A Query Language With the Star Operator

Johan Lindqvist
Torbjörn Lundkvist
Ivan Porres
TUCS Turku Centre for Computer Science
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5A, FIN-20520 Turku, Finland
{johan.lindqvist,torbjorn.lundkvist,ivan.porres}@abo.fi
Abstract

Model pattern matching is an important operation in model transformation and therefore in model-driven development tools. In this paper we present a pattern based approach that includes a star operator that can be used to represent recursive or hierarchical structures in models. We also present a matching algorithm, motivating examples and we discuss its implementation in a modeling tool.

Keywords: Visual languages, Model transformation, Graph query, Graph subgraph matching
1 Introduction

In the context of model-driven software development, a query language is used to find parts of a model that fulfill some given constraints. A query language is a fundamental element in rule-based model transformation languages. Query languages are also used to define model constraints, where a model is invalid if it does not satisfy the query. Finally, a query language combined with different aggregation operators can be used to compute metrics.

We consider that query languages should be declarative, in the sense that they should state what to search for in a model, but not how to perform the actual search. Also, we are interested in expressive query languages that can define complex patterns in a succinct way. The Object Management Group (OMG) proposes a standard for a model transformation language called Query-View-Transform (QVT) [17], that contains a query language. The OMG Object Constraint Language [16] can also be used to query models.

In this article we explore the idea of a query language based on graph matching. Understanding modeling languages and models as graphs brings many benefits to our approach since the application of graph theory to computer science provides a solid foundation to model-driven development tools, specially in the area of model transformations [19]. Successful approaches to graph transformation in the context of software development are presented for example in [4, 22, 2].

Probably, the simplest graph matching approach is the one based of subgraph isomorphism. A software model and a query are represented as graphs and a match of the query is any subgraph of the target model that is isomorphic to the query. However, this approach is not sufficient to express many queries succinctly. Therefore, it has been extended to include, negative application conditions [7] and multi-objects [20]. Still, these extended forms of graph pattern matching may not be able to express many interesting queries. Many computer languages contain hierarchical and recursive structures. Examples of these structures in UML [18] are package containment hierarchies in class diagrams or state hierarchies in statecharts. As a consequence, we often need to specify queries to match recursive structures where the number of elements to match it is not known a priori.

In this article, we propose a new query language that support what we call the star operator. This operator conceptually resembles the Kleene star operation over sets of strings. Used in our query language, it can match against a subgraph that appears repeatedly zero, one or more times in a graph representing a model. In our opinion, when we combine the star operator with the isomorphism operator that denotes isomorphic matches and the negation operator, that denotes the absence of a match, we can express complex queries using rather short and intuitive pattern.

We proceed as follows: In Section 2 we describe the basics of our query language and provide some examples of queries for the UML language. Section 3 presents an overview of a matching algorithm for this query language. The next section discusses the practical implementation of the approach in an experimen-
tal modeling tool. Finally, we conclude in Section 5 with a description of future work.

2 Regions in a Pattern

In this section we will describe the concept of regions in a pattern, and introduce three operators that can be applied regions: the isomorphic, star and negation operator.

A pattern consists of a typed and directed graph annotated with information necessary to perform a query. The graphs in the patterns are constructed according to a metamodel. The pattern graph can be compared against a target graph. A match occurs if all nodes and edges of the pattern graph can be mapped to a subgraph of the target graph, with respect to the annotations of the pattern graph. The result is a mapping of the pattern and target graphs, which allows the nodes and edges of a pattern graph to be bound to the target graph.

In order to increase the expressibility of a pattern, we have introduced the concept of operators and regions in a pattern. A region is defined over a connected subgraph of a pattern, such that a node belongs only to one region. In our approach we have defined a region as the scope of a matching operator. As a consequence of this, a pattern consists of several non-overlapping regions, where each region is associated with an operator. Edges can still be connected between two nodes in separate regions. Edges that cross the boundaries of a region are called connection points since they connect two regions. These connection points can be computed from the pattern graph and are used, depending on the operator associated with the region, to validate whether the region fulfills the specific requirements of the operator associated with the region.

Next, we will describe the definition of the isomorphic, negation and star operator applied over a region.

2.1 Isomorphic Regions

The semantics of an isomorphic region as part of a pattern graph, is that it is possible to find a subgraph in the target model that is isomorphic to the region. A pattern graph can have several isomorphic regions. However, if a pattern consists only of isomorphic regions, the regions could be merged without affecting possible mappings retrieved when matching the pattern against a target graph.

2.2 Negative Regions

The semantics of a negative region as part of a pattern graph, is that an occurrence of all nodes and edges of a negative region in the target results in a failed match.
Since the negation operator is always defined over a region in the pattern, it is possible to model complex negative conditions that involve several nodes and edges. A similar approach where the negation operator is defined over regions in a graph can be found in [7].

2.3 Star Regions

In order to be able to describe patterns with recursive or hierarchical structures, we have introduced the concept of a star operator. The star operator in a pattern is conceptually similar to the star operator in Kleene algebra [11]. A pattern with a star region can be used to generate a set of patterns where the contents of the star region has been inserted an arbitrary number of times and replaced by an isomorphic region. Analogous to the Kleene star operator, the generation of patterns begins with a pattern where the subgraph is not inserted. We will discuss the constraints that apply to valid star regions later in this section.

The structure of the subgraph represented in a star region must follow some specific requirements. This is necessary, as the patterns used for defining a query as well as new patterns that are generated by expanding the star regions into several subgraphs must preserve the structure defined by the metamodel. To ensure that the star region can be expanded, the star region needs to have two edges or connections crossing the border of the region. The connections are always defined in pairs, where one edge is incoming and the other edge is outgoing with respect to the nodes in the star region. The connections define the position in the pattern graph where subgraphs generated from the star region are inserted. In addition, the star region can contain other nodes and edges which are instantiated in each generated subgraph.

Figure 1 shows the generation of patterns based on a pattern with a star region in more detail. In the top part of the figure an example pattern $G$ with two isomorphic regions $R_1$ and $R_3$ and a star region $R_2$ is shown. From this figure we can see that $R_2$ consists of two interconnected nodes, $2'$ and $3'$. There are also two directed edges labeled $m$, one incoming edge from node $1'$ in $R_1$ to $2'$ in $R_2$ and one outgoing edge from node $3'$ in $R_2$ to $4'$ in $R_3$. The bottom part of the figure shows three different patterns that can be generated based on $G$. The generation of $G_1$ is done by applying a production that replaces $R_2$ with the empty graph, and creates a new edge $m$ from $1'$ to $4'$. The pattern $G_2$ is retrieved by replacing the previously rewritten edge $m$ in $G_1$ with an instance of the star region $R_2$ and the edges in the connection points are rewritten. Similarly, the pattern $G_3$ is retrieved by again replacing one of the rewritten edges with a new instance of the star region. To make the figures clearer, all rewritten edges are drawn with a wider stroke. These patterns can now be used to find a mapping to a target graph.

Using this approach, it is possible to use a single pattern to describe recursive and hierarchical structures by generating a set of patterns that can be compared to a target graph using subgraph isomorphism.
A star region can be seen as a extension of the concept of multi-objects, or set nodes as defined in PROGRES [20]. Where a multi-object can express multiple instances of a single node, the star region can express multiple instances of a subgraph. We have extended the concept of multi-objects by defining star regions in the query graph, where all connections of the nodes within or at the border of the region are explicit. This extension is also partly due to the fact that a multi-object can have an edge to another multi-object, but it is unclear whether the edge represents a single edge or multiple edges. Other related approaches are the works of graph transformations with variables presented in [14, 9, 8]. Karsai and Agrawal present in [10] and approach that allow cardinalities in individual nodes, but it is unclear whether this approach supports whole regions.

2.4 Examples

In this section we will present some examples that illustrate how the CQuery language can be used define queries to match common model structures in UML.
We have chosen to display the both the patterns and matching model fragments using the abstract syntax, which is an object graph syntax similar to UML object diagrams rather than the concrete syntax of the target modeling language, since the concrete syntax hides information about relations between the objects. In a tool environment, however, creating the queries using the concrete syntax of the modeling language can be beneficial.

In the examples we will use a slightly simplified version of the UML 1.4 metamodel, which is shown in Figure 2.

2.4.1 UML Generalizations

An example of a query with two isomorphic regions and a star region is illustrated in the left part of Figure 3. The star region is marked with a dashed rectangle with the ‘*’ symbol, and the isomorphic regions with rounded rectangles and a ‘=’ symbol. The connection points for the star region are marked with circles at the border of region. The star region in this case is formed over a pair of a UML Generalization and a Class. A Generalization references the superclass with a parent relationship and the subclass with a child relationship, these relations connect the star region to the adjacent isomorphic regions. On the right hand side of the figure two different UML model fragments in object diagram syntax are shown that are matches of the query on the left hand side. Here, the mappings between the pattern and the target models are shown using corresponding object names, i.e. (1’) in the pattern corresponds to (’1) in the target. As we can see from this figure, the generalization and the class in the star region was matched once in the first model fragment, and twice in the second model fragment. The pattern can be matched to targets where the star region is mapped to the empty set. This case is not illustrated in the figure. However, that particular case had implied that Class (’1) has exactly one subclass (’5).
Figure 3: (Left) An example of a query with a star region defined over a class and a generalization and the \textit{parent} relation. (Center, Right) Two model fragments that match the query defined on the left, shown as object diagrams. The mapping is indicated with the corresponding numbers.
2.4.2 UML StateMachines

The second example in Figure 4 shows a pattern that can be used to query a UML model for a state machine with a transition between two state vertices, where the state vertices are transitively owned by any number of composite states. The state machine owns a composite state in \texttt{StateMachine.top} that transitively can own other state vertices in the \texttt{CompositeState.subvertex} slot. Transitions, however, are always owned by the state machine, and has associations to two state vertices in \texttt{Transition.source} and \texttt{Transition.target}.

The pattern described here is rather complex, as we can identify three different star regions. Each of the three star regions consists of one composite state and is connected to the other regions using the \texttt{CompositeState.subvertex} relations to the other regions. This pattern describes that the two state vertices \(5'\) and \(8'\) that connect to the transition \(9'\) in the figure, can be nested in an arbitrary number of common container composite states (star region with composite state \(3'\)). Additionally, each of the state vertices can independently be contained by any number of composite states ((4' and 7')). It must be noted, however, that there are three connections in the star region with composite state \(3'\). This is possible, since a composite state can have any number of subvertices.

The right side of Figure 4 shows two possible matches. Due to the fact that each star region can individually be expanded, it is possible to model all these different compositions for a state machine with a transition in one single pattern. This query can be seen as a validation that a transition has been inserted correctly in a statechart. Although the structure of state machines have changed remarkably in UML 2.0, a relatively similar pattern with a larger amount of elements are required for the UML 2.0 counterpart.

3 Matching Algorithm

In this section we will present a matching algorithm for patterns with isomorphic, star and negative regions.

We have discussed an intuitive interpretation of the query language where star regions are expanded into regular graphs. In practice, the actual patterns are not expanded prior to matching since an arbitrary number of possible patterns could be generated. Instead, the star regions are expanded while matching, and only as far as valid mappings against the target graph are found.

The algorithm presented below is used to match the pattern against the target graph and expand the star regions. To match individual regions, any traditional graph matching algorithm may be used; we have used an algorithm based on CSP [21] and VF2 [5, 6], as presented in [13].

The result of the matching algorithm is a set where each element is a mapping from the pattern graph to the target graph. In every such mapping, each node in an isomorphic region in the pattern is mapped exactly once, each node in a negative
Figure 4: (Top) An example of a pattern that illustrates how transitions are connected to state vertices and state machines. (Bottom) Two model fragments that matches the query.
region exactly 0 times and each node in a star region 0..n times. A node in the
target graph can be mapped only once in each mapping.

The algorithm as presented here is split into two functions—query and matchRe-
gion. Query initializes the matching by selecting which region to start from, in-
vokes the recursive matchRegion and lastly discards any results where negative
regions are successfully matched. Generally, the fewer mappings we find for the
first isomorphic region matched, the faster the algorithm will work. Therefore, we
generally start from the largest isomorphic region in the pattern as we are likely
to find relatively few mappings for that region.

1 query (pattern, target):
2   r = choose one isomorphic region in pattern
3   mappings = matchRegion (r, {}, target, {})
4 for each negative region in pattern:
5     c = a connection from a non-negative region to negative region
6     discard each mapping in mappings for which matchRegion (negative region, mapping, target, c) returns results
7 return mappings

The function matchRegion recursively traverses the regions in the pattern,
trying to expand the mappings found until all regions have been matched.
When this function is called, we either have the situation where no mapping has
been passed, or where one or more neighbors of the passed region have been
matched in the inputMapping. In the first case (lines 2–3), the function starts by
finding all valid mappings for the passed region, in the second case (lines 5–18),
it identifies a set of candidate mappings for the partial pattern consisting of all
previously matched regions and the passed region, i.e. a set of mappings where
the most recently matched connection of the passed region is satisfied (lines 5–
11). The matching is done recursively for star regions, implementing the pattern
generation described in Section 2.3 (lines 12-16).

The function then checks that all other connections to previously matched re-
gions are satisfied, thereby ensuring that the mapping is valid, i.e. that the topol-
ygy of the candidate mapping is consistent with that of the pattern (lines 17–18).
At this stage we have identified all valid mappings for the partial pattern matched
so far and continue with the next region in lines 19–20.

A note on connections: in this algorithm, we assume each connection consists
of two nodes in separate regions that are connected through an edge. A connection
is satisfied by a mapping where the two nodes are mapped to nodes in the target
graph that are likewise connected. There is an implicit connection between the
two ends of a star region which is dealt with on line 13 below. The connection
between the region to match and the previously matched region is passed on to
matchRegion as a parameter in order to identify a starting node for matching.
matchRegion (region, inputMapping, target, connection):
  if inputMapping is empty:
    Mappings = all valid mappings region → target
  else
    Mappings = {}
    starting node = the node in region connected through connection
    for each target node in these mappings:
      start with inputMapping plus a mapping starting node → target node
      from there, find all valid mappings region → target
      add these mappings to Mappings
    if region is a star region:
      c = connection to next instance of star region to be mapped
      for each Mapping in Mappings:
        replace Mapping with M = matchRegion (region, Mapping, target, c)
        add inputMapping to Mappings
    for each matched neighbor of region:
      discard all Mappings where a connection between region and neighbor is not satisfied
    for each connection c to a non-negative, unmatched neighbor of region:
      replace each Mapping in Mappings with M = matchRegion (neighbor, Mapping, target, c)
  return Mappings

4 Validation and Applications

We have built an experimental modeling tool called Coral [1]. In this tool we have implemented CQuery and a matching engine that supports the concepts we have discussed in this paper.

The main idea in the design of the CQuery language is that the base of the pattern is a model in the target language. The CQuery language consists of elements that extend a modeling language to include information to control a query. That is, the pattern consists of a model fragment in the target language, annotated with query configuration in the CQuery language. Since the CQuery language itself is separated from the modeling language of the target, the target language does not need to be modified to support CQuery.

We have chosen this approach for two reasons: First, there is no need to have a separate component that verifies that the patterns are possible to construct using the target metamodel, since adherence to the metamodel is implicit. Second, we believe that the creation of patterns is easier, since a significant part can be constructed as any other model in the target modeling language. However, this approach does not prevent the queries being presented in any particular syntax, including the concrete syntax of the target modeling language or a more general object diagram syntax. A discussion on using the concrete syntax of a modeling language in model transformation rules can be found in [3].

The CQuery metamodel is shown in Figure 5. The metamodel is rather small, containing only 3 metaclasses, where the Element can point to any abstract model element, and hence is not directly a part of the CQuery language. The base element is Pattern, which acts as the starting point of a query. Each Pattern consists of a set of Regions and an abstract container element which is an element in any
modeling language. This element owns all model elements in the pattern which are not annotations of CQuery. A Region is either an isomorphic, a negative or a star region. This is indicated by the corresponding flags. However, only one of these flags can be set for a particular region in a pattern. Region element in turn consists of an arbitrary number of QueryElement. The QueryElement contains information to control which attributes and outgoing edges should be ignored when matching a single element. In a well-formed pattern, all abstract elements have a corresponding QueryElement, and all QueryElements are owned by a Region.

The version of CQuery implemented in this tool is slightly different, but shares the same features that have been discussed in this paper. In our tool it is possible to create a query using the concrete syntax of the target modeling language. If the target language does not have a concrete syntax, it is still possible to create queries, but without the benefit of having diagrams. The matching engine in CQuery is based on the algorithm described in Section 3 and [13]. The algorithm is based on the VF2 and CSP algorithms and facilitates search planning and backtracking.

All star regions can optionally set an isMaximal flag. This flag can be used to indicate whether the matching engine should attempt to expand a star region a maximal number of times, instead of attempting to match an adjacent region to a subgraph that could actually be seen as a match to the star region. This feature can be very useful since an application that uses CQuery does not need to evaluate all possible matches if the point of interest is only the maximal possible matches of the star region. It must, however, be noted that although the isMaximal flag is set, this does not rule out the possibility that a star region could match a target graph where the star region had no occurrences.
4.1 Applications

The CQuery implementation is used by a variety of components and addins in the Coral tool. The most straightforward application of CQuery is a model search facility. In this component it is possible to load a set of query patterns defined in the Coral tool and search for occurrences of a pattern in open modeling projects. The results of the CQuery based search are reported as a set of mappings between elements in the query pattern and the target model.

We have also implemented a constraint evaluation component based on CQuery. This component is an integral part of the Coral tool, and uses a set of CQuery patterns to detect if modeling language constraints or well-formedness rules have been violated. This component is based on an approach where user models are continuously checked for errors. If an error is detected, the offending elements are reported along with an explanation, or a suggestion for correcting the problem. An example of how this constraint evaluation component has been used in a domain-specific language for System-on-Chip design called MICAS, can be found in [12].

Another application is a generic model to text transformation engine [15], which uses the CQuery language as the query facility. This application can e.g. be used for generating source code or documentation based on UML models.

Perhaps the most ambitious use of CQuery is a transformation engine based on the double pushout approach [19]. The transformation rules are given as a pair of a left-hand side (LHS) and a right-hand side (RHS), and an explicit mapping between the LHS and RHS. This transformation engine uses CQuery for matching the LHS to an occurrence in a model, and to specify the RHS. The transformation engine has support for negative, isomorphic regions and star regions, and provides in-place transformation of models. DPOTrans is extensively used in the Coral tool for defining the rules for editing models, e.g. inserting states or transitions in a statechart, or classes and associations in class diagrams. We have found that especially the star regions are necessary when defining model editing transformations in UML, where complex hierarchies of model elements occur frequently.

The Coral tool, including CQuery and all components mentioned in this section are open source and is available for download from http://mde.abo.fi/

5 Conclusions and Future Work

We have presented a query language for model-driven development applications that introduces the concept of star regions to represent hierarchical and repetitive structures. This query language has been implemented in a modeling tool and used successfully in different applications based on UML and other domain-specific modeling languages.

There are two clear future directions. First, introduce new regions operators, such as cardinality or disjunction. However, the need for these new operators
should arise from actual modeling tools. Also, we are studying the application
of our query language to model transformations. In fact, a model transformation
tool component based on CQuery has been already implemented and we plan to
present these results in the near future.

References

[1] Marcus Alanen and Ivan Porres. The Coral Modelling Framework. In Jo-
han Lilius Kai Koskimies, Ludwik Kuzniarz and Ivan Porres, editors, Pro-
ceedings of the 2nd Nordic Workshop on the Unified Modeling Language
NWUML 2004, number 35 in TUCS General Publications. TUCS Turku

[2] Carsten Amelunxen, Tobias Rötschke, and Andy Schürr. Graph Transforma-
tions with MOF 2.0. In Holger Giese and Albert Zündorf, editors, Fujaba

[3] Thomas Baar and Jon Whittle. On the Usage of Concrete Syntax in Model

[4] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transfor-
Kreowski, and G. Rozenberg, editors, Proc. Graph Transformation - First
International Conf., ICGT 2002, Barcelona, Spain, volume 2505 of LNCS.

for matching large graphs. In Proceedings of the 3rd IAPR-TC-15 Inter-
national Workshop on Graph-based Representations. Italy, pages 149–159,

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomor-

[7] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars
with negative application conditions. Fundamenta Informaticae, 26(3-

[8] Berthold Hoffmann. Graph transformation with variables. In Formal Meth-

[9] Berthold Hoffmann, Dirk Janssens, and Niels Van Eetvelde. Cloning and


