Christopher Werner, Manuel Wimmer, Uwe Aßmann

A Generic Language for Query and Viewtype Generation By-Example

Erstveröffentlichung in / First published in:

Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). München, 15. – 20.09.2019. IEEE Xplore, S. 379 – 386. ISBN 978-1-7281-5125-0.

DOI: https://doi.org/10.1109/MODELS-C.2019.00059

Final edited form was published in "Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). München 2019", S. 379 - 386. ISBN: 978-1-7281-5125-0
https://doi.org/10.1109/MODELS-C.2019.00059

A Generic Language for Query and Viewtype Generation By-Example

1st Christopher Werner
Software Technology Group
Technische Universität Dresden
Dresden, Germany
christopher.werner@tu-dresden.de

2nd Manuel Wimmer
Institute of Business Informatics - Software Engineering
CDL-MINT, Johannes Kepler University Linz
Linz, Austria
manuel.wimmer@jku.at

3rd Uwe Aßmann
Software Technology Group
Technische Universität Dresden
Dresden, Germany
uwe.assmann@tu-dresden.de

***Abstract*—In model-driven engineering, powerful query/view languages exist to compute result sets/views from underlying models. However, to use these languages effectively, one must understand the query/view language concepts as well as the underlying models and metamodels structures. Consequently, it is a challenge for domain experts to create queries/views due to the lack of knowledge about the computer-internal abstract representation of models and metamodels.**

To better support domain experts in the query/view creation, the goal of this paper is the presentation of a generic concept to specify queries/views on models without requiring deep knowledge on the realization of modeling languages. The proposed concept is agnostic to specific modeling languages and allows the query/view generation by-example with a simple mechanism for filtering model elements. Based on this generic concept, a generic query/view language is proposed that uses role-oriented modeling for its non-intrusive application for specific modeling languages. The proposed language is demonstrated based on the role-based single underlying model (RSUM) approach for AutomationML to create queries/views by-example, and subsequently, associated viewtypes to modify the result set or view.

***Index Terms*—query by-example, view by-example, role-based, viewtype generation**

I. Introduction

In all areas, more and more software-intensive systems are developed, which are adapted over time to master new situations, and thus, constantly evolve. In View-Oriented Software Engineering (VOSE), the complexity of the underlying model is reduced externally by dividing the entire model into views. This allows developers to work only on those parts of the model that are relevant to them. To use such an approach, viewtypes must be generated on the underlying model. This step can either be done at design time by predefining a set of viewtypes, or runtime by defining and generating new viewtypes on demand.

There are already approaches that provide languages to define a query or viewtype on an underlying model, e.g., AutomationQL [1] and ModelJoin [2]. However, these languages are usually domain-specific or require deep knowledge of the language and the underlying model [1], [3] making usage difficult for domain experts not being computer scientists. In addition, current languages are usually a combination of different existing languages [2], [4], which means an increased training effort for the user. Since learning a language is a time-consuming process, it should be possible to create queries without advanced knowledge of the underlying model and modeling language.

For this reason, the aim of this paper is to present a generic concept for specifying queries and viewtypes. The presented concept must not contain any domain dependencies, so that it can be used universally on every domain model without specific requirements to the underlying model. This point describes the independence from the domain model. However, it is still necessary to know the domain models in order to display the links correctly in the queries and viewtypes. To reduce this effort, queries and viewtypes are created based on example elements of the underlying model. This ensures that each user only creates queries and viewtypes on elements to which the user has access and can filter them based on the appearance of their attributes and reference values.

In this paper, we present a role-oriented approach that allows to provide a by-example concept for generating queries and viewtypes. For this, we define a novel query context, which can be used by any element of a model. This context is first introduced as a general variant, to subsequently apply it for several modeling approaches. We show the usability of the query context by adapting it to the Role-based Single Underlying Model (RSUM) approach using AutomationML (AML) as an example.[1] In addition, the power of the query concept and the creation of viewtypes are presented.

The remainder of this paper is structured as follows. The next section summarizes background knowledge about closely related topics like query languages, VOSE approaches, and the role concept. Sect. III provides an in-depth discussion of the overall concepts and describes the underlying process. Sect. IV maps the presented concept to the RSUM approach exemplary with AML. Sect. V evaluates the concept and RSUM realization on the AML example. We demarcate our approach from related work in Sect. VI. Finally, in Sect. VII, we conclude the paper and discuss lines of future work.

This work has been funded by the German Research Foundation within the Research Training Group ”Role-based Software Infrastructures for continuous-context-sensitive Systems” (GRK 1907), by the Austrian Federal Ministry for Digital and Economic Affairs, and the National Foundation for Research, Technology and Development.

[1]https://git-st.inf.tu-dresden.de/cwerner/rsum

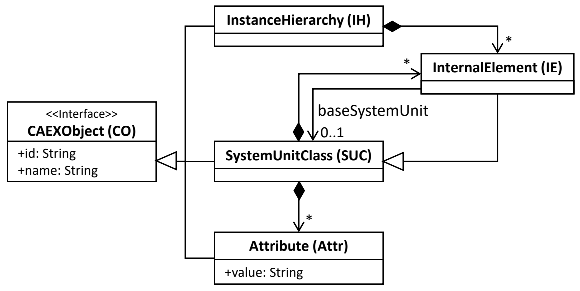

Figure 1. AutomationML metamodel excerpt.

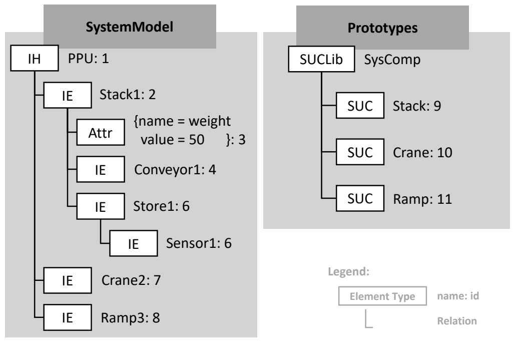

Figure 2. Example instance model of AML.

II. Background

A. Motivating and Running Example

For this paper, we select AutomationML (AML) [5] as a motivating and running example. AML[2] is a standardized data format for representing engineering knowledge in the area of process automation and control of production systems. A metamodel excerpt is presented in Fig. 1 and an example instance model is shown in Fig. 2. The basis for AML is CAEX for representing plant topology information. The entire plant topology model is represented as an `InstanceHierarchy` in AML. Components of the plant are represented as (potentially nested) `InternalElements`. The type of a component is represented as `SystemUnitClass`. For expressing further details about internal elements or system unit classes, attributes can be defined to state internal properties of the components. To utilize the benefits offered by modern model-driven frameworks for AML, we have developed a model-driven engineering workbench for AML in previous work [6].

Fig. 2 shows a small AML model. On the left hand side, the figure depicts a typical pick and place unit (PPU) which consists of a stack, crane, and ramp. The stack is modeled in more detail by containing further internal elements. The main components are typed by the system unit classes provided by the library shown on the right hand side of Fig. 2.

Of course, query languages such as the Object Constraint Language (OCL) may be directly employed for querying AML models. Although such query languages offer powerful query concepts and mechanisms, for domain experts it is challenging to use such general query languages to formulate queries. For instance, finding all leaf elements in an instance hierarchy may require deep knowledge how internal elements are queried and filtered with respect to a negative condition of not containing further elements. In [1], we presented a by-example query language for AML which provides a more user-friendly interface to define queries for domain experts. However, we used a generative approach to extract the query language from the AML metamodel, which results in an additional language to define the queries, additional tool dependencies, and additional languages to represents the result sets of the queries. Therefore, in this paper we aim for a different solution which allows to work more natively and dynamically with the given models and tools.

[2]https://www.automationml.org

B. View-Oriented Software Engineering

View-oriented software engineering (VOSE) describes the management of views that represent parts of the underlying model. These parts are used to restrict access of specific areas in the model, to ensure clarity, to only display relevant information, and to separate responsibilities of user groups. The first definition of views and viewtypes comes from the IEEE 1471/ISO 42010 standard [7] where views are considered as instances of viewtypes. Goldschmidt *et al.* [8] define views as “the actual set of objects and their relations displayed using a certain representation and layout” [8, p 63] and viewtypes as “rules according to which views of the respective type are created” [8, p 64]. Current view-based approaches can be divided into two categories: (1) synthetic approaches where an architect defines fixed views of an underlying model and (2) projective approaches where the views are automatically generated by means of a domain-specific language (DSL). The OSM [9] approach is a synthetic approach, whereby a single underlying model (SUM) is created in a top-down process, from which fixed views are defined. Projective approaches are the RSUM [10] approach, which is specified in more detail in the next section, and the VITRUVIUS [11] approach. VITRUVIUS generates a virtual SUM using a bottom-up approach by keeping several models consistent using different constraints. Therein, ModelJoin [2] is used to create editable views. The MoConseMI [12] approach is a hybrid of both categories, whereby special operators are used to transfer models into each other, and thus, unify them.

C. Role Concept

The idea of role-oriented software development goes back to the 1970s. In the 2000s Steinmann [13] and Kühn *et al.* [14] identified 27 features of roles from analyzing related work. The features can be divided into three categories that reflect the nature of roles. (1) The **behavioral** nature expresses that unrelated objects can play roles and that the roles change the behavior of their playing object. (2) The **relational** nature describes roles as ends of relations, as they are already used in

Final edited form was published in "Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). München 2019", S. 379 - 386. ISBN: 978-1-7281-5125-0
https://doi.org/10.1109/MODELS-C.2019.00059

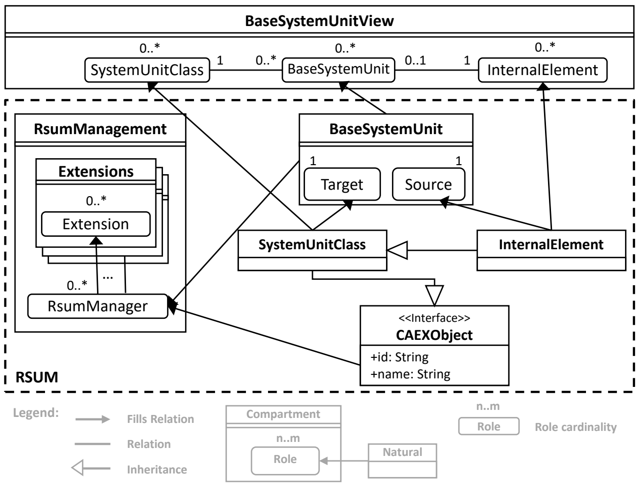

Figure 3. RSUM approach with AML example.

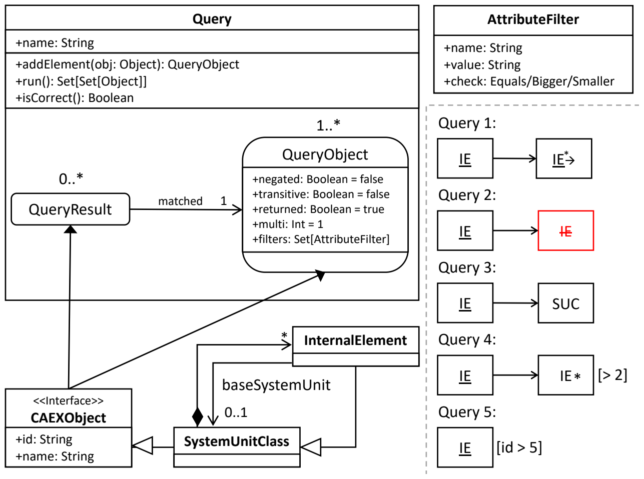

Figure 4. Role concept for the query language with a visual query representation of the properties.

conceptual modeling languages such as in UML. However, relations are important when it comes to (3) the **context-dependent** nature of roles, since relations and roles change behavior with respect to a particular context, i.e., roles describe different behaviors depending on the context, and thus objects adapt to the active context.

This paper focuses on the Compartment Role Object Model (CROM) [15], which represents the role features and a graphical notation that permits correct visual representation of role models. Fig. 3 shows a graphical representation of a role model. The models consist of three different types that interact with each other and constitute the role concept. Natural types form the basic types in the role concept and have alone no interaction possibilities with each other. It is possible for the natural types to fulfill role types in compartment types. Compartment types in turn contain role types that interact with each other and act as a kind of context. In the role concept, each interaction takes place in compartment types, where *fills* relations are used to create links across compartment boundaries. The role-based programming language used for this approach is the SCala ROLes Language (SCROLL) [16], which is an embedded DSL for Scala and provides many of the role features. It allows binding and unbinding roles at runtime and with Scala, it is possible to load new roles and compartments at runtime and extend by this the runtime model.

The RSUM [10] approach already presents a role-based view approach that enables and implements the creation and consistency management of views from a SUM. The RSUM approach provides a simple and fine-grained mechanism to maintain consistency between views and an underlying model. In this approach, views are represented as compartments, with roles acting as connectors between the underlying model and the views. Moreover, the fills relations act as traceability links. The role concept allows runtime adaptation with roles so that new mechanisms can be integrated into the core with small effort. Fig. 3 shows the concept of the RSUM approach using a minimal piece of the AML metamodel from the previous section. The core includes the base classes as natural types and the relations as relational compartments. There is a mapping for all relations to a relational compartment that allows the runtime adaptation and extension of the RSUM. The RSUM core can be seen as a graph with nodes and edges, because of the usage of relational compartments and naturals. Therefore, it is necessary to detect graph patterns in this construction to query elements with specific properties. In addition, the `RsumManagement` compartment is the global coordinator in the RSUM to manage all naturals, relational compartments, viewtypes, and views as instances of the viewtypes.

III. Concept

This section presents the general concept for a role-based query/view language that does not need the adaptation of an underlying metamodel. It describes on the one hand the use of this query language and on the other hand the variety of queries that can be expressed. With the help of this concept, it is possible to use existing or create new elements for the specification and formulation of the queries. The disadvantage of using an already existing element in the query is the addition of new structural concepts to this element without changing the element in the underlying model. On the contrary, the advantage of using existing elements is the simple creation of queries. However, if these elements are deleted from the underlying model, the connected queries are changed because the queries do not save copies of the elements. If queries are only created with new elements that are not integrated into the underlying model, the query objects do not have dependencies to the underlying model and provide reusability and extendibility. In the following subsection, the creation of suitable viewtypes for the queries is presented that contain a minimal number of view elements and allow the direct displaying and editing of the query results.

A. *Role-based query/view language*

Fig. 4 shows the query concept at meta level with a small excerpt of the AML metamodel and a set of five sample queries. The concept is based on a general `Query` compartment, which contains the roles `QueryObject` and `QueryResult`. The two roles have *fills* relations to all classes in the underlying model, which is represented by the existing *fills* relations between `CAEXObject` and the roles. All elements can be part of one or more queries and one or more result sets of different queries. There is also a *matched* relationship between `QueryObject` and `QueryResult` that expresses why elements are part of the solution set. The *matched* relationship is built during the execution of the query by storing in each `QueryResult` by which `QueryObject` it was created. After executing the query, the query compartment contains all information about the query itself and its results through the played roles.

Fig. 4 also presents a graphical notation of five queries that can be formulated with our approach. The graphical language is only used to visualize the queries and is not linked to the presented approach. The relations between the query objects are not shown directly in the general concept of Fig. 4. However, they can be recognized by the *fills* relations, because each role has access to its attributes and references via its player. This allows the recognition of the relations between the base objects. Since this method is very complex it is extended in the RSUM implementation with roles for relations.

In the query language, it is possible to combine all query blocks from Fig. 4. There are certain rules which must be considered: (a) an object may only play a `QueryObject` role once in a query compartment. (b) A relation must always be defined with exactly one direction. (c) With a transitive query object, any attached object can be placed anywhere in the chain. (d) The negating and prescribing of an equals connection between two objects must not take place. This list describes the most important points and limitation to consider when creating a query.

The process of creating a query is a five-step process: (1) creating an instance of the query compartment type, (2) assigning elements to the query (binding `QueryObject` roles using the *addElement* method), (3) adjusting the properties for each `QueryObject` in the query, (4) adding `AttributeFilters` to `QueryObjects`, and (5) executing the query on a set of objects using the *run* method. The result of a query now is comprised by a set of objects that fulfill the query and play `QueryResult` roles in it. To determine the results, the properties of the `QueryObject` play an important role, which are explained next in more detail.

- **Transitive:** This property creates the transitive hull of a specific relationship. Query 1 returns a chain of all `InternalElements` and their associated `InternalElements`.
- **Negated:** The negated field searches for the non-existence of relations. In Query 2, all leaf nodes in the chain or tree structure of the `InternalElements` are searched.
- **Returned:** The returned field specifies whether elements that map to this `QueryObject` should be represented in the solution set. In Query 3, this capability is used to return only `InternalElements` that have a *baseSystemUnit*. The corresponding `SystemUnitClass` does not appear in the solution set (not underlined).
- **Multi:** The multi value specifies the minimum number of connected elements of a relationship. In Query 4, `InternalElements` are searched which are linked to at least three `InternalElements`.
- **Filters:** With this option, the solution set can be filtered by different attribute values. Query 5 filters out all `InternalElements` whose *id* value is less than six.

It is now possible to directly generate a result set based on the existing element set with the *run* method specified in the query compartment. However, this step only creates a general set of sets of untyped objects, because it is not possible to return the typed elements due to the generality of the query concept. For this reason, an interface must be provided to visualize and modify the result set. The next section describes how to use queries for generating viewtypes to make the result set easier to edit and display.

B. *Definition of views using queries*

After creating a query, the query is used to define and generate a corresponding viewtype. First, a textual representation of the query is generated. This step creates a textual abstraction of the query removing the dependencies to the example elements. This textual representation is usable in other query frameworks as well. Either the query language used in our example can be directly employed or a transformation into another language can be carried out. Second, a viewtype is generated from the textual language, which only represents the types from the result set. For Query 1, i.e., that only `InternalElements` are visible in the viewtype and all other elements cannot be visualized. After creating the viewtype, a view is created as an instance of the viewtype which gets the result set as input and provides a link between the view, the result set, and the query. Afterwards, the elements can be modified directly via the provided interfaces in the view. In Sect. IV, AML is used as an example to illustrate how the query and view concept are implemented using the RSUM approach. The generated viewtype no longer represents a unique state of the underlying model, i.e., it must react to changes of the underlying model and may change it itself. These requirements raises two new questions: (1) Which changes can be made in the view? (2) To which changes in the underlying model must a view react? These questions depend on the properties presented in the previous subsection.

To answer question (1), there is an optimistic and a pessimistic solution. The pessimistic solution is to prohibit changes that result in removing query results from the view. The optimistic solution, on the other hand, is to allow all modifications, i.e., changes can cause query results to disappear from the view. Examples of such changes are changing attributes that are part of a filter that is no longer fulfilled or deleting relations that

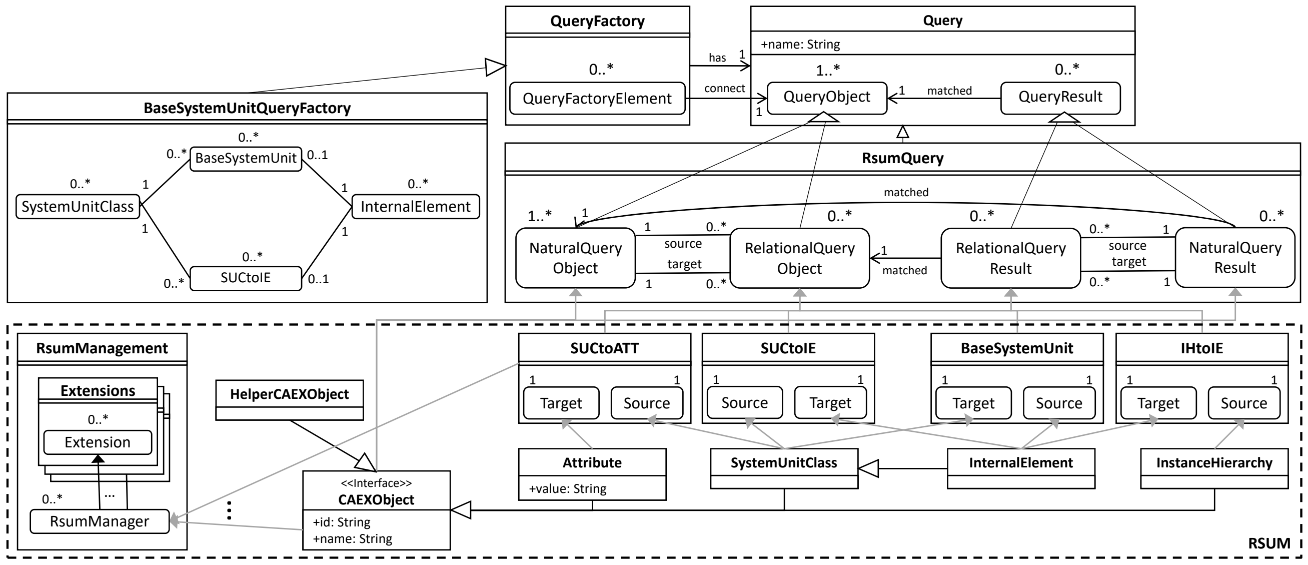

Figure 5. Map the AML example to the RSUM approach.

are necessary to include elements in such a view. As a result, the pessimistic approach does not allow deletion or creation of elements or changing of filter attribute values. On the contrary, the optimistic approach allows all changes to the objects in the view that can remove such objects afterwards.

Question (2) can also be answered in two ways. First, a complete recalculation of the query result could be performed after each change in the underlying model, or the view could only react on important changes in the underlying model. The recalculation of the complete view is easy to perform but time and resource consuming. For this reason, we only look at the second approach in more detail. We distinguish between what general changes need to be considered and how the properties of the query objects affect change propagation. In general, only the instances of the class types that occur in the query need to be considered. This minimizes the effort enormously. An occurrence of the properties transitive, negated, or multi can lead to a recalculation of the query, since previously excluded elements can slip back into the result set. In most cases, changes of elements only result in changes in these elements and directly connected elements slipping back into the query result. This fact reduces the computational effort. In general, i.e., that changes of elements usually only affect neighboring elements that must be investigated for integration into the query result set. If the internal dependencies of the model and the complexity of the query are too high, the complete recalculation of the query results could be more efficient than incremental modification. In the next section, this process is presented using the current example of AML and the RSUM approach presented in Sect. IV.

IV. Concept Implementation with RSUM

This section describes how to adapt and apply the previously described query concept to the RSUM approach. Fig. 5 shows the adaptation of the `Query` compartment to the RSUM approach and visualizes it using AML as an example. In order to adapt the AML metamodel of Fig. 1 to the RSUM approach, all classes are converted to natural types as shown in Fig. 5 and all relations are represented as relational compartments. Since the RSUM approach distinguishes between these two types of elements, we also separate these types in the modified query compartment (`RsumQuery`). This separation represents a graph structure with nodes (natural types) and edges (relational compartments). The `RsumQuery` compartment inherits from the `Query` compartment and implements all defined methods adapted to the RSUM approach. The compartment is able to verify the correctness of a query, as only queries with one connected structure are allowed. Thus, it is not possible to formulate queries that simply consist of two `QueryObjects` that are not connected to each other. If the user still wants to formulate such queries the user has to define two queries and combine the result sets. Like the general compartment, the roles `NaturalQueryResult` and `RelationalQueryResult` are specifications of the `QueryResult` role and `NaturalQueryObject` and `RelationalQueryObject` are specifications of the `QueryObject` role. A distinction between the specific roles must be made because roles with the prefix *Natural* can only be played by natural types and roles with the prefix *Relational* can only be played by relational compartments. However, this specification makes it possible for queries to be formed on all possible natural types and relational compartments without making any adjustments.

In addition, there are two other elements shown in Fig. 5 that are necessary for creating queries of any kind. First, there is a new natural type (`HelperCAEXObject`) that inherits from the `CAEXObject` interface which allows on the way the instantiation of a `CAEXObject` for queries. We implemented

Final edited form was published in "Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). München 2019", S. 379 - 386. ISBN: 978-1-7281-5125-0
https://doi.org/10.1109/MODELS-C.2019.00059

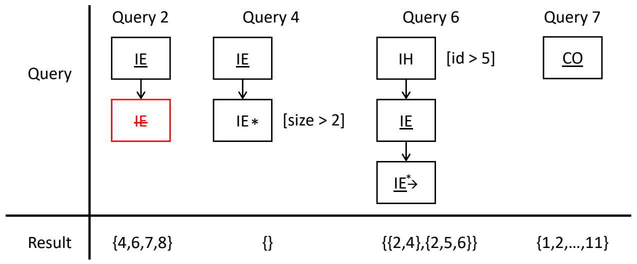

Figure 6. Queries and results on the AML instance model example.

```
1 AmlQueryFactory q2 = new AmlQueryFactory
2 InternalElement ie1 = q2.createInternalElement()
3 InternalElement ie2 = q2.createInternalElement()
4 ie1.addInternalElements(ie2)
5 //Set Properties
6 ie2.getQueryObject.negated = true
7 //Run Query
8 Set[Set[Objects]] result = q2.getQuery().run()
```

Listing 1. Creation of Query 2 in RSUM with factory.

a generator that generates helper classes for each interface and abstract class. These types allow the definition of more general queries, e.g., it is now possible to query all inherited types using a natural of this type.

The other new elements are the `QueryFactories`, which own one query. These factories create queries from new objects, which are not integrated in the underlying model. This mechanism allows to create queries with completely new objects with an easily understandable interface. The `QueryFactory` compartments also hide the helper classes described before to the user by returning the interfaces or abstract class types. In Fig. 5, a minimal example of such a factory is modeled. The `BaseSystemUnitQueryFactory` can create new `InternalElements` and `SystemUnitClasses` and relations between them for a query. It can be used to create all queries in Fig. 4. To improve the clarity of Fig. 5, all *fills* relations between the RSUM and the query factory are not shown. Such *fills* relations must exists between elements of the same name, i.e., the natural type `SystemUnitClass` fills the role `SystemUnitClass` etc. The query factories are special kinds of viewtypes because they do not influence the instances or react on changes in the RSUM.

After a query has been defined in this way, a textual representation is created in the ModelJoin [2] language, since the viewtype generation process is already implemented with ModelJoin in the RSUM approach. Support for other languages will be offered in the future. This textual representation can be used in approaches like VITRUVIUS [11] to create viewtypes. In our case, we use the language in a generator that creates viewtypes for the RSUM approach. In the viewtypes, we currently implement an optimistic mechanism to allow all possible changes. However, it is possible to manually modify these manipulation rights for each viewtype. The generated viewtypes behave like all other viewtypes in the RSUM approach, in that any number of views can be generated. The only difference is that it has a reference to the query to be able to recalculate the result set if necessary.

V. Evaluation

This section presents the complete process from creating a query from existing or newly created elements to the generated result set and viewtype. This process is visualized on four queries of the AML example. These queries are shown in Fig. 6 with the solution sets below using the example model from Sect. II. The queries 2 and 4 are taken from Fig. 4 and refer to the properties *multi* and *negated*. Query 6 is new and combines all properties of the previous queries 1, 3, and 5 from Fig. 4. For this query, all `InternalElements` are searched that belong to an `InstanceHierarchy` with an *id* value smaller than 5. In addition, the `InstanceHierarchy` elements are not mapped in the result set because they are not returned (not underlined in Fig. 6). In contrary, Query 7 collects all `CAEXObjects` from the underlying model.

After describing the queries, they must be created with the proposed by-example concept. As described in the previous section, there are two different instantiation mechanisms for the creation process. The `QueryFactory` compartments can be used for this or objects from the underlying model are transferred directly into an instance of the `RsumQuery` compartments.

Listing 1 shows the process using a `QueryFactory` compartment by creating Query 2 in pseudocode. In line 1 a new instance of the `AMLQueryFactory` is created, which automatically creates a new instance of the `RsumQuery` compartment in the query-factory. The new query-factory provides functions for creating new elements as used in lines 2 and 3. Creating an object automatically creates a new `QueryObject` in the owned `RsumQuery` instance. The objects can now be connected with relations (line 4). The interfaces in the query-factory provide functions that are used to modify these elements and automatically create attribute filters. However, to change the properties of the `QueryObjects` of these elements, it is necessary to go one step into the connected `QueryObject` (line 6). After the query has been completely created, it can be executed on all elements of the underlying model using the *run* method (line 8). In the *run* method, the correctness of the query is proven before the computation of the query is started.

Listing 2 shows how existing objects can be used to create Query 4 in pseudocode. In line 1, an instance of the `RsumQuery` compartments is created. In line 2 to 4, objects are shown to which the user already has access. These objects describe two `InternalElements` and the ownership relationship between them. Adding these objects to the query is shown in line 6 to 8. In line 10, the *multi* property of the *ie2* object is set to three and in line 11 the *return* value is set to false. Query 4 searches for all `InternalElements` containing more than three `InternalElements`. In line 13 the query is than executed. The first type of query creation uses the query-view as a wrapper around the process as shown

```
RsumQuery q4 = new RsumQuery("Query␣4")
InternalElement ie1 = /*InternalElement in RSUM*/
InternalElement ie2 = /*InternalElement in RSUM*/
SUCtoIE ieToIe = /* Relational Compartment between
      ie1 & ie2 */
//Add Query Roles
QueryObject r0 = q4.addQueryRole(ie1)
QueryObject r1 = q4.addQueryRole(ie2)
QueryObject r2 = q4.addQueryRole(ieToIe)
//Set Properties
r2.multi = 3 //means > 2
r2.returned = false
//Run Query
Set[Set[Objects]] result = q4.run()
```

Listing 2. Creation of Query 4 in RSUM with objects from RSUM.

```
1 natural join aml.InternalElement with aml.
      InternalElement {
2   keep attributes aml.CAEXObject.name
3   keep attributes aml.CAEXObject.id
4   keep outgoing aml.InternalElement.has {}
5 }
```

Listing 3. ModelJoin Representation of Query 4.

in Listing 2 to simplify the process for the user.

The execution of the queries leads to a set of sets of untyped objects which can be modified again, when the query is not read-only. To make this possible, a textual representation can be created from a query to create a viewtype in which new elements can be created and old elements can be modified or deleted. Listing 3 shows how Query 2 looks like as a ModelJoin representation. Since ModelJoin always starts with a join statement, the query produces a neutral natural join that connects a type to itself. The `RsumQuery` compartment creates this representation by displaying all query objects and their relations.

The ModelJoin representation for Query 4 is illustrated in Listing 3. Since only `InternalElements` are considered in the query, a natural join between these elements is created first. Since this step creates mental overhead, a ModelJoin query must always has such a join statement. It is planned to develop a modified language for this approach that no longer requires this join statement. In lines 2 to 4, the attributes and references are determined that should still be contained in the viewtype. References are the ones used in the query, whereby all attributes of the elements are included in the ModelJoin query as *name* and *id* here.

However, we currently only support a subset of ModelJoin with a minimum number of OCL constrains, i.e., a viewtype created from this query does not know that the link between the `InternalElements` must exist at least three times. For this reason, a view only displays elements that are contained in the result set of the query. If a query has no special properties, it is possible to omit the extra step about executing the query. In simple queries like Query 7, where no special filter operators are used, it is possible to omit this computation step in the query. The created viewtype can subsequently simply display any element from the underlying model. In addition, this query shows the usage of the natural type `HelperCAEXObject`. Without this type, no instance of the `CAEXObject` may be created for the query. The overall process shows how a ModelJoin representation can be created using the presented query/view concept without having any special knowledge about the underlying model. However, since join statements cannot be represented in the query concept, this is a point to extend the query/view concept in the future.

VI. Related Work

The concept of defining queries by-example has been used since 1975 [17] when a concept was introduced for using database tables as an example to formulate queries. A notation was introduced, which is still in use. However, the by-example concept is rarely used and often textual query, view, and transformation languages are developed. On XML documents query languages like XQuery, XSLT, XPath etc. have prevailed. These languages work on the abstract tree structures and are applicable to general XML models. The disadvantage of XML based languages is the learning of concepts like path descriptions and navigation in documents.

The query/view language presented in this paper is based on graph pattern structures, therefore some of these languages [3] are presented here. FUJABA [18] (From Uml to Java And Back Again) is a graph replacement system where manipulations on the object structure are graphically specified and then performed. Cypher is a textual SQL-like language used by Neo4J. Other graph-based languages are SPARQL, GraphQL, and Gremlin. SPARQL, like Cypher, is a textual language based on SQL. It works declaratively and was developed by the W3C consortium to generate queries on RDF. GraphQL is developed by Facebook, where users describe the structure of the data they want to query. Finally, Gremlin is a DSL that traverses the graphs and can be easily programmed in native languages like Java, Python, etc. In contrast, AutomationQL [1] is a by-example query language based on AML. Since, it is based on the AML metamodel, it is not possible to use the language in other areas. The concept to define languages by-example is already used in the area of transformation languages [19]. It is possible to define by-demonstration [20], [21] or by-correspondence [22] transformations. By-demonstration means that the transformations are demonstrated in a model editor by the user, whereby the editor records the changes and creates a transformation. This process is already used for in-place and out-place transformations. Current approaches usually take a semi-automatic approach, since the rules usually must be adapted at the end. If one wants to define transformations by-correspondence, the user defines the source, target, and correspondence model, whereas the comparison describes the final query. Usually several of these relationships are defined to cover critical cases.

Finally, we consider related work in the area of view languages. In the work of Brunelier *et al.* [23] an overview of current view approaches is presented. The approaches are examined regarding their query and view languages. The ModelJoin [2] approach was already mentioned in Sect. V and uses a textual SQL-like syntax where complicated comparison operators are described using OCL. EMF Views [4] also uses a SQL-like language and describes complex expressions with the Epsilon Constraint Language (ETL). Epsilon Merge Language [24] connects models and uses the complete collection of Epsilon languages like the Epsilon Object Language (EOL). In Epsilon, all DSLs are linked to each other and use similar concepts, whereby each language is adapted to its field of application. The discussed languages of this paragraph as well as VIATRA [25] all have their own textual languages to define views. In contrast, Triple Graph Grammars (TGGs) [26] work with a graphical notation to represent relationships between models and views.

VII. Conclusion

In this paper, we have presented a generic approach for a by-example query/view language. This approach is based on the role concept and rests upon a graph pattern-based query language. The concept supports the formulation of positive and negative graph patterns and can search for specific structures in the underlying model. In addition, we have shown the applicability of this approach using the example of AML, where the concept is implemented upon the RSUM approach and queries are executed on the underlying structures. The query language is also used to create viewtypes and may be applied to other approaches as well.

Although many different queries are already supported, the expressive power of the concept must be extended in the future to deal with, e.g., properties such as the order and the merging of elements in the query/view concept. This would potentially allow in the future to represent all properties of the ModelJoin language through our query by-example concept and to create other existing viewtype or graph query languages from this concept. Moreover the current implementation does not automatically optimize the queries to speed up the generation of the result set. This is left as future work. Finally, the question arises how to adapt the by-example query/view concept to generate and implement a by-example transformation concept in order to make the definition of transformations easier and to minimize the learning effort in this area.

References

[1] M. Wimmer and A. Mazak, "From AutomationML to AutomationQL: A By-Example Query Language for CPPS Engineering Models," in *IEEE 14th International Conference on Automation Science and Engineering (CASE)*, 2018, pp. 1394–1399.

[2] E. Burger, J. Henss, M. Küster, S. Kruse, and L. Happe, "View-based model-driven software development with ModelJoin," *Software & Systems Modeling*, vol. 15, no. 2, pp. 473–496, 2016.

[3] F. Holzschuher and R. Peinl, "Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J," in *Joint EDBT/ICDT Workshops*. ACM, 2013, pp. 195–204.

[4] H. Bruneliere, J. G. Perez, M. Wimmer, and J. Cabot, "EMF Views: A View Mechanism for Integrating Heterogeneous Models," in *Conceptual Modeling*, P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, and Ó. Pastor López, Eds. Springer, 2015, pp. 317–325.

[5] R. Drath, A. Lüder, J. Peschke, and L. Hundt, "AutomationML - the glue for seamless automation engineering," in *ETFA*, 2008, pp. 616–623.

[6] T. Mayerhofer, M. Wimmer, L. Berardinelli, and R. Drath, "A Model-Driven Engineering Workbench for CAEX Supporting Language Customization and Evolution," *IEEE Transactions on Industrial Informatics*, vol. PP, no. 99, pp. 1–11, 2017.

[7] M. ISO, "Systems and software engineering–architecture description," ISO/IEC/IEEE 42010, Tech. Rep., 2011.

[8] T. Goldschmidt, S. Becker, and E. Burger, "Towards a Tool-Oriented Taxonomy of View-Based Modelling," in *Modellierung 2012*, E. J. Sinz and A. Schürr, Eds., vol. P-201. Gesellschaft für Informatik e.V. (GI), 2012, pp. 59–74.

[9] C. Atkinson, D. Stoll, and P. Bostan, "Orthographic Software Modeling: A Practical Approach to View-Based Development," in *Communications in Computer and Information Science*. Springer, 2010, pp. 206–219.

[10] C. Werner and U. Aßmann, "Model Synchronization with the Role-oriented Single Underlying Model," *MART Workshops*, pp. 62–71, 2018.

[11] M. E. Kramer, E. Burger, and M. Langhammer, "View-centric Engineering with Synchronized Heterogeneous Models," in *1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, 2013, pp. 5:1–5:6.

[12] J. Meier and A. Winter, "Model Consistency ensured by Metamodel Integration," in *6th International Workshop on The Globalization of Modeling Languages (GEMOC)*, 2018.

[13] F. Steimann, "On the representation of roles in object-oriented and conceptual modelling," *Data & Knowledge Engineering*, vol. 35, no. 1, pp. 83–106, 2000.

[14] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann, "A Metamodel Family for Role-Based Modeling and Programming Languages," in *Software Language Engineering*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8706, pp. 141–160.

[15] T. Kühn, S. Böhme, S. Götz, and U. Aßmann, "A combined formal model for relational context-dependent roles," in *International Conference on Software Language Engineering*. ACM, 2015, pp. 113–124.

[16] M. Leuthäuser and U. Aßmann, "Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-based Programming," in *Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, ser. MORSE/VAO'15. ACM, 2015, pp. 25–33.

[17] M. M. Zloof, "Query by Example," in *National Computer Conference and Exposition*. ACM, 1975, pp. 431–438.

[18] L. Grunske, L. Geiger, A. Zündorf, N. Van Eetvelde, P. Van Gorp, and D. Varró, *Using Graph Transformation for Practical Model-Driven Software Engineering*. Springer, 2005, pp. 91–117.

[19] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, *Model Transformation By-Example: A Survey of the First Wave*. Springer, 2012, pp. 197–215.

[20] Y. Sun, J. White, and J. Gray, "Model Transformation by Demonstration," in *Model Driven Engineering Languages and Systems*, A. Schürr and B. Selic, Eds. Springer Berlin Heidelberg, 2009, pp. 712–726.

[21] P. Langer, M. Wimmer, and G. Kappel, "Model-to-Model Transformations By Demonstration," in *Theory and Practice of Model Transformations*, L. Tratt and M. Gogolla, Eds. Springer, 2010, pp. 153–167.

[22] D. Varró, "Model Transformation by Example," in *Model Driven Engineering Languages and Systems*, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Springer Berlin Heidelberg, 2006, pp. 410–424.

[23] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer, "A feature-based survey of model view approaches," *Software & Systems Modeling*, 2017.

[24] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Merging Models with the Epsilon Merging Language (EML)," in *Model Driven Engineering Languages and Systems*, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Springer, 2006, pp. 215–229.

[25] C. Debreceni, A. Horváth, A. Hegedüs, Z. Ujhelyi, I. Ráth, and D. Varró, "Query-driven Incremental Synchronization of View Models," in *2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, 2014, pp. 31:31–31:38.

[26] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr, "Efficient Model Synchronization with View Triple Graph Grammars," in *Modelling Foundations and Applications*, J. Cabot and J. Rubin, Eds. Springer, 2014, pp. 1–17.