Abstract. Graph databases allow the efficient querying and storage of data as a property graph, which is used to represent the domain of many modern applications such as social analytics. Property graphs and the graph pattern matching queries typically used to query them, are imposing new challenges when it comes to follow a conceptual schema driven software design. Tools like the Unified Modeling Language (UML) in conjunction with the Object Constraint Language (OCL), are not still prepared for this type of queries effectively, thus risking the uptake of conceptual modeling practices when designing and implementing modern graph based applications. In this paper, we propose extensions to UML and OCL for the effective specification of graph based applications, thus preparing them for the advent of the graph based era.

1 Introduction

Graph database management systems (GDBMs) are a very representative type of NoSQL systems that have become very popular during the last decade [15, 16]. They implement the property graph data model, where data is modeled as a graph nodes connected by edges. Nodes are tagged with multiple labels representing different roles that the entities they represent in a domain, while edges can be also labeled to represent semantically rich relationships between entities. In the property graph model, both nodes and edges are treated as “first-class citizens” (data model elements), and can be enriched with properties.

Many modern applications such as search engines [7], fraud detection [13] or social analytics [14], can be naturally modeled using a property graph. In such applications, the underlying graph is typically queried by means of complex graph pattern matching queries aiming at finding the occurrences of subgraphs with a given structure (including constraints over the properties of the matched nodes) [17]. GDBMs are accompanied with graph query languages like Cypher (the query language of the Neo4j graph database [15]) or PGQL (the query language of PGX, the graph processing framework provided by Oracle [8]), specifically designed to express these queries effectively.
In their early development stages, applications using GDBMs like any other information system, are specified by building their conceptual schemas (CSs). CSs are the general knowledge an information system needs to know to perform its functions, and are composed by the structural schema, which describes the application data model and the behavioral schema, which describes the set of queries and actions that the system can perform. However, the property graph model and graph pattern matching queries, raise new challenges when it comes to create their specifications. The uptake of the conceptual schema centric development paradigm [9] is at risk if we do not provide the means to specify the type of queries implemented in such systems effectively. Tools like the Unified Modeling Language (UML) [6], in conjunction with the Object Constraint Language (OCL) [5], widely used to specify information systems, are not yet ready for such a promising yet challenging graph based era.

In this paper, we make a step forward in assuming that UML and OCL come closer to graph modeling, formalizing the required extensions for expressing graph pattern matching queries typically executed in GDBMs based applications. For such an endeavor, this paper makes the following contributions:

1. We extend UML with the new Graph and Path data types, which allows us to formalize the property graph model and graph pattern matching queries.
2. We propose a formal way to infer a property graph from the set of instances that conform a UML structural schema, a step required to properly reference property graphs from OCL.
3. We define in OCL the needed operations to express graph pattern matching queries.
4. We propose a concrete syntax for OCL and the corresponding modifications to the OCL abstract syntax tree, to express graph pattern matching queries in a less verbose way and, in general, more effectively.

The rest of this paper is organized as follows. In Section 2, we introduce the property graph model and graph pattern matching queries. In Section 3 we formalize the property graph model in UML. In Section 4 we provide a formal means to transform the instance of an existing UML structural schema into their equivalent property graph. In Section 5 we define in OCL the most common operations executed in GDBMs, graph pattern matching and path queries, and in Section 6 we propose a new OCL syntax for such queries with the corresponding OCL abstract syntax tree modification. Finally, in Section 7 we introduce some related work and in Section 8 we conclude the paper.

2 Preliminaries

The property graph model. In the literature, there is not an agreement on a formal definition for the property graph model. However, the most important and widely used GDBMs, define it as a set of nodes connected by edges, with the following characteristics [12]:
Both nodes and edges are first-class citizens with a unique id.

Nodes can be tagged with multiple labels.

Edges must be tagged with a single label.

Both nodes and edges can have properties in the form of key-value pairs.

Edges can be either directed or undirected. A directed edge connected two
nodes: the source which is the node the edge is leaving from, and the target,
which is the node the edge is pointing to. An undirected can be interpreted
as having two directed edges connecting the two nodes in opposite directions.

The property graph model is a schema-less data model. Some GDBMs allow
defining constraints over the required and unique property nodes and edges with
specific labels must have (e.g. a node with label Person must have a name
property set). However, GDBMs do not typically allow restricting the labels
of the nodes connected by edges with a given label. Finally, GDBMs support
multigraphs, i.e., two given nodes can be connected by multiple edges.

Figure 1 shows an example of a small property graph representing a simple
social network where persons follow other persons, and these are interested in
topics. Persons are represented as nodes with the label Person and contain the
properties name and age. Topics are represented as nodes with the label Topic
and contain the property name. Follow relationships are represented as an edge
with the label follows and have a property t which is the timestamp of the follow
relationship creation. Finally, the interests of persons in topics are represented
by means of edges with a label likes.

Graph Pattern Matching. Graph query languages allow expressing complex
graph pattern matching queries. A graph pattern matching query aims at re-
trieving the nodes and edges of a graph forming a specific structure that fulfills
a set of predicates over properties and labels of their nodes and edges. Graph
pattern matching queries are composed by the following elements:

Structural constraints used to specify the shape of the pattern, that is,
the number of nodes and edges, whether some parts of the pattern have a
variable length and the directions the edges can be navigated.

Node/Edge constraints used to specify the labels and property values the
nodes and edges of the pattern must contain.
Languages like Cypher and PGQL provide a syntax to express graph patterns textually in a very user friendly manner. In the rest of this paper, we will use the syntax of PGQL. Complex graph patterns can be expressed as a composition of linear patterns. A linear pattern is a pattern that connects a node, the tail, to another node, the head, through a sequence of other nodes. Tail and head have a degree of one while the rest of nodes of the pattern have a degree of two. Since linear patterns are the building blocks of more complex patterns (i.e. trees), in the rest of this paper we will focus on linear graph patterns.

Figure 2 shows two examples of graph pattern matching queries, with their pattern expressed both visually and using the PGQL syntax, and their expected results. The first query aims at finding people younger than 30, connected in either direction, to a person who likes “Soccer”. The structure pattern is expressed in PGQL by means of the following structural constraints:

\[(x:Person)-[y]-(Person)-[:likes]->(\text{WITH name = } "Soccer")\]

where () represents a node and -- represents an edge. The first edge can be navigated either from source to target or from target to source, while the second just from source to target. Navigation direction is specified by means of appending or prepending the > and < characters respectively.

The query contains four three node constraints and one edge constraint. The first node constraint, specifies that the tail must have the label Person and have a property age with a value smaller than 30. The second node constraint, specifies that the second node must have the label Person while the third node constraint specifies that the head must have a property name with value “Soccer”. Finally, there is an edge constraint on the second edge specifying that it must have the label likes. In PGQL, node constraints have the following form:

\[:\text{labels WITH predicates}\]

Note that matched subgraphs must not necessarily meet these requirements.
Table 1. Summary of the PGQL syntax.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>a node.</td>
</tr>
<tr>
<td>(:labels WITH predicates)</td>
<td>a node with labels and fulfilling the predicates.</td>
</tr>
<tr>
<td>{)--()</td>
<td>an edge that can be navigated in both ways.</td>
</tr>
<tr>
<td>{)--() , ( ) &lt;--()</td>
<td>an edge that can be navigated from source/target to target/source.</td>
</tr>
<tr>
<td>( )-[label WITH predicates]-()</td>
<td>an edge with label and fulfilling the predicates.</td>
</tr>
<tr>
<td>( )-{[ALL</td>
<td>ANY]}-()</td>
</tr>
<tr>
<td>( )-{[ALL</td>
<td>ANY] SHORTEST}-()</td>
</tr>
<tr>
<td>( )-{[ALL</td>
<td>ANY] SHORTEST USING operation}-()</td>
</tr>
<tr>
<td>( )-{[ALL</td>
<td>ANY] WITH length() &lt; X}-()</td>
</tr>
</tbody>
</table>

and are inserted within the parentheses ( ) in the case of nodes, and brackets [ ] in the case of edges. Finally, variables X and Y are used to reference the first node and the first edge of the path in order to query their elements.

The second query asks for all simple paths between a person named “John” and the topic “Soccer”. A path describes a finite sequence of edge navigations that bring us from a starting node, the tail, to an end node, the head of the path, through a sequence of intermediate nodes. A path is simple if the nodes that form the path are distinct (a node is not visited more than once). The length of the path is the amount of edges in the sequence.

Query two has the following structural constraints expressed in PGQL:

( )-{ALL}--( )

which specifies all simple paths between two nodes (that is, the number of nodes and edges between the tail and the head is undetermined, and must follow a simple path structure), by navigating the edges from source to target.

The query has two node constraints, specifying that the tail must have the label Person and a property name with value “John”, and the head node must have the label Topic and a property name with value “Soccer”. Finally, Table 1 summarizes the syntax of PQGL.

3 Modeling Property Graphs in UML

Figure 3 shows the formal definition of the property graph model in UML. A new data type named Graph is added to represent a property graph, which contains a set of nodes and a set of edges. A Node is a datatype with a set of labels, a unique id and a set of properties. A node has a set of outgoing edges, which are those edges the node is the edge’s source, and a set of ingoing edges, which are those edges the node is the edge’s target. On the other hand, an Edge, like a node, has a unique id and a set of properties, but a single label. An edge contains two nodes, the source and the target nodes the edge is connecting. Given that edges are the main entities in property graphs’ data model, and they might contain
multiple edges between a source and a target nodes, they are represented as a UML class instead of an association. Graphs are collections because these admit set operations such as the intersection, the union or the difference between over the sets of nodes and edges they contain.

Paths are represented as a subtype of graph, with an ordered set of edges, whose length is the number of the weight of its edges, whose default value is 1. The associative class WeightedEdgeOnGraph represents the weight of each edge in a graph, whereas the class WeightedEdgeOnPath represents an edge in a path, used navigated applying an specific direction: ingoing or outgoing. The head of an edge e in a path is the target node of e using navigating it with outgoing direction and the source, otherwise.

Although the UML diagram of Figure 3 represents all the related concepts of a property graph, there are some constraints related with the property graphs that can not be visually represented. Thus, we express them textually (OCL versions of these constraints can be found in Appendix A.1):

- Two nodes can not have a same id value.
- Two edges can not have a same id value.
- The source and target node of an edge of the graph, must also exist as nodes of edge’s graph.
- The direction of a WeightedEdgeOnPath only could be ingoing or outgoing.
- The head of a WeightedEdgeOnPath is the target of the edge of the direction is ingoing. Otherwise, the head is the source of the edge.
- The tail if a WeightedEdgeOnPath is the source of the edge of the direction is ingoing. Otherwise, the tail is the target of the edge.
- The head of each WeightedEdgeOnPath is the tail of the next WeightedEdgeOnPath.
– The length of a path is the sum of the weight of its edges.
– The head of a path is the head of the last edge.
– The tail of a path is the tail of the first edge.

4 From UML structural schemas to property graphs

The property graph is the data model used by graph query languages. Therefore, in order to support OCL graph operations based on the previously specified UML property graph, there must be a deterministic approach to infer the equivalent property graph of the Information Base (IB), which is the set of entities and relationships that conform with the structural schema in a given moment. In general, entity types are usually represented as UML classes; and relationship types as UML properties, associations or associative classes. In UML, the instances of entity type are called objects; and the instances of a relationship types are called, depending in how relationship types are specified, links for associations, linkobjects for associative classes or slots for attributes. Thus, for element of the information base, we define an equivalent representation in a property graph as follows:

1. For each object \( o_i \) instance of the UML classes \( \{C_1, \ldots, C_n\} \), there is an equivalent Node instance \( n_i \) in \( G \), whose labels are the names of \( \{C_1, \ldots, C_n\} \). The value of the properties attribute of \( n_i \) is a tuple with an entry for each property of \( o_i \). Given an object \( o_i \), we define \( p \) as a function such that \( p(o_i) = n_i \).
2. For each link \( l_i \) of a binary UML association \( A \), between two objects \( o_1 \) and \( o_2 \) as member ends \( m_1 \) and \( m_2 \) respectively, where the name of \( m_1 \) is < than the name of \( m_2 \), there is an equivalent Edge instance \( e_i \), whose label is \( A \); the source node is \( p(o_1) \); and the target node is \( p(o_2) \). For those links that belong to multiple binary UML associations, there will be an edge per association. Figure 4 shows an example of the resulting edge from a link.
3. For each link \( l_i \) of a n-ary UML association \( A \), between \( N > 2 \) objects \( \{o_1, \ldots, o_n\} \) as the member ends \( \{m_1, \ldots, m_n\} \), there is an equivalent Node instance \( n_i \), whose label is the name of \( A \). For each participant object \( o_i \in \{o_1, \ldots, o_n\} \), there is an Edge instance \( e_i \), whose source is \( n \), and whose destination is \( p(o_i) \). Like, the case of binary associations, if \( l \) belongs to multiple binary UML associations, there will be an edge per association.
4. For each link object \( l_0 \) on an UML associative class \( AC \), we apply the same procedure as for n-ary associations. The value of the properties attribute for \( p(l_0) \), there is a tuple with an entry for each slot of \( l_0 \).

For all the equivalent node and edges of an IB, we assume that the system assigns as object identifier their own \( id \), like GDBMS perform when users insert new data. However, any approach to ensure a unique value would be correct.

The contents of UML structural schemas are organized in packages. For this reason, we propose adding the following operation to the UML specification in the Package class, which returns the equivalent property graph for all the containing classes. In case of inner packages, the operation is applied recursively.

\( \text{Package::asGraph()}: \text{Graph} \)
5 Formalizing graph patterns in OCL

In GDBMs, the most common ways to implement the graph pattern matching problem is by following the rules behind subgraph isomorphism or subgraph homomorphism. The later is a generalization of the former which, in very simple words, accepts a node to appear multiple times in a matched subgraph if cycles or self loops appear in the graph, as it does not requires the existence of a bijective function between the graph and the pattern. For simplicity, we specify our methods to implement subgraph homomorphism.

Graph pattern matching queries return a set of tuples, where each tuple has a property per variable binding and sometimes, a hidden property for a matched path. For example, the following graph pattern returns a table with three columns: one for the $x$ variable, one for the $y$ variable and one for $p$, which contains the matched path.

\[ ((x : \text{Person}) \rightarrow [:\text{Follows ALL}] \rightarrow (y) \) AS p \]

These are the corresponding steps to create an equivalent OCL expression to specify a graph pattern:

- **Step 1:** Select the valid set of starting nodes according the node constraint of the tail of the pattern. These become the current tail nodes.
- **Step 2:** If there is a next structural constraint, then resolve the set of valid paths that leave from each current tail node and proceed to step 3. The structural constraint can be of fixed length (e.g. () -> ()), or a variable length (e.g. () <- {ANY} -> ()). Otherwise, create the corresponding tuple with all the bound variables that appear in the graph pattern and the final path, which is the concatenation of all the intermediate paths found so far.
- **Step 3:** If the structural constraint has an associated edge constraint, filter those paths that all their edges satisfy such constraint.
- **Step 4:** If there is a node constraint over the head of the path, evaluate it over each filtered path.
- **Step 5:** Return to step 2, but now the heads of the remaining paths become the current tail nodes.
As an example, the resulting equivalent OCL expression for the previous graph pattern would be as follows:

```ocl
let
    graph: Graph = SS.asGraph()
in
graph.nodes->select(n| n.label->includes("Person"))
  ->iterate( n, acc: Sequence<TupleType> |
              graph.allPaths(n, Direction::Outgoing)
              ->select(p| p.edges->forAll(e| e.label->includes("Follows"))
              ->iterate(p, result:Sequence<TupleType> = acc |
                          result->including(Tuple{ x: n, y: p.head, path: p })))
```

The rest of this section is organized as follows. Sections 5.1 and 5.2 give show how to specify in OCL Node/Edge constraints, and structural constraints respectively. Whereas Node and edge are expressed as simple boolean OCL expressions, structural constraints are graph operations returning a collection of paths given a tail node \( n \), a direction \( d \) and a weight function \( w \).

### 5.1 Node and Edge constraints

Node and edge constraints are predicates over the label or the properties of a node or an edge. Node and Edge constraints specifications in OCL is straightforward. As an example:

- To select those nodes or edges with an specific label \( \text{lbl} \).

  ```ocl
  label->includes(lbl)
  ```

- To select those nodes or edges with an specific value \( v \) for a property \( x \).

  ```ocl
  properties.x = v
  ```

### 5.2 Structural constraints

We classify structural constraints into fixed and variable length. Both types of constraints return a set of paths given a node \( n \), a direction \( d \), and a weight function \( w \). The function \( w \) is evaluated for each edge of the returning path.

**Fixed length.** Fixed length structural constraints ask for those paths of length one whose tail is \( n \), and its head is the node reached when navigating single edge using direction \( d \). We formalize the operation that computes such paths as an UML operation called \texttt{paths} as follows\(^4\):

```ocl
context Graph::paths(n:Node,d:EdgeDirection,w:(Edge, Graph)->Double):Set
body:
let outgoingEdges:Set(Edge)=edges->select(e|e.source=n) in
  if d = EdgeDirection::Outgoing then pathsByEdges(outgoing,direction,w)
else d = EdgeDirection::Ingoing then pathsByEdges(ingoing,direction,w)
else paths(n, EdgeDirection::Outgoing,w)
->union(paths(n,EdgeDirection::Ingoing,w))
```

\(^4\) The weight function is a \texttt{FuncType}, which is a suggested OCL extension in \[4\].
This operation uses the auxiliary function *pathsByEdges*, which constructs for a given set of edges, the corresponding paths using a direction \( d \) and weight function \( w \), specified as follows:

```oclm
context Graph
  def pathsByEdges:
    edges:Set(Edge), d:EdgeDirection, w:(Edge, Graph)->Double): Set(Path)=
    edges->iterate(e, acc:Set(Path)|
      p.oclIsNew() and p.oclIsTypeOf(Path) and
      acc->including(p)
      and p.edgeOnPath = Set{x} and x.oclIsNew() and x.oclIsTypeOf(
        WeightedEdgeOnPath)
      and x.weight = w(e, g) and x.edge = e and x.graph = p)
```

**Variable length.** Variable length structural constraints are specified by means of the modifiers \texttt{ALL} or \texttt{ANY}. These structural constraints do not restrict the number of edges that can appear in the returning paths, but these need to be simple (without repeated nodes). The \texttt{ALL} specifies that we are interested in all the simple paths from a given node (the tail) to any other node of the path (these will be filtered based on directions, edge constraints and/or other length modifiers such as shortest path).

We specify the \texttt{ALL} modifier by means of the \texttt{allPaths} operation, as follows:

```oclm
context Graph::allPaths(n:Node, d:EdgeDirection, w:FuncType(Edge, Graph)):
  Set(Path)=
  def:
    if d=EdgeDirection::Outgoing or d=EdgeDirection::Ingoing
      then allPaths(n, Set[], d, w)
      else allPaths(n, Set[], Direction::Ingoing, w)
            ->union(allPaths(n, Set[], Direction::Outgoing, w))
    endif
```

This operation calls a similar auxiliary function with an additional parameter to manage which nodes have already been visited (paths must be simple) and, in case of applying a navigation using a \texttt{both} direction, it executes the \texttt{allPaths} for each direction. This operation is specified in OCL as follows (all auxiliary functions are specified in appendices A.2 A.3 A.4):

```oclm
context Graph::allPaths(n:Node, visited:Set(Node), d:EdgeDirection, w:(Edge,
  Graph)->Double): Set(Path)=
  def:
    if x = null then Set()
    else
      neighbors(n, d)->excluding(visited)
      ->collect(y| edges(n, y, d)->flatten())->iterator(e, result:Set(Path) |
      let
        childrenPaths:Set(Path) = graph.allPaths(
          e.head(d),
          visited->union(nodes(e)->excluding(n)),
          direction)
      in
        childrenPaths->iterate(p, acc:Set(Path)=childrenPaths |
          acc->including(pathsByEdges(Set{e}, d, w)->any()))
    endif
```

This OCL code resolves all simple paths using the following strategy:
1. Obtains the neighbors of a node \( n \) (the tail) according to a specific direction \( d \). In other words, obtains the nodes that appear as source, or target according the specified direction \( d \).

2. Excludes from neighbors, those that have already been visited, which initially is an empty set.

3. For each non visited node \( y \), computes the edges from \( n \) to \( y \) with an specific direction \( d \).

4. For each computed edge \( e \), the allPaths operation is evaluated recursively with the heads of the computed edges using the same direction \( d \), and adding the already visited nodes to the collection.

5. For each of the returned paths \( p \), the edge \( e \) becomes the tail of those paths, which conforms the final result.

The ANY is similar to the ALL modifier but returns just one of the paths. Thus, it is implemented by calling the allPaths function, and picking just one path for each pair of nodes that appear as the tail and the head of the path. The corresponding OCL specification for the ANY modifier is as follows:

```ocl
class Graph {
  def anyPaths(node: Node, d: EdgeDirection): Set(Path) {
    let paths: Set(Path) = allPaths(node, d)
    in paths->iterate(it, acc:Set(Path) |
      if not acc->exists(p | p.tail=it.first() and p.head=it.last())
      then acc->including(it)
      endif)
  }
}
```

The ALL or ANY accept additional modifiers to ask for paths of a specific length. One of these modifiers is SHORTEST, which specifies that we are interested in the shortest simple paths out of all those found. The length of the paths can be computed using the weight, by means of the USING function modifier. We enrich the Graph type with the shortestPaths operation between two nodes according to an edge direction, as follows:

```ocl
class Graph {
  def shortestPaths(Node from, Node to, d: EdgeDirection): Set(Path) {
    body:
      let paths: Set(Path) = allPaths(from, d)
      in paths->select(p | p.head = to)
      ->iterate(it, acc:Set(Path) |
        if not paths->exists(p | p.weight < it.weight)
        then acc->including(it)
        endif)
  }
}
```

which can be used to compute the shortest path between a node and any other node of the graph.

Additionally, users can also ask for paths with a given length, also considering an operation to compute a weight if desired, using the modifier WITH length() operator \( x \). For example, the expression WITH length() operator \( x \), can be translated to OCL with a select operation over the resulting set from the allPaths or anyPaths operations in such a way:

```ocl
let weight = fun(e: Edge, g: Graph): Double = 1 end
in allPaths(n, d, weight)->select(p | p.length > 5 )
```
6 OCL concrete syntax for graph patterns

Even though graph patterns can be expressed in OCL with the constructs described in Section 5, we think the syntax is more verbose and less natural for those users used to write graph applications. For this reason, we propose extending OCL to support patterns using the PGQL syntax.

The concrete syntax of OCL allows modelers to write down OCL expressions in a standardized way. In this paper, we suggest to extend its syntax with an additional rule called $\text{GraphPatternExpCS}$ to express graph patterns like PGQL, as follows:

$$\text{GraphPatternExpCS} ::= \text{"Path{" NodeConstraintCS (StructuralConstraintCS NodeConstraintCS)* "\}" ("AS" VariableExpCS)?}$

$$\text{NodeConstraintCS} ::= \text{"("VariableExpCS? LabelConstraintCS* PropertiesConstraintCS")"}$

$$\text{LabelConstraintCS} ::= \text{:simpleNameCS}$

$$\text{PropertiesConstraintCS} ::= \text{"WITH" OCLExpression}$

$$\text{StructuralConstraintCS} ::= \text{("--"EdgeConstraintCS? ModifiersCS? ("--"|"->") | ("<-"|"->") EdgeConstraintCS)}$

$$\text{EdgeConstraintCS} ::= \text{"[" VariableExpCS? LabelConstraintCS? PropertiesConstraintCS? "]"}$

$$\text{ModifiersCS} ::= \text{("ALL"|"ANY") ("SHORTEST" (USING OCLExpression)? | OCLExpression)?}$

Now, after having defined all the specifications to resolve graphs queries and a concrete syntax add some syntax sugar to OCL, software engineers can express graph queries with OCL expressions like this:

```ocl
testObject.asGraph()->collect(Path{(x)-[Label('Follows ALL length = 5')]->(y)})
```

Figure 5 shows the UML specification of the OCL abstract syntax tree part corresponding to the proposed operation $\text{GraphPatternExpCS}$. This abstract syntax tree represents all the parts that conform a graph pattern expression represented by the $\text{GraphPatternExpr}$ class. Each part is a $\text{GraphConstraint}$ class, which admits a variable for the corresponding bindings of a graph pattern.

Since edge constraints can be viewed as being embedded inside structural constraints, we can simplify their representation without loosing semantics, assuming that structural constraints also can restrict the labels or the properties of
an edge. Additionally, structural constraints require direction used to navigate
and edge, and an optional modifier, to specify if the query requires to compute
all, or any of, the simple paths from a node, with or without the weight function.
Both properties are represented as attributes of the StructuralConstraint class.
In order to express constraints over the length property of a path, the StructuralConstraint has an optional property, whose type is an OCL expression for
that except to express the shortest path, which is represented with the boolean
attribute onlyShortest.

7 Related Work

Generally, the utility of graphs in a model-driven development has been dis-
cussed from three different perspectives. On one hand, to validate and test the
correctness of OCL specifications. In this case, there are some translations from
OCL to other graph grammars [10, 2, 1]. On the second hand, graphs have also
been used by model transformation languages and there are some approaches [3]
to translate their semantics into OCL to validate the correctness of the trans-
formations specifications. Finally, [18] proposes a model-driven approach based
on Entity-Relationship (ER) models to infer the best graph database design to
reduce the data accesses from queries.

However, although support for pattern matching queries is desired for future
versions of OCL [4], and exists a proposal for them, these do not support graph
patterns. Besides this, the PROGRES [11] language, which is very similar to
OCL to write software specifications, supports path expressions, which returns
the nodes reached at the end of the path, but never paths or graphs as an
existing datatype. This language, has important similarities with Cypher and
PGQL, and thus, with our proposed syntax to support graph patterns in OCL.

8 Conclusions

In this paper, we bridge property graphs and conceptual modeling by propos-
ing a set of mechanisms to specify systems based on executing graph pattern
matching queries easily. We have formalized the property graph model in UML
by extending the later with the new data types Graph and Path. Since graph
pattern matching queries require a property graph, we have proposed a mecha-
nism to automatically transform conceptual models into their property graph
equivalents. Then, we have shown how to express graph pattern matching queries
in OCL on top of the proposed property graph formalism. However, resulting
queries are verbose and unnatural from the perspective of a graph graph database
user, thus, we have proposed an extension to OCL to support the syntax of a
graph query language such as PGQL.

Our work is the first to grant OCL graph pattern matching capabilities effec-
tively, and aims at making OCL a good language for querying graphs. In order
to achieve this goal, our future work will consist of implementing other operators
typically found in graph and relational databases such as aggregates, group-bys
and sorts. A graph database implementing OCL with our proposed extensions, would be able to query graphs yet taking advantage of those features already present in UML and OCL.

References

A Appendix

A.1 Integrity constraints and derivation rules

```plaintext
context Node
inv: Node.allInstances()->isUnique(id)

context Edge
inv: Edge.allInstances()->isUnique(id)

context Edge
inv: source.graph->includes(graph)
and target.graph->includes(graph)

context WeightedEdgeOnPath:head:Node
derive: if direction = EdgeDirection::Ingoing then edge.target else edge
  .source endif

context WeightedEdgeOnPath:tail:Node
derive: if direction = EdgeDirection::Outgoing then edge.source else
  edge.target endif

context Path
inv: weightedEdgeOnPath->forAll(x,y| weightedEdgeOnPath->indexOf(y)=(weightedEdgeOnPath->
indexOf(x)+1) implies x.head = y.tail )

context Path
inv: edges->forAll(x| x.direction=EdgeDirection::Outgoing or x.direction=
  EdgeDirection::Ingoing)

context Path::length:int
derive: self.weightedEdgeOnPath->collect(weight)->sum()

context Path::head:Node
derive:
  if not edges->isEmpty() then
    edges->last().weightedEdgeOnPath.head
  endif

context Path::head:Node
derive:
  if not edges->isEmpty() then
    edges->first().weightedEdgeOnPath.tail
  endif
```

A.2 Neighbors

```plaintext
context Graph::neighbors(x:Node, d: EdgeDirection):Bag(Node)
body: if d = EdgeDirection::Ingoing then
  x.ingoing->intersection(graph.edges)->collect(source)
else if d = EdgeDirection::Outgoing then
  x.outgoing->intersection(graph.edges)->collect(target)
else
  neighbors(x, EdgeDirection::Ingoing)
  ->union(neighbors(x, EdgeDirection::Outgoing))
endif
```
A.3 Edges

```plaintext
context Graph::edges(x:Node, y:Node, EdgeDirection):Set(Edge)
body: if d = EdgeDirection::Ingoing
  then
    x.ingoing->intersection(edges)
    ->intersection(y.outgoing)
  else if d = EdgeDirection::Outgoing
    then
      x.outgoing->intersection(edges)
      ->intersection(y.ingoing)
  else
    edges(x, y, EdgeDirection::Ingoing)
    ->union(edges(x, y, EdgeDirection::Outgoing))
endif
```

A.4 Head

```plaintext
context Edge::head(d:Direction):Node
body: if d = EdgeDirection::Ingoing then target else source endif
```