Graph Query by Example[⋆]

Gábor Bergmann, Ábel Hegedüs, György Gerencsér, and Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2.
{bergmann,hegedusa}@mit.bme.hu, gyorgy.gerencser@gmail.com,
varro@mit.bme.hu

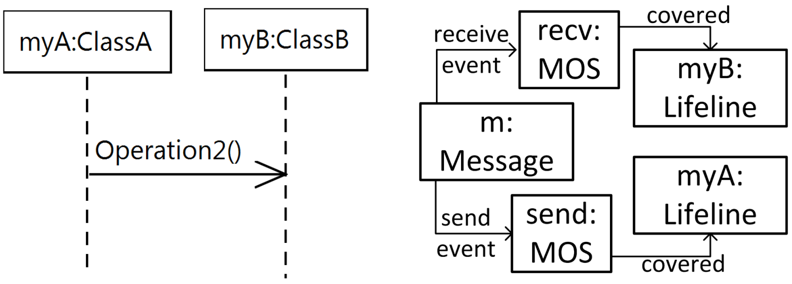
Abstract. Model-driven tools use model queries for many purposes, including validation of well-formedness rules, specification of derived features, and directing rule-based model transformation. Query languages such as graph patterns may facilitate capturing complex structural relationships between model elements. Specifying such queries, however, may prove difficult for engineers familiar with the concrete syntax only, not with the underlying abstract representation of the modeling language.
The current paper presents an extension to the EMF-IncQuery model query tool that lets users point out, using familiar concrete syntax, an example of what the query results should look like, and automatically derive a graph query that finds other similar results.
Keywords: by example, model query, graph pattern, EMF-IncQuery

1 Introduction

Model-driven Engineering (MDE) approaches treat *models* as primary artifacts of the engineering process, relying on automated model processing steps. Models are usually thought of as typed, attributed graphs. This underlying structure is defined by the metamodel of the modeling language and is called the *abstract syntax*. On the other hand, the preferred way the model is presented to (and edited by) humans is in the form of visual diagrams, textual notations, tree structures, etc., called the *concrete syntax*. The two representations can have substantial differences, e.g., an edge in concrete syntax may correspond to a node in abstract syntax, or to a structure of several elements (see Fig. 1).

Model queries are important components in model-driven tool chains: they are widely used for specifying derived features, well-formedness constraints, reports, and guard conditions for behavioural models, design space rules or *model transformations*. Although model queries can be implemented using a general-purpose programming language (Java), specialized query languages may be more

[⋆] This work was partially supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP 4.2.4. A/-11-1-2012-0001 'National Excellence Program', and by the CERTIMOT (ERC_HU-09-01-2010-0003) and EU FP7 MONDO (ICT-2013.1.2), CECRIS (FP7-PEOPLE-2012-IAPP 324334) projects partly during the fourth author's sabbatical.



(a) Concrete syntax (b) Excerpt of abstract syntax

Fig. 1: An UML2 Sequence Model (MOS = MessageOccurrenceSpecification)

concise and easier to learn, among other advantages. Query approaches that can express complex graph structures within the model graph can be called *graph queries*; they are often very useful for defining complex well-formedness constraints. Modeling platforms (e.g., the Eclipse Modeling Framework *(EMF)* [1]) support various graph query languages; some of the more declarative ones (such as EMF-IncQuery [2]) are inspired by *graph patterns* [3].

Queries formulated in a query language refer to metamodel concepts. However, as argued in [4], modelers are typically more familiar with the concrete syntax than with the abstract syntax. Thus, they may have substantial difficulties in formulating a query. Following the idea of *query by example* (QBE [5]), we present a technique that lets users point out, using familiar concrete syntax, an example of what the query results should look like, in order to automatically derive a graph query that finds other similar results. The proposed extension of EMF-IncQuery addresses the following important challenges:

Concrete syntax. The user should be able to specify a query using the concrete syntax (though the metamodel has to be at least comprehensible).

Selection. The proposed solution should allow the user to point out the example as a *selected* part of an existing, larger instance model. It should not be necessary to create a new instance model from scratch just for the sake of providing the example. It may not even be possible, if the example does not constitute a complete instance model on its own that the editor allows to create.

Discovery. The elements of the instance model that are selected to form the example do not directly determine a graph query that would retrieve them from the entire model. The proposed solution should figure out how the group of selected elements is connected in the graph structure of the model (it is possible that there are no direct connections, only through elements that may not even be directly visible in the concrete syntax), and construct a query that instructs the query engine to retrieve groups of elements that are similar to the selected ones and are connected with each other in a similar fashion.

The query formalism is introduced in Sec. 2. An overview of the proposed solution is given in Sec. 3, and an application case study is elaborated in Sec. 4.

Related “by example" techniques are reviewed in Sec. 5. The merits and weaknesses of the approach are discussed in Sec. 6.

2 Graph Patterns and EMF-IncQuery

The EMF-IncQuery framework [2] aims at the efficient definition and evaluation of model queries over EMF-based models. The query language [6] supports conjunction, disjunction, negation, quantification, even advanced features such as attribute expressions, transitive closures and aggregation. Here we focus only on the most basic idea that the language is built on.

A *graph pattern* is a graph-like structure, labeled by node and edge types from the metamodel, that represents a condition (or constraint) matched against a large instance model graph. A *match* of a graph pattern is a homomorphism from the pattern to the graph model, representable as a tuple (one image for each pattern node). The result of an (unbound) model query is the *match set*, a relation (in the mathematical sense) formed by the set of all matches.

For instance, assume that our goal is to enforce company-specific design conventions in UML2 models. One such rule places restrictions on messages passed between lifelines in Sequence Diagrams. In order to validate models, we need to specify a model query that finds all messages that *violate* the rule. A graph pattern isomorphic to the graph structure excerpt shown in Fig. 1b will match all messages passed between lifelines. Further attribute or graph constraints (see Sec. 4) would check whether the rule is actually violated.

For purposes of this paper, we assume that meaningful graph patterns are connected graphs (avoiding Cartesian products). As the paper focuses on the graph nature of queries, we have omitted the treatment of attributes for brevity.

3 Overview of the Approach

The proposed QBE mechanism is realized as a plugin of the Eclipse [7] environment, where it aids in specifying EMF-IncQuery-based graph queries against EMF models. The steps of the QBE workflow are illustrated by Fig. 2, and are explained in the following paragraphs.

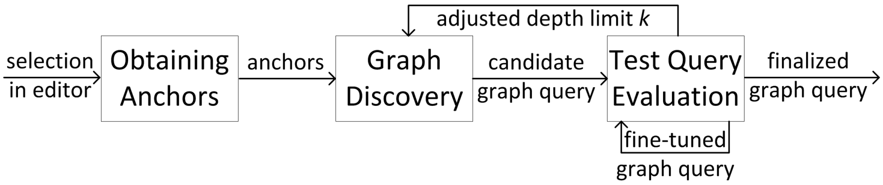


Fig. 2: Proposed QBE Workflow

3.1 Obtaining the Anchors

The first step is obtaining a set of selected model elements (the *anchors*) that serve as an example for query results.

The models can be loaded and then displayed in concrete syntax by arbitrary Eclipse-based domain-specific or generic editors (or views). The user can select one or more model elements using the selection functionality of the editor, and then invoke the QBE plugin from the user interface. Using the general Eclipse-wide selection mechanism, the QBE plugin can retrieve the set of selected elements upon invocation.

In some editors or views, these selected elements are the EMF model elements themselves. In other cases, such as in many diagram editors, the selection will actually contain notation objects of the concrete syntax that may in turn wrap model elements. Therefore, compatibility with such editors requires a corresponding Eclipse plug-in that unwraps the editor-specific selection to obtain the anchor model elements. Note that the source code of the editor does not have to be modified to achieve this. Furthermore, many editors are developed using editing frameworks such as GMF or Graphiti, where a single generic plug-in can perform this unwrapping, so there is no need for creating a separate plug-in for each modeling language.

3.2 Graph Discovery

The overall goal is to construct a graph query that finds instance model elements that are related to each other in the model graph similarly to the relationship between anchors. Therefore the QBE plug-in must analyze how the anchors are related to each other. They may directly reference each other, or there may be longer paths connecting them. The task of the *discovery phase* is to explore the model graph in the neighborhood of the anchors to find these paths.

A *connecting path* is a graph path in the model that starts and ends at anchor nodes, and traverses zero or more other nodes along graph edges (navigated in either direction). Not all connecting paths are *interesting*: it may be possible to find a very long “detour" path between two elements that are actually adjacent. Our approach therefore focuses on paths no longer than k hops. The key discovery routine $Discover_k$ starts a k-depth-limited search from each anchor node to find all interesting paths. The union of these paths will form the candidate graph query; note that each node and edge (irrespective of the direction of traversal) is included at most once, even if incorporated in multiple interesting paths.

If the user so wishes, a value for k can be provided manually. However, a minimal candidate for k can be determined automatically based on the observation that the graph pattern must form a connected graph. The automatic discovery routine $Discover_*$ starts by invoking $Discover_0$, $Discover_1$, $Discover_2, \ldots$, until either the resulting candidate graph pattern becomes connected, or the traversal proves (by not hitting the depth limit) that the anchors reside in separate components of the model graph (in which case there is no connected graph pattern that would match them). Thus $Discover_*$ forms the graph pattern from interesting paths according to the smallest valid k; the user may manually increase the value of k if needed.

Note that if not all elements of the abstract syntax are directly selectable in an editor, there may be *dead end* elements that do not lie on any path connecting selectable elements, thus they are not discoverable by the proposed approach.

3.3 Query Testing and Feedback

Trivially, any graph query constructed by the discovery routines will always yield the tuple of selected anchors as one of its matches on the example model. Which other tuples the query results will contain, depends on the actual graph pattern; from the point of view of the user, there may be some false positives (if the query is not restrictive enough) and false negatives (if the query is too restrictive). Therefore after the discovery phase, the user is presented with the constructed graph pattern, and the complete query results in the example model. If the results deviate from intentions, the user can interactively change the query and test the new version in an iterative *fine-tuning phase*.

The tool offers fine-tuning options including (1) the already mentioned adjustment of depth limit k, (2) purging some of the discovered paths, (3) binding the concrete attribute values found on the retrieved nodes as query constraints (analogously to [5]), (4) substituting supertypes instead of the actual types of the example, or (5) manually adding arbitrary constraints.

4 Case Study

The purpose of this case study is to demonstrate which elements of graph queries the QBE approach can derive, and which ones it can't produce automatically.

Following Sec. 2, the goal is to formulate a query over UML2 Sequence Models, in order to enforce a company-specific design convention on test scenarios. The design rule says that in the Sequence Model, the GUI Package must call operations from the core Engine Package *asynchronously* (to avoid user interface freezes). The query should find violations of this rule, i.e., synchronous messages.

4.1 Deriving the Graph Pattern of Fig. 1b

The first step is capturing the relationship between the Lifelines and the Message.

Assume that we open the UML2 Sequence Diagram of Fig. 1a in an editor, select the two Lifelines as anchors, and then invoke the QBE plug-in. There is no direct connection between the two anchors; $Discover_*$ stops at $k = 2$, as the two selected Lifelines belong to the same UML2 Interaction. However, "lifelines in the same Interaction" is not the query we are interested in, so we need to continue the fine-tuning feedback loop. By manually raising k to 4, the path depicted in Fig. 1b is discovered as well.

There may be other paths not longer than 4 hops that connect the anchors; e.g., at $k = 3$, the QBE tool discovers that the MessageOccurrenceSpecifications on the Lifelines also belong to the same Interaction. If such paths over-restrict the match set, they lead to false negatives, and should be removed from the resulting query. Note that two Lifelines exchanging a Message implies that they reside in the same Interaction, so the presence of the first discovered 2-hop path through the containing Interaction does not actually restrict the match set, thus its removal is optional. The same is true for the aforementioned 3-hop paths.

Alternatively, if the Message had been originally selected as an anchor in addition to the two Lifelines, $Discover_*$ would have come up with the right path at $k = 2$. The Interaction containing all three anchor elements would be discovered as well; once again, this latter part of the query can be removed.

4.2 Adding Attribute Restrictions

The query constructed above is incomplete: it ensures neither that the message is Synchronous, nor that the source and target Lifelines correspond to Classes in Packages describing the GUI and Engine, respectively. If we perform an evaluation of the query for testing purposes, we might find false positive matches (e.g., if the model contains some synchronous messages as well).

As described in Sec. 3, the actual attribute values found in the example are offered automatically in the fine-tuning phase as potentially useful constraints. In particular, since the Message is one of the nodes found in the discover phase, the QBE tool lists its attribute slots and values (e.g., the value of the "name" attribute is the character string "1:Operation2_Message") upon request. In our case, the value "Synchronous Call" of the attribute "Message Sort" shall be selected; it will then be added as an attribute restriction.

Not all attribute restrictions can be added automatically like this. Assuming that we have already further developed the query to include the Packages of the Classes of the two Lifelines (this will be discussed in the next Section), we merely need to identify the GUI and Engine Packages. There may be multiple GUI and multiple Engine packages within a single project, and we want the query to be compatible with models of different company projects. Unfortunately, UML2 Packages have no simple attribute indicators for their designation as belonging to either group, so that there is no fixed attribute value that all GUI Packages have in common, which the QBE tool could offer automatically based on the example. We have to assume that according to company conventions, GUI Packages can be recognized by their package URI containing a path segment "gui", while Engine Packages have the "engine" segment in their URI. It is possible to manually add the appropriate attribute restrictions to the query, but the proposed approach cannot automatically generate such regular expression checks based on examples.

4.3 Connecting Lifelines and Packages

Finally, the connection between a Lifeline and the corresponding Package is the same in case of both Lifelines. Thus it can be captured as a separate helper query, which is called (reused) twice in the main query, using the compositionality of the EMF-IncQuery language. The QBE tool does not compose patterns automatically, but such calls are easy to add in the fine-tune phase.

Unfortunately, UML2 diagrams do not commonly feature Lifelines and Packages in the same view; so with a given editor implementation, it might not be possible to select these two elements from the concrete syntax at the same time. However, one can select a Lifeline, the associated Class and the Package in the tree outline of the model. Then one can successfully apply QBE: at $k = 2$, the tool discovers that the Lifeline *represents* a Property, whose *type* is the Class, which is a *packaged element* of the Package. Thus here we lose the advantage of using the concrete syntax only, but the tool is still a great help in navigating through parts of the metamodel that are likely unfamiliar to typical UML2 users.

5 Related Work

QBE is an idea that originally came up for database systems [5], where the user could specify concrete values for attributes and thus constrain which entities are to be retrieved. We aim to extend the original idea in such a way to meet challenges we see in the context of MDE: the graph nature of models and model queries, as well as the difference between concrete and abstract syntax.

A survey [4] has found that MDE has significant ongoing research into specifying *Model Transformations by Example* (MTBE), where various machine learning techniques are applied to derive model transformations from inputs and outputs of example executions. Rule-based model transformations use model queries as rule guards; in fact, model queries can be considered a degenerate case of exogenous model transformations (creating query results as the target model). Thus, in theory, existing approaches could be applied to queries as well; however, this would be impractical. First, most approaches would rely on a computationally expensive machine learning phase. Second, the user would have to build query results as example target models. Third, some approaches cannot derive arbitrary graph queries as transformation rule preconditions, just a single source model element; while other approaches can produce more complex queries but require the user to provide explicit source-target correspondence for each element, with significant overhead in effort.

The most directly related work [8] focuses on well-formedness rules, which are one kind of model query. Taking positive examples (conformant models) and negative ones (violating models) as input, the proposed technique applies machine learning (genetic programming) to come up with the model query that best discriminates the two sets. Our approach avoids applying a resource-intensive machine learning search, but focuses on a query sublanguage with narrower expressiveness. Furthermore, our approach is simpler as selecting a single example within a larger model is sufficient; but since no counterexamples and additional examples can be given, fine-tuning of the results is only possible manually.

6 Discussion and Conclusions

The presented approach extends [5] by discovering the underlying graph structure of an example highlighted within a model displayed in concrete syntax. The QBE tool can be used even when it is difficult to intuitively construct queries using the abstract syntax. Resulting queries can be tested by interactive evaluation, but fine-tuning still requires that the metamodel is at least comprehensible.

There is an option to consider concrete attribute values as virtual nodes and value slots as edges. This may be useful to discover that two elements have the same name. In most cases, however, this option is not useful (e.g., we should not consider two Associations connected if they have the same multiplicity).

Future work shall include the automatic suggestion of negative conditions based on the example. For example, if the metamodel would allow a certain edge to point between two anchors (or intermediate nodes discovered along interesting paths), but that edge is not present in the provided example, then an automatically offered fine-tuning option could be to add the non-existence of the edge as a negative constraint to the pattern. A further direction to explore would be to optionally generate queries from multiple examples (preferably still without expensive soft computing techniques), offering gradual transition towards complex by-example approaches.

References

1. The Eclipse Foundation: Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.
2. Ujhelyi, Z., Bergmann, G., Ábel Hegedüs, Ákos Horváth, Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: An integrated development environment for live model queries. Science of Computer Programming (0) (2014) –
3. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
4. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: A survey of the first wave. In: Conceptual Modelling and Its Theoretical Foundations. Volume 7260 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 197–215
5. Zloof, M.M.: Query-by-example: The invocation and definition of tables and forms. In: Proceedings of the 1st International Conference on Very Large Data Bases. VLDB '75, New York, NY, USA, ACM (1975) 1–24
6. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A graph query language for EMF models. In: Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings. Volume 6707 of Lecture Notes in Computer Science., Springer, Springer (2011) 167–182
7. The Eclipse Foundation: The Eclipse Project. `http://www.eclipse.org`.
8. Faunes, M., Cadavid, J., Baudry, B., Sahraoui, H., Combemale, B.: Automatically searching for metamodel well-formedness rules in examples and counter-examples. In: Model-Driven Engineering Languages and Systems. Volume 8107 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 187–202