERE p.name LIKE '%i%' AND p.price <= 2.6`
`&$orderby=Name desc`	`ORDER BY p.name DESC`
`$skip=5&$top=10`	`LIMIT 5,10`

the Maven project (see left panel) and a browser showing the result of request in Atom format (see right panel). The service currently implements the support for: (1) resource path URL transformation; (2) data querying using the query options `$filter`, `$top`, `$skip` and `$orderby`; and (3) data modification as described in the previous section. The generated application returns `501, ``Not Implemented''` for any unsupported functionality as required by the protocol. The complete generated application can be found in our repository [12]. The repository includes also the steps to install the plugin, generate the OData service, and deploy the generated application in a Servlet container.

VIII. RELATED WORK

Model-driven approaches have been widely used in the Web Engineering field to generate different kinds of web applications (e.g., [5–7, 14–19]). While existing approaches already provide methodologies and tools to cover a variety of technologies (e.g., web services, ubiquitous applications), specific support for Web APIs is rather limited. For instance, Porres et al. [20] and Tavares et al. [21] propose to model REST APIs using UML and a REST metamodel, respectively, but only generate a WADL document [22] describing the behavior of a REST API and do not generate the API implementation. A few exceptions to generate REST APIs are: (1) EMF-REST [14] (but for generating web modeling environments); (2) ODaaS [15] (for the exploitation of existing Open Data and social media streams); (3) the work by Haupt et al. [16] (targeting REST-compliant services in a broad sense); (4) ELECTRA [17] and MockAPI [23] (for fast prototyping of mockup APIs); (5) the work by Rodríguez-Echeverría et al. [18] (deriving REST APIs from legacy Web applications); and (6) MicroBuilder [24] and the work by Haupt et al. [16] (using ad-hoc DSLs for the specification and the realization of REST APIs). None of them, though, have explicit support neither for modeling OData nor for automatically generating OData Services from high-level models.

Some SDKs provide support for developing OData applica-


Project Explorer

```
> odata.demo [odata-generator maste
  Deployment Descriptor: odata.dem
  JAX-WS Web Services
  Java Resources
    > src/main/java
      > odata.demo.service
      > odata.demo.util
      > odata.demo.web
    Libraries
  JavaScript Resources
  Deployed Resources
  bin
  > src
    > main
      > java
      resources
      webapp
  > target
  Users
  pom.xml
```

http://localhost:8081/DemoService/DemoService.svc/Products?$format=xml

```
<?xml version="1.0" encoding="UTF-8"?>
- <a:feed m:context="$metadata#Products" xmlns:d="http://docs.oasis-
  open.org/odata/ns/data" xmlns:m="http://docs.oasis-open.org/odata/ns/metadata"
  xmlns:a="http://www.w3.org/2005/Atom">
    <a:id>http://localhost:8081/DemoService/DemoService.svc/Products</a:id>
  - <a:entry>
      <a:id>Products(1)</a:id>
      <a:title/>
      <a:summary/>
      <a:updated>2018-02-09T23:14:54Z</a:updated>
    - <a:author>
        <a:name/>
      </a:author>
      <a:link href="Products(1)" rel="edit"/>
      <a:link title="Supplier" href="Products(1)/Supplier" rel="http://docs.oasis-
        open.org/odata/ns/related/Supplier"
        type="application/atom+xml;type=feed"/>
      <a:category term="#com.example.ODataDemo.Product" scheme="http://docs.oasis-
        open.org/odata/ns/scheme"/>
    - <a:content type="application/xml">
      - <m:properties>
          <d:ID m:type="Int32">1</d:ID>
          <d:Name>Milk</d:Name>
```

Fig. 5. A screenshot of the generated application for the running example.

tions for a target platform (e.g., *RESTier*[22], *Apache Olingo*[23], *SDL OData Frameworks*[24]). These frameworks are handy for developers but require knowledge to deal with the intricacies of their architecture[25] to create OData applications.

Support for generating OData applications is so far limited to commercial tools like Cloud Drivers[26], OData server[27] or Skyvia Connect[28]. Still, these solutions only offer ways to expose OData services from already existing data sources such as databases but not to create new OData services from scratch nor to configure the full support to OData specification. In fact, OData services generated from relational databases just mirror the data structure (i.e., tables and relationships to entities and navigation entities, respectively), thus not leveraging on OData protocol which supports richer data structures (e.g., hierarchies, complex type or multivalued properties) and capabilities. For instance, we tested the trial version of *OData server* to create an OData server for the MySQL database of our running example. Besides the limitation regarding the use of richer data structures (e.g., `Address` was transformed to an `Entity` and not a `ComplexType`), we also detected other issues related to Open API capabilities: (1) there is no entity container and therefore clients are not able to query the data; (2) the foreign keys are plain properties instead of navigation properties; (3) data is read-only.

[22]https://github.com/OData/RESTier
[23]https://olingo.apache.org/
[24]https://github.com/sdl/odata
[25]There are 129 open issues in the Github repository of *RESTier* and StackOverFlow lists 396 questions regarding *Olingo*.
[26]http://www.cdata.com/odata/
[27]https://rwad-tech.com/
[28]https://skyvia.com/connect/

Other tools such as *simple-odata-server*[29] and *JayDATA*[30] allow generating a basic OData server but require providing both an OData Entity model of the desired application and the corresponding database. Also, they only support a subset of the query options offered by OData protocol. On the other hand, our approach has advanced support for OData protocol and provides the database implementation of the data model out of the box.

IX. Conclusion

In this paper we have presented a model-driven approach to specify and generate OData services. UML models are used to generate the required artifacts to deploy OData services relying on a relational database as storage solution. The generation process covers the specification of the OData metadata document, the database schema, the resolution of URL requests into SQL statements and a de/serialization mechanism for the exchanged messages. Our approach advances towards the definition of an MDE infrastructure for the generation of OData services, where developers can rely on the plethora of modeling tools to easily design, generate and evolve their web applications.

As future work we aim at extending our OData metamodel to capture additional OData behavioral concepts such as functions and actions to enable the design and generation of more complex aspects. We plan to extend our approach to comply with the advanced OData conformance level, which implies adding support to other OData functionalities such as canonical functions. We aim at extending the generative approach to add predefined support for a number of basic features in

[29]https://github.com/pofider/node-simple-odata-server
[30]https://github.com/jaystack/jaydata

any web infrastructure like security (i.e., authentication and encryption). Finally, we would like to integrate our OData models with other web-based modeling languages like IFML that focus on the modeling of the user interaction with the web application. With this integration we aim to provide a rich modeling environment combining both front-end and back-end development.

Acknowledgment

This work has been supported by the Spanish government (TIN2016-75944-R project).

References

[1] M. Pizzo, R. Handl, and M. Zurmuehl, "OData version 4.0 part 1: protocol," OASIS, Tech. Rep., 2014.

[2] H. Ed-Douibi, J. L. Cánovas Izquierdo, and J. Cabot, "A UML profile for OData APIs," in *Int. Conf. on Web Engineering*, 2017.

[3] M. Pizzo, R. Handl, and M. Zurmuehl, "OData version 4.0 part 2: URL Conventions," OASIS, Tech. Rep., 2014.

[4] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, 2003.

[5] W. Schwinger, W. Retschitzegger, A. Schauerhuber, G. Kappel, M. Wimmer, B. Pröll, C. Cachero Castro, S. Casteleyn, O. De Troyer, P. Fraternali *et al.*, "A Survey on Web Modeling Approaches for Ubiquitous Web Applications," *Int. J. of Web Information Systems*, vol. 4, no. 3, pp. 234–305, 2008.

[6] P. Valderas and V. Pelechano, "A Survey of Requirements Specification in Model-driven Development of Web Applications," *ACM Transactions on the Web*, vol. 5, no. 2, p. 10, 2011.

[7] P. Fraternali, "Tools and Approaches for Developing Data-intensive Web Applications: a Survey," *ACM Computing Surveys*, vol. 31, no. 3, pp. 227–263, 1999.

[8] M. Pizzo, R. Handl, and M. Zurmuehl, "OData version 4.0 part 3: Common Schema Definition Language (CSDL)," OASIS, Tech. Rep., 2014.

[9] R. Handl, M. Pizzo, and M. Biamonte, "OData JSON Format version 4.0," OASIS, Tech. Rep., 2014.

[10] M. Zurmuehl, M. Pizzo, and R. Handl, "OData Atom Format Version 4.0," OASIS, Tech. Rep., 2014.

[11] B. Selic, "A Systematic Approach to Domain-specific Language Design using UML," in *Int. Symp. on Object and Component-Oriented Real-Time Distributed Computing*, 2007, pp. 2–9.

[12] "OData generator https://github.com/SOM-Research/odata-generator."

[13] R. D. N. Fidalgo, E. M. De Souza, S. España, J. B. De Castro, and O. Pastor, "EERMM: a metamodel for the enhanced entity-relationship model," in *Int. Conf. on Conceptual Modeling*, 2012, pp. 515–524.

[14] H. Ed-Douibi, J. L. Cánovas Izquierdo, A. Gómez, M. Tisi, and J. Cabot, "EMF-REST: Generation of RESTful APIs from Models," in *ACM/SIGAPP Symp. On Applied Computing*, 2016, pp. 1446–1453.

[15] A. M. Segura, J. S. Cuadrado, and J. de Lara, "ODaaS: Towards the Model-driven Engineering of Open Data applications as Data Services," in *Int. Conf. on Enterprise Distributed Object Computing, Workshops and Demonstrations*, 2014, pp. 335–339.

[16] F. Haupt, D. Karastoyanova, F. Leymann, and B. Schroth, "A Model-Driven Approach for REST Compliant Services," in *Int. Conf. on Web Services*, 2014, pp. 129–136.

[17] J. M. Rivero, S. Heil, J. Grigera, E. Robles Luna, and M. Gaedke, "An Extensible, Model-driven and End-User Centric Approach for API Building," in *Int. Conf. on Web Engineering*, S. Casteleyn, G. Rossi, and M. Winckler, Eds., 2014, pp. 494–497.

[18] R. Rodríguez-Echeverría, F. Macías, V. M. Pavón, J. M. Conejero, and F. Sánchez-Figueroa, "Model-driven Generation of a REST API from a Legacy Web Application," in *Int. Conf. on Web Engineering, Workshops*, 2013, pp. 133–147.

[19] A. Vallecillo, N. Koch, C. Cachero, S. Comai, P. Fraternali, I. Garrigós, J. Gómez, G. Kappel, A. Knapp, M. Matera, S. Meliá, N. Moreno, B. Pröll, T. Reiter, W. Retschitzegger, J. E. Rivera, A. Schauerhuber, W. Schwinger, M. Wimmer, and G. Zhang, "MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods," in *Int. Conf. on Web Engineering, Workshops*, 2007.

[20] I. Porres and I. Rauf, "Modeling Behavioral RESTful Web Service Interfaces in UML," in *Symp. on Applied Computing*, 2011, pp. 1598–1605.

[21] N. A. Tavares and S. Vale, "A Model Driven Approach for the Development of Semantic RESTful Web Services," in *Int. Conf. on Information Integration and Web-based Applications & Services*, 2013, p. 290.

[22] M. J. Hadley, "Web Application Description Language (WADL)," Tech. Rep., 2006.

[23] J. M. Rivero, S. Heil, J. Grigera, M. Gaedke, and G. Rossi, "MockAPI: an Agile Approach Supporting API-first Web Application Development," in *Int. Conf. on Web Engineering*, 2013, pp. 7–21.

[24] B. Terzić, V. Dimitrieski, S. Kordić, G. Milosavljević, and I. Luković, "MicroBuilder: A Model-driven Tool for the Specification of REST Microservice Architectures," in *Int. Conf. on Information Society and Technology*, 2017, pp. 179–184.