The Diagramed Model Query Language 2.0: Design, Implementation, and Evaluation

Patrick Delfmann, Dennis M. Riehle, Steffen Höhenberger, Carl Corea, and Christoph Drodt

Abstract The Diagramed Model Query Language (DMQL) is a structural query language that operates on process models and related kinds of models, e.g., data models. In this chapter, we explain how DMQL works and report on DMQL's research process, which includes intermediate developments. The idea of a new model query language came from observations in industry projects, where it was necessary to deal with a variety of modeling languages, complex query requirements, and the need for pinpointing the query results. Thus, we developed the Generic Model Query Language (GMQL) tailored to deal with models of arbitrary modeling languages and queries that express model graph structures of any complexity. GMQL queries are formulas and professionals expressed the need to specify queries more conveniently. Therefore, the next development step was DMQL, which comes with functionality similar to GMQL, but allows to specify queries graphically. In this chapter, we describe both query languages, their syntax, semantics, implementation, and evaluation and come up with a new version of DMQL, which includes new functionality. Finally, we relate GMQL and DMQL to the Process Querying Framework.

1 Introduction

Manual analysis of process models has become unfeasible. Process models used in industry often contain thousands or even tens of thousands of elements [15, 19]. Process model querying has become established as a useful means of extracting relevant information out of process models, for instance, to support business process

P. Delfmann (✉) · D. M. Riehle · C. Corea · C. Drodt
University of Koblenz-Landau, Koblenz, Germany
e-mail: delfmann@uni-koblenz.de; riehle@uni-koblenz.de; ccorea@uni-koblenz.de; drodt@uni-koblenz.de

S. Höhenberger
University of Münster – ERCIS, Münster, Germany
e-mail: steffen.hoehenberger@ercis.uni-muenster.de

A. Polyvyanyy (ed.), *Process Querying Methods*,
https://doi.org/10.1007/978-3-030-92875-9_5

compliance management [13] or business process weakness detection [34]. Several approaches and tools have been developed to provide automatic or semi-automatic process model querying. Process *model* querying is applied at design time in advance to and independent of the actual process execution [4, 13]. Most process model query languages work in a similar way: one creates a formalized query that describes the model part to be searched. Once the search is initiated, an algorithm processes the search by analyzing the model for occurrences that adhere to the query (matches). As a result, the query returns either TRUE or FALSE to denote that the model matches the query or not, or it returns the entire set of matches depending on the query approach. Figure 1 shows a small stylized example. Here, we query the model for parts in which a document is edited after it is signed, which may be a possible compliance violation or process weakness.

For detecting the described model part, the query in Fig. 1 describes two activities that follow each other (not necessarily immediately, as indicated by the dashed arrow). Both activities have an assigned document. The activities are labeled with the respective verbs and wildcard characters to allow partial label matches. The two documents are equated to enforce the detection of activities dealing with the same document. After the search is finished, the detected match is highlighted, cf. Fig. 1.

Based on our experiences regarding requirements of model querying from industry projects and various studies we conducted on existing query languages [2, 8, 10], we identified a research gap which lead us to develop the Generic Model Query Language (GMQL). GMQL aims to provide a structural query language fulfilling the following requirements:

1. The query language should be applicable to any kind of graph-like conceptual model, regardless of its modeling language or view (e.g., data model, process model, or organizational chart). This means that it should be possible to formulate queries for different model types (note that this does **not** mean that the same GMQL query fits all kinds of models, but it is necessary to formulate an event-driven process chain (EPC [30]) query for EPC models, a Business Process Model and Notation (BPMN[1]) query for BPMN models, a Petri net [24] query for Petri nets, etc.).
2. It should not be necessary to transform a conceptual model into a special kind of representation (e.g., a state machine or the like) before a query can be executed.
3. It should be possible to formulate structural queries of any complexity.
4. The query language should consider attributes of any kind, i.e., attributes of model vertices (such as type, name, cost, etc.) and edges (such as type, label, transition probability, etc.).
5. Every match of a query in the regarded model or model collection should be returned/highlighted.

Thus, the query language should realize *configurable subgraph homeomorphism* working on both directed and undirected, vertex and edge attributed multigraphs,

[1] ISO/IEC 19510:2013, https://www.iso.org/standard/62652.html.

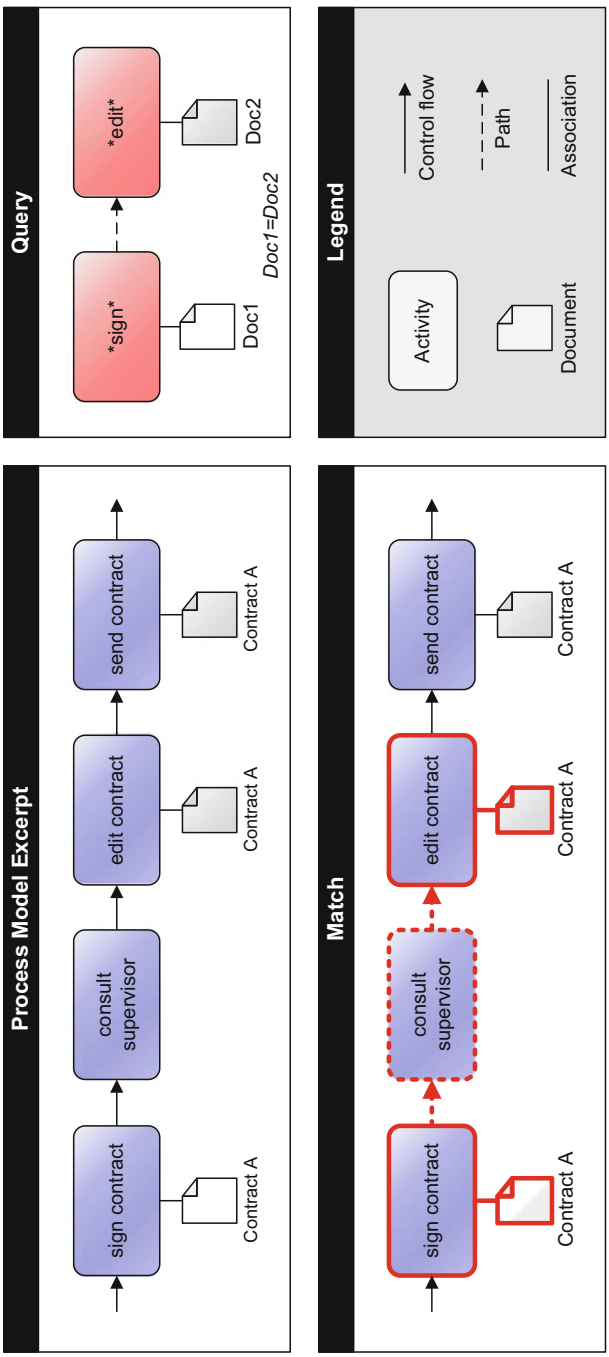

Fig. 1 Model analysis using queries

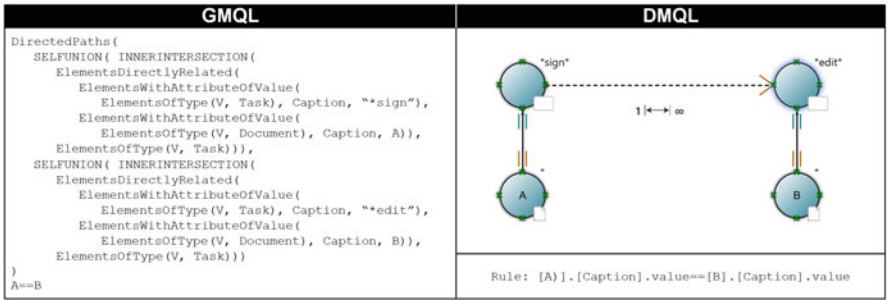

Fig. 2 Exemplary queries in GMQL and DMQL

where parts of the attributes of the multigraphs represent the syntax of the examined model's modeling language. Unlike pure subgraph homeomorphism [20], *configurable* means that it must be possible to adjust the kinds of returned subgraphs according to the attributes of the vertices and edges and the length and the vertex/edge contents of the paths that get mapped.

GMQL realizes the mentioned requirements and consists of the query language, a query editor, and a query algorithm (we will further refer to all these components as GMQL). It takes a query as input, which consists of a composition of set-altering functions working on the sets of vertices and edges of the model to be queried. The user defines (nested) functions, which denote specific sets of graph structures, e.g., elements of a specific type, elements with a specific label, or elements that are connected to other elements with specific characteristics. The sets that result from the functions can be combined (e.g., unified, joined, subtracted, or intersected) to assemble a query step by step.

We received feedback that GMQL is indeed helpful, but it is also difficult to use as the query specification is hierarchical and text-based, and people are usually not used to think in sets and formal notations [4]. Moreover, the users emphasized the need for graphical specification of queries.

Therefore, we developed the Diagramed Model Query Language (DMQL) [8]. DMQL meets the aforementioned requirements, but, in addition, it provides a graphical query editor and slightly extended analysis possibilities compared to GMQL. DMQL is not based on GMQL, nor is it just a new concrete syntax for GMQL. It comes with a different way of query specification, query formalization, and search algorithm (for details, cf. Sect. 3). To provide a first impression of GMQL and DMQL queries, the example query of Fig. 1 is formalized with GMQL and DMQL in Fig. 2.

The development of GMQL and DMQL followed the Design Science Research (DSR) methodology proposed by Peffers et al. [21]. First, we developed GMQL and implemented it as a plug-in for the meta-modeling tool [εm].[2] GMQL was

[2] https://em.uni-muenster.de.

evaluated by applying it in a business process compliance management project in the financial sector [4]. The mentioned issues (mainly the ease of use) resulted in the development of DMQL, which was also implemented in [εm] (partly demonstrated in [25]). DMQL was evaluated in two real-world scenarios, one dealing with compliance checking in the financial sector [18] and the other one dealing with business process weakness detection in several domains [9]. Based on the findings of these evaluations, in this chapter, we introduce DMQL 2.0, which provides enhanced functionality.

The remainder of this work is structured as follows: Sect. 2 introduces preliminaries including definitions of conceptual models, modeling languages, query occurrences, and matches, which are valid for both GMQL and DMQL. Section 3 explains GMQL and its abstract and concrete syntax and its semantics. Section 4 explains DMQL including its basic functionality (Sects. 4.1 to 4.4) and the extensions of DMQL 2.0 (Sect.4.5). Section 5 reports on the runtime complexity of GMQL's and DMQL's matching algorithms, their runtime performance measured empirically, and a utility evaluation. In Sect. 6, we position GMQL and DMQL within the Process Query Framework [23]. Section 7 closes this work with a conclusion.

2 Preliminaries

GMQL and DMQL allow to query models regardless of the respective modeling language, as both query languages are based on a meta-perspective of conceptual models [10]. In essence, any model is seen as an attributed graph, which is represented by its objects (vertices) and relationships (edges). The semantics of the model's objects and relationships is defined through attributes in the graph, e.g., by specifying the type or other properties of an object or relationship. The syntax is given through constraints on the graph that prescribe which kinds of vertices and edges can be connected to each other. Given the attributed graph, it is possible to formalize it (respectively, the original model) in a set-theoretic manner. Accordingly, GMQL and DMQL are based on a generic meta-model of conceptual models, shown in Fig. 3. Next, we provide a definition of a conceptual model based on the meta-model.

Definition 2.1 (Conceptual Model) A *conceptual model* is a tuple $M = (V, E, T_v, T_e, C, \alpha, \beta, \gamma)$, where V is a set of vertices and E is a set of edges between the vertices. Here, E is defined as $E = E_D \cup E_U$, where $E_D \subseteq V \times V \times \mathbb{N}$ is the set of directed edges, and $E_U = \{\{v_1, v_2, n\} \mid v_1, v_2 \in V, n \in \mathbb{N}\}$ is the set of undirected edges. We denote $Z = V \cup E$ as the set of all vertices and edges. $T_v(T_e)$ are sets of vertex (edge) types such as "activity", "event", "control flow", "data input", and the like. C is a set of captions, i.e., labels or other attribute values. For assigning vertices (edges) to vertex (edge) types or vertices and edges to their values, we use the functions $\alpha : V \mapsto T_v$, $\beta : E \mapsto T_e$, and $\gamma : Z \mapsto C$, where α

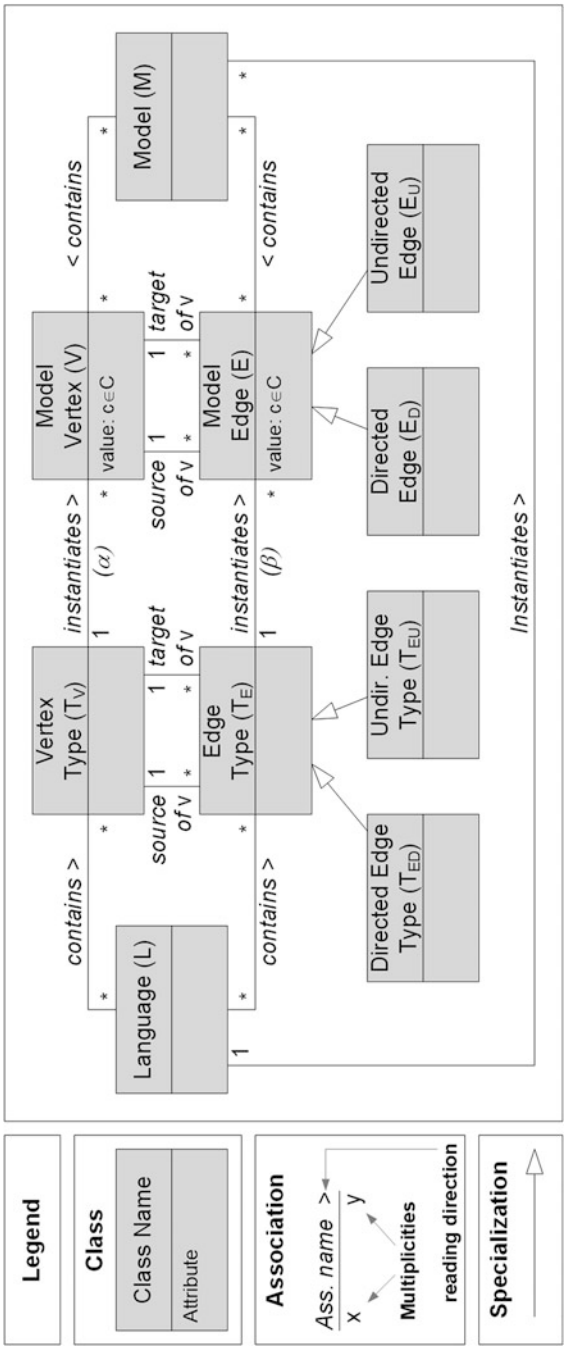

Fig. 3 Meta-model of conceptual models, adapted from [8, p. 479]

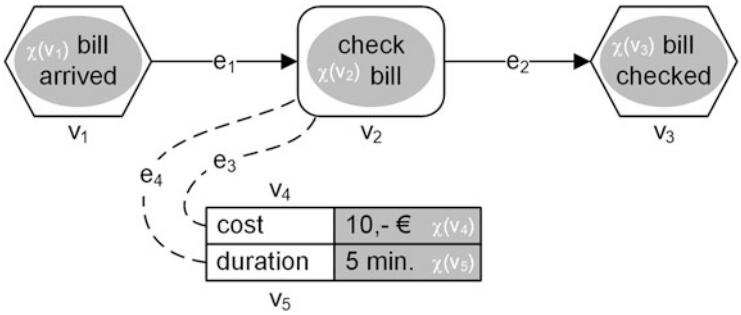

Fig. 4 Exemplary conceptual model

assigns a node in V to its type in T_v, β assigns an edge in E to its type in T_e, and γ assigns an element in Z to a caption in C.

The reason why we use 3-tuples to define edges is that we need to be able to define multiple, distinguishable edges between the same vertices in multigraphs. This means that for each new edge between the same vertices, we create a new 3-tuple with an increased index $n \in \mathbb{N}$. Note that we explicitly refrain from the statement $v_1 \neq v_2$ in the definition of undirected edges to allow undirected self-loops.

Vertices and edges can both have a value (i.e., their caption). It depends on the modeling language if vertices should have additional attributes (such as *duration*, *cost*, *description*, or the like). In such cases, we define additional attributes in the same way as vertices. That is, such additional attributes *are* vertices. However, they are visualized differently from common vertices. While common vertices are usually visualized as shapes, attributes are not. They can rather be accessed by opening a context menu. Attributes are assigned to vertices via undirected edges. Attributes also have values (e.g., "30 min" for the attribute *duration* of an *activity* vertex).

Figure 4 shows an exemplary process model in EPC notation. It can be formalized via $V = \{v_1, v_2, v_3, v_4, v_5\}$, $E = \{e_1, e_2, e_3, e_4\}$, $e_1 = (v_1, v_2, 1)$, $e_2 = (v_2, v_3, 1)$, $e_3 = \{v_2, v_4, 1\}$, $e_4 = \{v_2, v_5, 1\}$, $T_v = \{$*function*, *event*, *cost*, *duration*$\}$, $T_e = \{$*control-flow_ef*, *control-flow_fe*, *function-cost*, *function-duration*$\}$, $C = \{$"bill arrived", "check bill", "bill checked", "10,- €", "5 min"$\}$, $\alpha(v_1) = \alpha(v_3) =$ *event*, $\alpha(v_2) =$ *function*, $\alpha(v_4) =$ *cost*, $\alpha(v_5) =$ *duration*, $\beta(e_1) =$ *control-flow_ef*, $\beta(e_2) =$ *control-flow_fe*, $\beta(e_3) =$ *function-cost*, $\beta(e_4) =$ *function-duration*, $\gamma(v_1) =$"bill arrived", $\gamma(v_2) =$"check bill", $\gamma(v_3) =$"bill checked", $\gamma(v_4) =$"10,- €", and $\gamma(v_5) =$"5 min".

Definition 2.2 (Modeling Language) A *modeling language* is a pair $L = (T_V, T_E)$. T_V describes a set of vertex types. $T_E = T_{ED} \cup T_{EU}$ is a set of edge types, where $T_{ED} \subseteq T_V \times T_V \times \mathbb{N}$ is a set of directed edge types and $T_{EU} \subseteq \{\{t_{vx}, t_{vy}, n\} | t_{vx}, t_{vy} \in T_V, n \in \mathbb{N}\}$ is a set of undirected edge types (cf. [8]).

The definition above makes it possible to allow different edge types between the same vertex types, as they are present in several modeling languages. For instance, in

EPCs, it is common to use both the edge type *input* and the edge type *output* between functions and data objects. As another example, class diagrams allow four edge types, *composition*, *association*, *generalization*, and *aggregation* between classes. Note that we explicitly refrain from the statement $t_{vx} \neq t_{vy}$ in the definition of undirected edge types to allow undirected self-loops.

Definition 2.3 (Query Matches) A query Q that is applied to a model M returns a set of query occurrences, which are subsections of the queried model graph. The details of the formal definitions of GMQL and DMQL queries can be found in the dedicated subsections below.

Given a conceptual model M, a *query occurrence* of a query Q in M is a set $Z' \subseteq Z$, where Z is the set of all vertices and edges (see Definition 1).

GMQL and DMQL can be used to query a conceptual model M and as a result present the user a set of all query occurrences in M. We call such a set of query occurrences *query matches*.

Let $\mathfrak{M}$ be the set of all conceptual models, $\mathfrak{Q}$ the set of all queries, and $\mathfrak{P}_M$ the set of all possible query occurrences. Then, the query matches are defined as a function $Match: \mathfrak{M} \times \mathfrak{Q} \rightarrow \mathscr{P}(\mathfrak{P}_M)$, which maps a specific model M to the set of all query occurrences of $\mathfrak{Q}$ contained in the model.

In other words, $Match(M, Q)$ returns a set of subgraphs of M, which correspond to query Q. Detailed formalizations of how matches are identified are given in Sects. 3 and 4.

3 The Generic Model Query Language (GMQL)

GMQL provides set-altering functions and operators, which exploit the set-based representation of a model to execute queries. GMQL queries return result sets, i.e., parts of models, which satisfy the conditions of the queries [10]. A GMQL query can be composed of different nested functions and operators to produce arbitrarily complex queries.

GMQL can be used to find model subgraphs, which are partly isomorphic and partly homeomorphic with a predefined query. The query execution returns every query occurrence and thus allows to present the user all sections of the model that satisfy the query. In the following, we introduce the syntax and semantics of GMQL.

3.1 Syntax

A GMQL query consists of functions and operators, cf. [10]. A GMQL function has an identifier and at least one input parameter. For instance, regard the GMQL function

$$ElementsOfType(X, t). \tag{1}$$

The identifier of this function is "ElementsOfType". This indicates that the function can be used to return model elements of a certain type. The function takes two parameters. The first parameter is a set $X \subseteq Z$ (i.e., any set of model elements). This set is the search space. It provides the information that is meant to be queried by the function. In the statement of listing (1), the user could, for instance, provide the set Z of all elements of a model as an input to the function. This would allow to search the entire model. The second parameter is a parameter expression. In GMQL, parameter expressions are inputs to functions which are not sets, that is, single values. Depending on the specific functions, these could be integers, strings, values of variables, or in the case of listing (1), an element type the elements of Z should be of (i.e., a value of an enumerated type). For the example in Fig. 4, the query *ElementsOfType*(Z, "*Event*") would return the query occurrences $\{v_1\}$ and $\{v_3\}$.

Other than providing sets as inputs to functions, it is also possible to nest functions and operators. To this end, a function can, for instance, be provided as the parameter X, e.g.,

$$\mathit{ElementsOfType}(\mathit{GMQLFunction}(\mathit{Parameters}), t). \tag{2}$$

Here, an initial search space is modified via the inner *GMQLFunction*, with its respective parameters, and this modified search space is then used as a basis for the introduced *ElementsOfType* function. Nesting functions and operators allows to define complex queries tailored by the user. Next, we define the syntax of GMQL [10] using the Extended Backus-Naur Form (EBNF).[3] A GMQL query Q is defined as follows:

```
Q = subQueryExpression {"," equationExpression};

subQueryExpression = functionExpression |
                     operatorExpression |
                     setExpression;
```

Each query consists of a `subQueryExpression` that carries the actual query and, optionally, one or more equation expression(s) used to compare variables from the query. A `subQueryExpression` is either a `functionExpression`, an `operatorExpression`, or a `setExpression`.

A `setExpression` is the simplest input of a query and represents the basic input set V, E, or Z.

```
setExpression = "V" | "E" | "Z";
```

[3] ISO/IEC 14977:1996, https://www.iso.org/standard/26153.html.

A functionExpression consists of the function's identifier (see the list of possible function identifiers below), followed by an opening bracket and one or more subQueryExpressions or parameterExpressions. Using the subQueryExpression as a function parameter makes it possible to nest queries.

```
functionExpression = functionIdentifier "(" subQueryExpression
["," (parameterExpression | subQueryExpression)] ["," 
  (parameterExpression | subQueryExpression)]")";

functionIdentifier = "ElementsOfType" |
    "ElementsWithAttributeOfValue" |
    "ElementsWithAttributeOfDatatype" |
    "ElementsWithRelations" |
    "ElementsWithSuccRelations" |
    "ElementsWithPredRelations" |
    "ElementsWithRelationsOfType" |
    "ElementsWithSuccRelationsOfType" |
    "ElementsWithPredRelationsOfType" |
    "ElementsWithNumberOfRelations" |
    "ElementsWithNumberOfSuccRelations" |
    "ElementsWithNumberOfPredRelations" |
    "ElementsWithNumberOfRelationsOfType" |
    "ElementsWithNumberOfSuccRelationsOfType" |
    "ElementsWithNumberOfPredRelationsOfType" |
    "ElementsDirecltyRelated" | "AdjacentSuccessors" |
    "Paths" | "DirectedPaths" | "Loops" |
    "DirectedLoops" | "PathsContainingElements" |
    "DirectedPathsContainingElements" |
    "PathsNotContainingElements" |
    "DirectedPathsNotContainingElements" |
    "LoopsContainingElements" |
    "DirectedLoopsContainingElements" |
    "LoopsNotContainingElements" |
    "DirectedLoopsNotContainingElements";
```

A parameterExpression is used to input a single value into a function, such as vertex or edge types. It is either an Integer allowing arbitrary numbers, an AttributeValue allowing arbitrary strings, an ElementType allowing one of the element types in $(T_E \cup T_V)$, a variable used to make comparisons in equationEx- pressions, or an AttributeDataType.

```
parameterExpression = Integer | ElementType |
  AttributeDataType | AttributeValue | Variable;

AttributeDataType = "INTEGER" | "STRING" | "BOOLEAN" |
  "ENUM" | "DOUBLE";
```

An `operatorExpression` is either a `unaryOperatorExpression` or a `binaryOperatorExpression`. Both start with an identifier, either a `unaryOperatorIdentifier` or a `binaryOperatorIdentifier`, followed by an opening bracket, one or two parameters, and a closing bracket. The possible operator identifiers can be taken from the list below. `SELFUNION` and `SELFINTERSECTION` take one parameter, and all the others take two.

```
operatorExpression = unaryOperatorExpression |
  binaryOperatorExpression;

unaryOperatorExpression = unaryOperatorIdentifier
  "(" subQueryExpression ")";
unaryOperatorIdentifier = "SELFUNION" | "SELFINTERSECTION";

binaryOperatorExpression = binaryOperatorIdentifier
  "(" subQueryExpression "," subQueryExpression ")";
binaryOperatorIdentifier = "UNION" | "INTERSECTION" |
  "COMPLEMENT" | "JOIN" | "INNERINTERSECTION" |
  "INNERCOMPLEMENT";
```

Finally, an `equationExpression` is used to compare values of variables that have been used in a query.

```
equationExpression = Variable ("=" | "!=" | "<" | ">" |
  "<=" | ">=") Variable;
```

The values of the variables are calculated at runtime. For instance, the query

$$\mathit{DirectedPaths}(\mathit{ElementsOfType}(V, A), \mathit{ElementsOfType}(V, B)),$$
$$A = B,$$

returns all directed paths that start and end with a vertex of the same type (recall that V is the set of all vertices of the input model). If the query was applied to an EPC, for instance, it would return all directed paths from function to function, from event to event, from XOR connector to XOR connector, etc. Each path that matches the query is returned as one result set. This means that the overall result of the query can consist of several sets. Each such set contains the start and end vertex of the path as well as all vertices and edges *on* the path.

3.2 Semantics, Notation, and Query Example

A GMQL query is performed by applying it to the set-based formalization of a model in order to retrieve the respective matches. GMQL provides an extensive set of functions and operators. Functions allow to query for element types, element values, relations to other elements, paths, and loops. Operators, such as `UNION`, `INTERSECTION`, `COMPLEMENT`, and `JOIN`, allow to combine functions to create arbitrarily complex queries. Due to space limitations, we omit a full specification of functions' and operators' semantics. Please refer to [10] for details.

To provide an example, we revisit the exemplary function *ElementsOfType*(X, t). As discussed, this function returns a subset of elements from X which are of type t. We recall that Z is the set of all elements of a conceptual model M, $X \subseteq Z$, and α and β are functions that assign vertices and edges to their types. The semantics of *ElementsOfType* is then defined by $\mathit{ElementsOfType}(X, t) = \{x \in X | \alpha(x) = t \vee \beta(x) = t\}$.

When computing matches, query occurrences are determined through an application of the respective function or operator semantics. In case of nested functions, the output of inner functions is passed as input to outer functions, and the semantics is evaluated accordingly. The overall query result builds up incrementally, allowing to traverse the resulting query tree in post order.

To provide an example, we consider a scenario from business process weakness detection. In business process models, certain constructs may be syntactically correct but represent a shortcoming (i.e., represent an inefficient or illegitimate part of the process) and should hence be avoided. To detect such parts, we can apply a corresponding GMQL query. The query returns all occurrences of the weakness, and the user can decide how the weak sections of the process model shall be improved. An example of such a weakness pattern is a document that is printed and scanned later on. The user may be interested in all parts of the process model where such a "print–scan" pair of activities exists.

The corresponding GMQL query would require several functions and operators, the semantics of which we introduce exemplarily. To express that we are searching for execution paths, we use the function *DirectedPaths*(X_1, X_2), which takes two sets of elements, where the first set represents possible starting points of the paths and the second one represents possible end points of the paths. *DirectedPaths* returns all directed paths that start in an element of X_1 and end in an element of X_2, where each path is captured as a set of elements (i.e., both the vertices and edges of the path). To express that the process model's activities are annotated with documents, we make use of the GMQL function *AdjacentSuccessors*(X_1, X_2). It returns all pairs of elements (one from X_1 and the other from X_2) that are connected by a directed edge, where the source of the edge is an element from X_1 and the target of the edge is an element from X_2. To access the contents (i.e., the labels) of model vertices, we use the function *ElementsWithAttributeOfValue*(X, t, u) that

takes a set of elements X, an attribute type t, and a value u as input and returns all elements from X that carry an attribute of type t, which in turn carries the value u.

```
 DirectedPaths(
  SelfUnion(InnerIntersection(
    ElementsOfType(V,activity),
      AdjacentSuccessors(
        ElementsWithAttributeOfValue(
          ElementsOfType(V,activity),caption,"*print*"),
        ElementsWithAttributeOfValue(
          ElementsOfType(V,document),caption,A)))
  SelfUnion(InnerIntersection(
    ElementsOfType(V,activity),
      AdjacentSuccessors(
        ElementsWithAttributeOfValue(
          ElementsOfType(V,document),caption,B),
        ElementsWithAttributeOfValue(
          ElementsOfType(V,activity),caption,"*scan*"))))
A=B
```

Consider the query shown above. The most inner function *ElementsWithAttributeOfValue*(*ElementsOfType*(V, *activity*), *caption*, "$*$ *print* $*$ ") returns all activity vertices that carry "print" in their names. The other inner function *ElementsWithAttributeOfValue*(*ElementsOfType*(V, *document*), *caption*, A) returns all document vertices that carry a specific name not known at this stage, however represented through the variable A. The next outer function *AdjacentSuccessors* takes the results of the two inner functions as input and returns all model sections where an activity vertex with "print" in its name is connected to a document vertex via a directed edge pointing to the document vertex. The result of the other function *AdjacentSuccessors* (sixth last row of the listing) is calculated analogously. As the results are model sections, *AdjacentSuccessors* consequently returns a set of sets as a result. To find paths that reach from a "print" activity vertex having a document as output to a "scan" activity vertex having a document as input, we search for paths from the former to the latter. *DirectedPaths* takes sets as input and not sets of sets. Thus, we need to extract the activity vertices out of the intermediate result sets before we use them as inputs for the *DirectedPaths* function. We do this through the use of operator *INNERINTERSECTION*, which performs a set intersection of *ElementsOfType*(V, *activity*) with the inner sets of the results of the *AdjacentSuccessor* function. What remains is the activity vertices we searched for, still in sets of sets. Hence, the last step is to transform the sets of sets into single sets, which is done via *SELFUNION*. The outer function *DirectedPaths* now takes the single sets as input that only carry the activity vertices having documents annotated. The final result is a set of paths, each from an activity with a document annotated to another activity with a document annotated. Each path is encoded as a set of vertices and edges that lie on the path. To assure that we only find sections of a

process where the same document is printed and scanned subsequently, we use the variables *A* and *B* for the names of the documents and require that their values are equal (cf. the last line of the listing).

The rest of the functions and operators of GMQL work similarly to those above and allow to search for different kinds of structures and to assemble the sub-queries in novel ways. To conclude, we can construct arbitrarily complex queries that address the granularity of elements, the relationships between elements, constraints regarding paths and loops, or specific multiplicities of certain elements or relations, that is, counting. The queries can then be used to search conceptual models for query occurrences and present these to the user.

Results of GMQL queries are highlighted in the model that is currently examined by surrounding the vertices and edges involved in a query occurrence with colored bold lines. If there is more than one query occurrence, the user can browse through the results [10]. The GMQL function names are given in a comprehensible manner to increase usability for unexperienced users. For further details on the semantics of individual functions and operators, refer to [10].

3.3 The Transition from GMQL to DMQL

While GMQL is a powerful language for querying arbitrary conceptual models in the context of the requirements identified in [8], the queries must be formulated using textual formalisms. A utility evaluation of GMQL [4] has shown that users appreciate graphical specification of queries, as formal textual statements are regarded complicated with low ease of use. Hence, we developed a graphical concrete syntax for GMQL called vGMQL [32]. However, we experienced that assembling queries through defining, reducing, nesting, intersecting, unifying, and joining sets of model elements as it is required by the abstract syntax of GMQL merely with graphical symbols does not increase its ease of use. In other words, we cannot "draw" a query in a way such that it looks like the kinds of model sections we are searching for. Therefore, we created DMQL [8]. This model query language is built on the same formal foundation as GMQL (see Sect. 2) and supports graphical specification of queries. Particularly, the concrete syntax of DMQL is shaped in a way such that a query looks much like the kinds of model sections that should be searched for. Consequently, DMQL is not just a new concrete syntax for GMQL (like vGMQL is), but a completely new language with an own way of specifying queries, an own abstract and concrete syntax, and own semantics. The following section introduces DMQL and discusses its relation to GMQL.

4 The Diagramed Model Query Language (DMQL)

While GMQL allows the user to define textual queries, DQML focuses on a visual syntax [8–10]. This helps users to formalize queries in a more user-friendly way. DMQL's matching algorithm, unlike the one of GMQL, is based on an

adapted graph matching algorithm known from algorithmic graph theory [8, 11]. In particular, the query algorithm extends subgraph homeomorphism [20] working on both directed and undirected, vertex and edge attributed multigraphs, where parts of the attributes of the multigraphs encode the syntax of the examined models. Thus, DMQL does not build up a query through set-altering functions and operators, like GMQL does, but uses a visual query graph as input. DMQL then searches for extended-homeomorphic occurrences of the input graph in the queried models [8]. In order to make the visual graph suitable as input for the algorithm, it is transformed into a formal representation. Like GMQL, DMQL is capable of processing any conceptual model, and therefore it is based on the same meta-model as GMQL (see Fig. 3). Thus, the definition of the conceptual model, the modeling language, and the query occurrence and query match is identical for both query languages and is listed in Sect. 2, Definitions 1 to 3.

DMQL allows to query models of multiple modeling languages. This means that we can define queries whose occurrences might span models of different modeling languages. For instance, a query expressing the so-called *four-eye-principle* requires that a document is checked twice in a process, where the second person who checks is the supervisor of the person who makes the first check. The information about the order of the activities can be taken from the process model; however, the information if the second person is a supervisor of the first one can only be taken from an organizational chart. Thus, a query that checks if the *four-eye-principle* is realized in a business process has to work on multiple modeling languages.

4.1 Syntax

A DMQL query is a tuple. It is defined as $Q = (V_Q, E_Q, P_V, P_E, \delta, \varepsilon, \Gamma, G)$. In DMQL, a query is a directed multigraph consisting of vertices V_Q and edges $E_Q \subseteq V_Q \times V_Q \times \mathbb{N}$. Each vertex v_Q in V_Q and each edge e_Q in E_Q can be assigned properties, which define how the query vertices and edges should be mapped to model vertices and edges. The properties are assigned by two functions: $\delta : V_Q \rightarrow P_V$ and $\varepsilon : E_Q \rightarrow P_E$. P_V and P_E are sets of tuples and contain vertex and edge properties, where $P_V \subseteq VID \times VCAPTION \times VTYPES$ and $P_E \subseteq EID \times ECAPTION \times DIR \times MINL \times MAXL \times MINVO \times MAXVO \times MINEO \times MAXEO \times VTYPESR \times VTYPESF \times ETYPESR \times ETYPESF \times \Theta$. Γ is the set of modeling languages the query is applicable to, and G is a set of global rules (see explanation at the end of this section).

A property of a query vertex is a tuple and consists of the following components: *VID* is a set of query vertex IDs, and $vid \in VID$ is a string that represents the ID of one query vertex. IDs are not immediately used for matching but to build rules within a query (similar to the equation expression in GMQL, see section on global rules below). *VCAPTION* is the set of all vertex matching phrases. Correspondingly, $vcaption \in VCAPTION$ defines the caption that a vertex in a model should have to be matched. *vcaption* is a string and can contain wildcards. $VTYPES = \mathscr{P}(T_V)$ is

the set of all possible sets of vertex types. Consequently, *vtypes* $\in$ *VTYPES* is a set of vertex types. Each vertex in a model that has one of these types is a matching candidate.

A property of a query edge is a tuple and consists of the following components: *EID* is a set of query edge IDs that are used analogously to vertex IDs. Correspondingly, *eid* $\in$ *EID* is a string that represents the ID of one query edge. *ECAPTION* is the set of all edge matching phrases. *ecaption* $\in$ *ECAPTION* defines the caption that an edge in a model should have to be matched. It is a string and can contain wildcards. $DIR = \mathscr{P}(\{org, opp, none\})$ is the set of all possible combinations of edge directions. Therefore, $dir \in DIR$ is a subset of $\{org, opp, none\}$ (i.e., $dir \subseteq \{org, opp, none\}$) and defines which direction a model edge should have in order to be mapped to the query edge. If a model edge to be mapped must have the same direction as the original query edge, *org* needs to be chosen. If we choose *opp*, then the model edge must have the opposite direction of that in the query. Lastly, *none* means that the model edge to be mapped must be undirected. We can combine these three options in an arbitrary way, for instance, if we choose *org* and *opp*, we allow the mapped edges to have any direction.

MINL, *MAXL*, *MINVO*, *MAXVO*, *MINEO*, and *MAXEO* are sets of natural numbers including -1, i.e., $MINL = MAXL = MINVO = MAXVO = MINEO = MAXEO = \mathbb{N}_0 \cup \{-1\}$. $minl \in MINL$ and $maxl \in MAXL$, ($minl \leq maxl$) define if the query edge should be mapped to a single edge or a path in the model. If both are equal to 1, then the query edge is mapped to a single edge in the model. If $maxl > minl$, then the query edge is mapped to a path with at least *minl* and at most *maxl* edges. If $maxl = -1$, then the paths to be mapped can have any maximum length. The properties explained next are only evaluated if $maxl \geq 2$.

$minvo \in MINVO$, $maxvo \in MAXVO$, $mineo \in MINEO$, and $maxeo \in MAXEO$ define if a path that gets mapped to a query edge is allowed to cross itself, that is, run multiple times over the same model vertices and/or edges (so-called vertex and edge overlaps). For instance, if $minvo = 0 \wedge maxvo = 1$, then the path to be mapped is allowed to cross itself once in one vertex, that is, one vertex on the path is allowed to be visited twice (but does not have to because $minvo = 0$). Edge overlaps are controlled in a similar way by *mineo* and *maxeo*. If $maxvo = -1$ ($maxeo = -1$), then the matching algorithm allows any number of vertex (edge) visits.

$vtypesr \in VTYPESR$ (with $VTYPESR = \mathscr{P}(T_V)$) is a set of vertex types and requires that for each vertex type contained in *vtypesr* there is at least one vertex belonging to that type on the path. $vtypesf \in VTYPESF$ (with $VTYPESF = \mathscr{P}(T_V)$) is also a set of vertex types and requires that *none* of the vertices on the path belongs to any of the vertex types contained in *vtypesr*.

$etypesr \in ETYPESR$ (with $ETYPESR = \mathscr{P}(T_E)$) and $etypesf \in ETYPESF$ (with $ETYPESF = \mathscr{P}(T_E)$) work analogously to *vtypesr* and *vtypesf*, however with edge types. Element types cannot be both required and forbidden on paths, so $etypesr \cap etypesf = \emptyset$ and $vtypesr \cap vtypesf = \emptyset$. All properties that relate to paths are ignored as soon as $maxl = 1$. An exception is *etypesr*, which relates to a single edge in case $maxl = 1$. Then, a single matched model edge must have one of the types in *etypesr*.

$\theta \in \Theta$ is a set of tuples, where $\Theta = \mathscr{P}(\mathfrak{Q} \times \{req, preq, forb, pforb\})$. Each such tuple refers to another query Q' in $\mathfrak{Q}$ and a constraint *req*, *preq*, *forb*, or *pforb*. The constraint determines whether occurrences of Q' are allowed/required to lie on the path that is mapped to the query edge. For example, a query edge with $\theta \neq \emptyset$ can be mapped to a path in a model if the following requirements hold [8, p. 485]:

- $(Q', req) \in \theta$: The path contains at least one complete occurrence of Q', i.e., all elements of at least one occurrence of Q' lie on the path.
- $(Q', preq) \in \theta$: The path contains at least one partial occurrence of Q', i.e., at least one element of at least one occurrence of Q' lies on the path..
- $(Q', forb) \in \theta$: The path does not contain any element of any occurrence of Q', i.e., no element of any occurrence of Q' lies on the path.
- $(Q', pforb) \in \theta$: The path does not contain a complete occurrence of Q' but can contain parts of it.

Finally, $G = (gmaxvo, gmaxeo, R)$ is a set of global rules, which apply to the overall query. *gmaxvo* and *gmaxeo* are numbers handled similarly to *maxvo* and *maxeo*. They define if the matched paths are allowed to overlap among each other and how often. R is a set of rules that are operating on the properties of query vertices and edges. A rule $r \in R$ is an equation that must obey the following syntax given in EBNF [8, p. 485]:

```
rule = ["NOT"] subrule | ["NOT"] "(" subrule boolop subrule")";
subrule = rule | comparison;
boolop = "AND" | "OR" | "XOR";

comparison = compitem compop compitem;
compop = "==" | "!=" | "LIKE" | "<" | ">" | "<=" | ">=";
compitem = number_expression | string_expression | boolean;

number_expression = number calcop number;
number = "(" number_expression ")" | number_function |
  number_primitive;
number_function = "[" ID "].[" ID "].value" | "[" ID "]."
  property "." aggregate;
property = "PREDECESSORS" | "SUCCESSORS" |
  "UNDIRECTED_NEIGHBORS" | "NEIGHBORS" |
  "OUTGOING_EDGES" | "INCOMING_EDGES" |
  "UNDIRECTED_EDGES" | "EDGES" | "NODES";
aggregate = "count()" | "max([" T_V "])" | "min([" T_V "])" |
  "avg([" T_V "])" | "sum([" T_V "])";
calcop = "+" | "-" | "*" | "/" | "%";
number_primitive = INTEGER | FLOAT;

string_expression = single_string {"+" string_expression};
single_string = string_function | string_primitive;
```

```
string_function = "[" ID "]."type | "[" ID "].[" ID "].value";
string_primitive = STRING;

Boolean = "[" ID "].[" ID "].value" | "TRUE" | "FALSE";
ID = VID | EID;
```

Such a rule evaluates properties of query matches and returns either *TRUE* or *FALSE*. Only those matches the rules of which evaluate to *TRUE* are returned. The properties of everything that can be given an ID in the query can be compared (i.e., vertices, edges, and paths). For instance, we can compare not only values of attributes (i.e., strings, numbers, or Boolean values) of vertices and edges and types of vertices and edges (i.e., strings) but also other aggregated properties such as the number of adjacent edges or the maximum value of an attribute of the neighboring vertices (i.e., numbers).

Strings, numbers, and Boolean values can be compared via comparison operators (*compop*) that evaluate to *TRUE* or *FALSE*, while *LIKE* only works on strings and >, <, <=, and >= only work on numbers. Attribute values can be strings, numbers, or Boolean values. Properties describe the environment of a vertex or edge (*PREDECESSORS*, *SUCCESSORS*, etc.) and can be evaluated with different aggregate functions (*count*, *max*, etc.) and always return numbers. Numbers can be combined with common math operators (*calcop*), and strings can be concatenated using the + operator.

Consider the following example describing a rule that works on a fictional query containing a vertex v_1, and an edge e_1, among others, where $vid(v_1) =$ "A", $eid(e_1) =$ "B", $minl(e_1) = 1$, and $maxl(e_1) = -1$. The following rule would require that the number of edges a vertex mapped to v_1 is connected to must equal the sum of the values of attributes of the type *cost* connected to each of the vertices that exist on the path mapped to e_1.[4]

```
[A].edges.count() == [B].nodes.sum([cost])
```

4.2 Notation

DMQL queries are mainly visualized through graphical symbols (cf. Fig. 5 and [8]). As DMQL queries have several properties, we cannot display all of them with graphical symbols without overcomplicating the query. Hence, some of the properties are specified with lists and menus. Consequently, it only makes sense to use DMQL when it is implemented as a modeling software add-on.

[4] A full description of properties and aggregate functions can be found at https://em.uni-muenster.de/wiki/GraphBasedModelAnalysisPlugin/DMQL#Global_Rules.

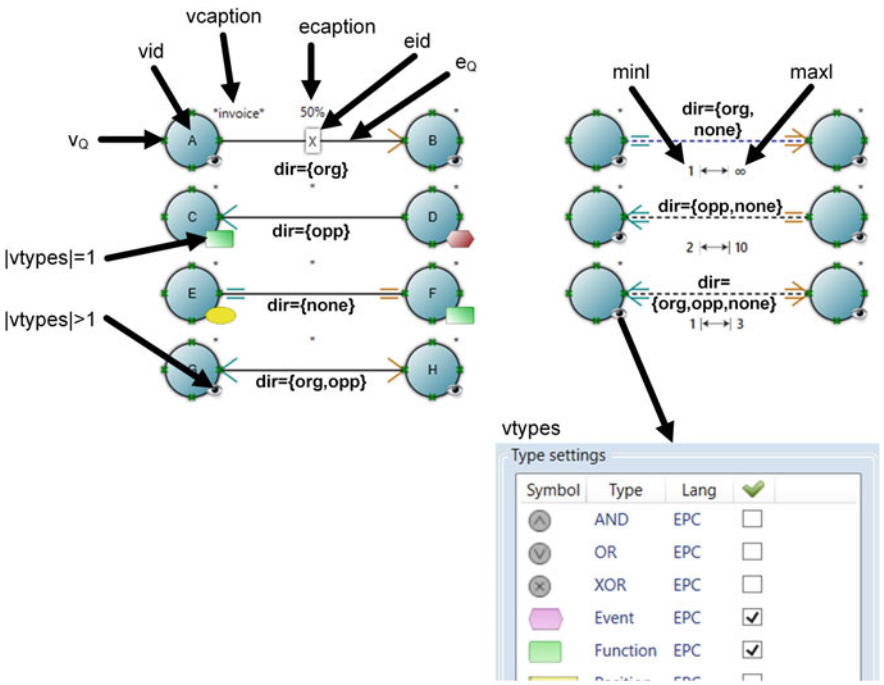

Fig. 5 DMQL concrete syntax examples: Basics

In Fig. 5, shaded circles represent query vertices V_Q with their IDs placed within them. Their captions *vcaption* are placed onto the upper right corner (e.g., "*invoice*" in the figure). Depending on the number of allowed vertex types *vtypes*, we either see the corresponding symbol of the vertex type at the lower right corner, if $|vtypes| = 1$, or a small eye symbol, otherwise. The *vtypes* property is set by using a list. The list can also be used to look up the allowed vertex types in case $|vtypes| > 1$, refer to the figure.

Edges (E_Q) are represented by line segments drawn between vertices. The edges' IDs (*eid*s) are placed onto the lines and their captions (*ecaption*s) onto their side. Edges look different depending on their direction settings *dir* (cf. Fig. 5). While edges with $maxl = 1$ are solid, edges with $maxl > 1$ are dashed. In the latter case, *minl* and *maxl* are shown on the side of the edge.

All further properties of edges are set using lists that contain the currently supported property values (e.g., the vertex types of the currently supported modeling languages) or using free-text forms (cf. Fig. 6). In the figure, we show such selection lists for *etypesr* with $maxl = 1$, *etypesr* and *etypesf* with $maxl > 1$, *vtypesr*, *vtypesf*, and θ. The *minvo*, *maxvo*, *mineo*, and *maxeo* parameters are adjusted using forms. For global rules, we use forms for *gmaxvo* and *gmaxeo* and a formula editor for the rule equations R.

Fig. 6 DMQL concrete syntax: Edge settings and global rules

4.3 Semantics

Due to space restrictions, we abstain from defining DMQL's semantics here and refer the reader to previous work [8]. We have already commented on the impact of each component of a query onto the query results in Sect. 4.1. The corresponding matching algorithm is similar to the brute-force subgraph homeomorphism algorithm [20] implemented using depth-first search. It is extended by several checks that examine whether the vertex, edge, and path mapping candidates in the model graph meet the requirements of the query vertex and edge properties (i.e., P_V and P_E), which makes it possible to exclude candidates early and this way keep the runtime relatively fast (cf. Sect. 5.1).

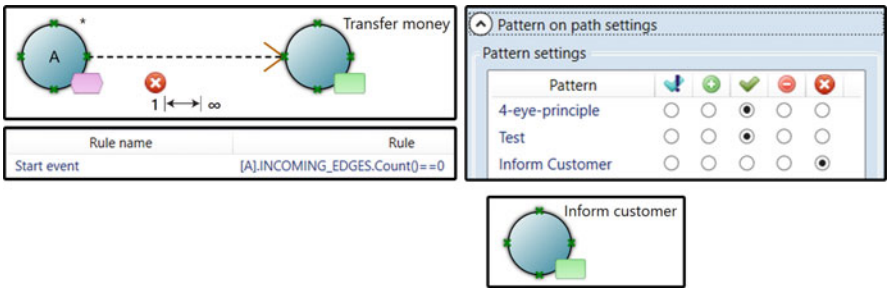

Fig. 7 An exemplary DMQL query

4.4 Query Example

Figure 7 shows an example DMQL query tailored for EPC models. It represents a violation of a compliance rule used in credit application processes. The rule requires that a customer has to be informed about all aspects of a granted credit *before* the money can be transferred. In this example, we assume that the models are terminologically standardized, so we do not have to cope with name clashes (e.g., we search for "check bill" but the modeler used "invoice auditing") as there are several approaches that already solve this problem (e.g., [17]). A violation of the compliance rule means that we find an EPC function somewhere in the model performing the money transfer, but the customer was *not* informed before. In order to identify such violations, we formulate a query that searches for a path from any start event of the process model to the transfer function with *no* function on the path that informs the customer.

The query is assembled as follows: we define a vertex A allowing only events as types ($vtypes = \{event\}$), however, arbitrary captions ($vcaption =$"*"). The vertex connects to another vertex via a path of arbitrary length ($minl = 1$, $maxl = -1$), where the direction points to the second vertex ($dir = \{org\}$). The second vertex must be a function ($vtypes = \{function\}$), and its caption should be $vcaption =$"Transfer money". As we do not access the second vertex in any global rule, we do not have to define any ID for it. To assure that the returned paths start at a start event of the process model, we define a global rule $[A].INCOMING\_EDGES.Count() == 0$ that requires vertex A to have no incoming edges (i.e., to be a start event). The path has a further property that forbids any function on it carrying the caption "Inform customer". The property is realized through a forbidden sub-pattern, i.e., occurrences of a sub-query that are not allowed to be part of the path. This sub-pattern is named "Inform customer" ($\theta = \{$("Inform Customer", $forb$)$\}$; see selection list on the right-hand side of the figure, and note that the selection list lists all available queries of $\mathfrak{Q}$, which is why we also find other entries in the list, which are not selected as req, $preq$, $forb$, or $pforb$ for the current query). It consists of a single function with the caption "Inform customer" (see the lower right box in the figure).

4.5 DMQL 2.0

The reason why we decided to develop a new version of DMQL was that both GMQL and DMQL were sometimes criticized due to their inability to express queries requiring universal and negation properties. In other words, except the feature that GMQL and DMQL can express that certain elements should not be part of a path, they cannot express that certain elements should **not** be part of a query occurrence in general. In turn, they also cannot express that properties of a model must **always** be true (e.g., if there is activity A, it must **always** be followed by activity B). As such features are common in some related query languages (e.g., those based on Computation Tree Logic [CTL; [7]]), we decided to include such features in DMQL, too. To cover the **always** and **not** requirements, we introduce the following new concepts in DMQL: *forbidden vertices*, *forbidden edges*, and *forbidden paths*.

These three concepts cover situations that require negation, that is, cases where certain elements should **not** be part of a query occurrence. In order to also cover universal statements, we make use of anti-patterns using the "forbidden" concepts. For instance, if it is required that specific activities must always be executed by a specific person, one can search for activities that are connected to a forbidden vertex representing that person. If we get a match, then we have found a violation.

In particular, the extensions are as follows. Firstly, we extend the definition of the properties of query vertices P_V and query edges P_E, such that $P_V \subseteq VID \times VCAPTION \times VTYPES \times FORBV$ and $P_E \subseteq EID \times ECAPTION \times DIR \times MINL \times MAXL \times MINVO \times MAXVO \times MINEO \times MAXEO \times VTYPESR \times VTYPESF \times ETYPESR \times ETYPESF \times \Theta \times FORBE$, where $FORBV$ and $FORBE$ are sets of Boolean values. When $forbv = TRUE$ (respectively, $forbe = TRUE$) the vertex (respectively, the edge) becomes forbidden. This means that a query occurrence will only be returned if the forbidden vertex (respectively, the edge) with all its other properties is **not** part of the occurrence. For the forbidden elements, all properties take effect, that is, only those elements are forbidden that match the properties. For instance, if we search for vertices with $vtypes = \{function\}$ connected to other vertices with $vtypes = \{document\}$, where the latter are forbidden, then DMQL would only return functions that are **not** connected to documents. Forbidden edges and paths are handled analogously. If a forbidden vertex is connected to the rest of the query, then its connection (i.e., the query edge) is automatically forbidden, too. Queries may contain multiple forbidden elements. In such a case, only if **all** forbidden elements are found in a prospective query match, the match is excluded from the return set. Forbidden vertices and edges are colored red and tagged with a small "X" in DMQL queries.

Consider the EPC query example in Fig. 8. The query searches for functions that may be connected via a directed path of arbitrary length. Matches are only returned if both of the functions are **not** connected to organizational units (denoted by yellow ovals on the lower right of the vertices) as the organizational units are forbidden.

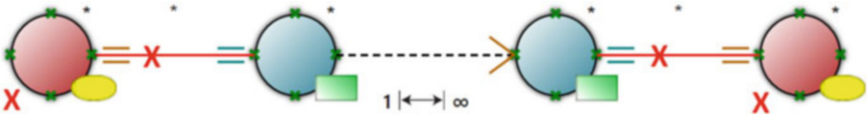

Fig. 8 Example query with forbidden vertices

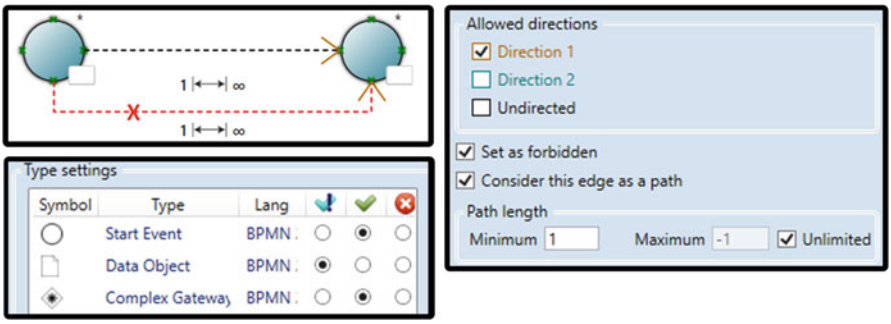

Fig. 9 Example query with a forbidden path edge

Another example shows a BPMN query (cf. Fig. 9). The query searches for subsequent tasks that may follow each other across a path of arbitrary length. These tasks are required to not communicate using data objects. The query realizes this requirement with an additional directed path between the tasks, which is set "forbidden" (see "forbidden" tag in the path properties menu on the right-hand side of the figure; the forbidden path is the lower one), where the path is required to contain at least one data object (see vertex-on-path settings in the selection list on the lower left side of the figure). So, if there is such a path between the tasks, they are not returned as a match.

5 Evaluation

The evaluation of GMQL and DMQL is manifold. First, the runtime complexity of a query language's matching algorithm gives first hints of its applicability. Second, as the runtime complexity is a theoretical construct that does not necessarily consider the actual performance of an algorithm when applying it in practice, an empirical assessment of its performance is also of interest. Third, a query language can be evaluated against its capabilities to express interesting queries, the effort it takes to formalize queries, and the commercial success, that is, how a company can benefit from using the query language. Next, we evaluate these aspects for GMQL and DMQL.

5.1 Runtime Complexity

The runtime complexity of GMQL strongly depends on the functions and operators used in the queries. It reaches from $O(|Z|)$ for *ElementsOfType* to $O(|Z|!)$ for path functions. The complexity of *AdjacentSuccessors*, for instance, is $O(|Z|^2 \cdot degree)$, where *degree* is the maximum vertex degree of the input model [10].

DMQL uses an extension of a subgraph homeomorphism algorithm that exploits the attributes of the vertices and edges of a query. This way possible matching candidates can be excluded early, and the average runtime complexity decreases. In the worst case, that is, if we define a pattern in which we leave every property open (i.e., we allow any vertex and edge type and caption and any path length), the query turns into a subgraph homeomorphism problem, the complexity of which is superexponential [20].

Such a complexity prohibits any practical use prima facie. However, conceptual models are commonly sparse [22], which influences the runtime positively.

5.2 Performance

GMQL and DMQL were implemented as plug-ins for the meta-modeling tool named [εm], which we have developed over the years.[5] It comes with a meta-modeling environment, where the user can specify the abstract and concrete syntax of modeling languages. Subsequently, the languages can be used to create conceptual models. The users of [εm] are guided to create syntactically correct models.

Both GMQL and DMQL plug-ins provide a query editor and a mapping algorithm, which executes the queries and shows their results by highlighting the query occurrences in the models. Riehle et al. [26] created a video, which shows the whole workflow of specifying a language using [εm]'s meta-model editor, creating models, and running queries with DMQL. Furthermore, a detailed video about the DMQL analysis plug-in is reported in [25].

The performance of the mapping algorithms was improved several times, and we have conducted multiple runtime measurements using real-world model repositories and challenging queries [10, 12]. An experiment consisted of 53 EPCs and 46 Entity-Relationship Models (ERMs [6]) containing from 15 to 264 model elements and 10 different queries of low complexity (i.e., containing few functions of low runtime complexity) to high complexity (i.e., containing several functions of high runtime complexity) for each model. In total, we executed $10 \cdot 53 + 10 \cdot 46 = 990$ queries. We observed runtimes from about 70 microseconds to about 0.9 s (time for one query execution including the return of all matches of the query). In particular, the queries took 633 microseconds on average for the EPC models

[5] https://em.uni-muenster.de.

(standard deviation: 3.55 milliseconds) and 10 milliseconds on average for the ERMs (standard deviation: 45.6 milliseconds). The comparatively high deviations result from few queries showing long runtimes for large models, whereas most of the queries showed short runtimes. We argue that because the size of the models used for the experiment was common (i.e., the number of elements of the models was in a range we often see in process models used in practice), and the high-complexity queries were challenging ones, the results of the experiments are satisfactory. DMQL showed a similar performance; however, it was slightly slower for the most complex query [8].

5.3 Utility

We define the utility of a query language as its ability to provide useful query results when applied to real-world process models with queries that arise from real-world problems. To evaluate the utility of GMQL and DMQL (version 1.0), we applied them in an experiment for concrete purposes in the field of business process management, namely both *weakness detection* (e.g., [1, 9]) and *business process compliance management* (e.g., [2, 4, 18]). We understand as weakness a process part that hinders the process execution or that has potential to be shaped more efficiently. A compliance violation is a process part which does not obey preset (legal) policies. Once possible weaknesses or compliance rules are known, they can be transformed into queries and searched in process models. There are several works on process model weaknesses [1, 5, 28, 33, 34], which we reviewed in order to identify weakness descriptions that can be formulated as queries for the evaluation. In addition, we analyzed 2000 process models of a public administration manually to derive further weaknesses [9]. In sum, we identified 280 process weaknesses and consolidated them into 111 weakness types. We further divided them into seven weakness categories, which we introduce in the following. The first category named *Modeling Error* is concerned with general modeling and syntax errors, for instance, a decision without a subsequent branch in the model. The second category is *Process Flow*, which contains weaknesses that may hinder fluent process execution such as manual editing after copying a document or redundant activities. The *Information Handling* category deals with inappropriate use of information (e.g., data is digitized twice). The *Technology Switch* category addresses inefficient use of technology (e.g., a re-digitization of a document after it is printed). The *Automation* category is concerned with manual tasks such as calculations that could be automated. The *Environment* and *Organization* categories deal with the organization of people, competencies, and communication. The former is directed at the external environment of a company (e.g., multiple contact persons for one customer), and the latter takes the internal view (e.g., "ping-pong" responsibilities). Figure 10 shows an overview of the derived categories including some examples.

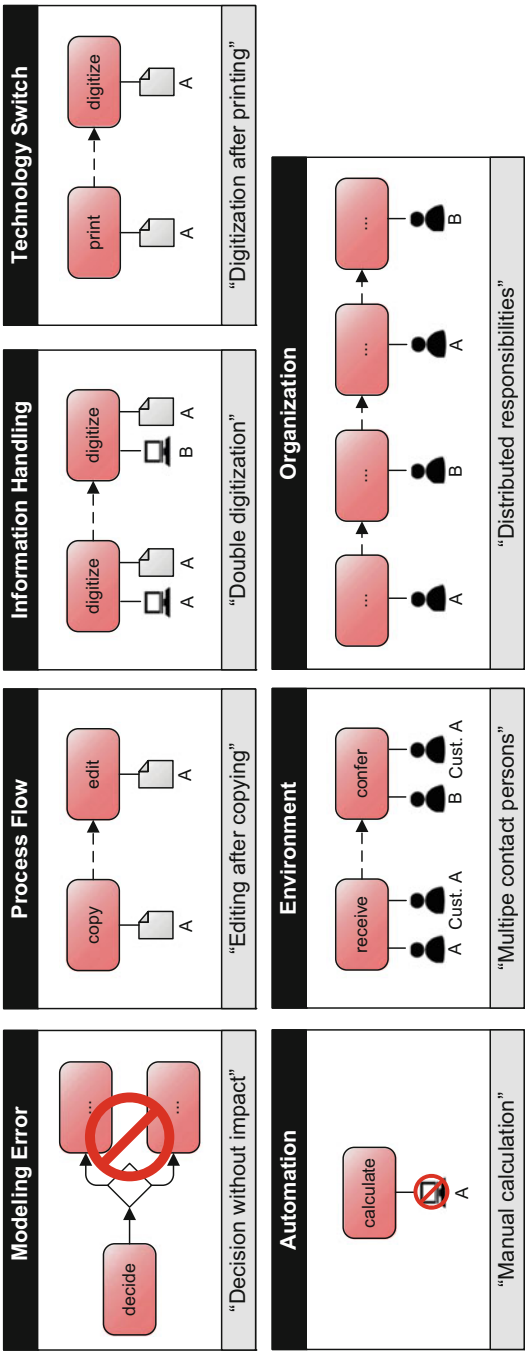

Fig. 10 Weakness categories

We formalized the identified weakness categories in queries and analyzed 85 process models of four companies from the domains of retail, logistics, consulting, and supply. They contained 1995 process activities and 5070 activity attributes such as organizational units and documents [9]. The language of the models was *icebricks* [3], which exists in different versions for different industries and comes with corresponding glossaries that have to be used mandatorily when specifying activity names. A nice side effect was that we did not have to cope with name clashes, as the natural language words used in the queries could be taken from the glossaries.

We analyzed different aspects of utility in the experiment, namely, the amount of issues that can be formalized with GMQL and DMQL, the time it takes to formalize a query, query runtimes, and the quality of the query results.

It was possible to formalize 84% of the derived 111 weaknesses. The remaining 16% could not be formalized; however, this was mainly due to limitations of the modeling language the models of the study were defined in (e.g., there was no object type for IT systems). A minor portion of weaknesses that could not be formalized was due to weaknesses that required negation or universal features that were not included in DMQL 1.0 (cf. Sect. 4.5) and not included in GMQL.

For the DMQL queries, the average formalization time for a single query was 10:18 min. For the same GMQL queries, the formalization took 17:24 min on average. The values include the time needed for the query creation from scratch to the final query without any further need for adaptation (i.e., the queries were correct). We measured the times manually during the creation of the queries. The queries were created by graduate students, who were familiar with the query languages. The final correctness check of the queries was done by the supervisors of the students. In the case of shortcomings, the query had to be revised by the students, and the additional editing time was recorded. We have to note that most of the time was needed for specifying the caption match terms, because the term glossaries contained synonyms.

A single search run (one model, one query, all matches) took 3:45 min on average, even though this strongly depended on the model size. While about 99% of the search runs (nearly 9000 search runs) could be executed in only 27 s on average, the remaining 1% (less than 80 search runs) needed about 4:45 h to execute. These long runtimes were mainly observed in 12 out of the 85 models. In particular, queries in these models took up 94.4% of the total runtime. Correspondingly, Fig. 11 shows that a few queries exhibit a significantly higher search duration than the majority of the search runs. In the figure, only every second query and every fifth process model are explicitly annotated to the axes for reasons of readability. The long runtimes were mainly for queries that searched for unrestricted paths within models. Furthermore, models that provided a high number of path possibilities (several branches and many tasks) induced some of these longer runtimes. The significantly longer runtimes of this experiment compared to those reported in Sect. 5.2 are due

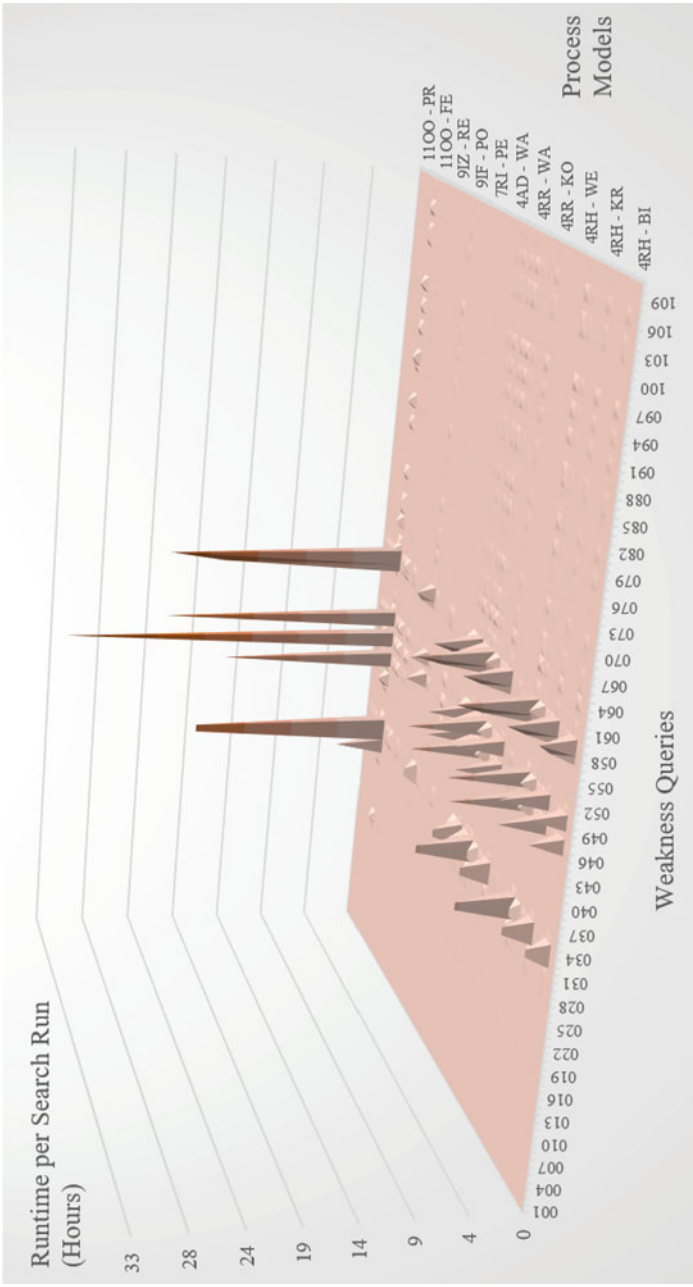

Fig. 11 Runtimes per search run

to significantly larger models examined in the experiment (about five times larger on average).

The searches returned 8071 matches in the process models, of which 998 results were identified as potential weaknesses. This result seems to be fair. However, the results contained 5,364 matches that were duplicates. Duplicates may occur, for instance, when a query contains undirected paths, the start vertices of which have the same properties as the end vertices (because then the path can be "found" from both directions). This insight provided us with valuable information for the further development of the DMQL mapping algorithm, that is, to discard duplicates. As a result, about 12% of the matches were identified as potential weaknesses (37% after removing the duplicates). The potential weaknesses mostly stem from the *Modeling Error* and *Automation* categories (about 85%). Especially, the weakness query "Missing activity attributes" of the *Modeling Error* category, which indicates necessary but missing attributes like organizational units, exhibited a high number of results classified as potential weakness. This is not surprising as we often find such shortcomings in process models, and sometimes such shortcomings are accepted or even regarded as harmless, so, in turn, not regarded as actual weaknesses. Besides possible modeling errors, the *Automation* category mainly concerned with tasks like manual calculations, yielded about 32% of the potential weaknesses. Even though many potential weaknesses stem from only a few queries, approximately one-third of the queries returned at least one match, and we found matches in nearly all models (95%). We further argue that approximately 12 potential weaknesses per process model (998 potential weaknesses in 85 process models) is quite a high number.

In a similar way, GMQL and DMQL can support business process compliance management, which is a well-elaborated topic in the literature, and the need for automatic support is obvious [2, 13, 14, 16, 27, 29, 31]. Thus, we applied GMQL [4] and DMQL [18] in this field. Query-based compliance checking works in the same way as weakness detection. The derivation of compliance rules is different, however, as these are often pre-formulated through regulations or laws. In our work [4], we defined 13 compliance queries textually and in GMQL notation. We later extended the collection to 29 queries and translated them into the DMQL notation. The detailed derivation procedure for DMQL queries (how to turn law texts into queries) is explained in [18]. We further queried process models of an IT service provider for banks with 25 process models containing 3427 activities and 17,616 attributes. The models represented the processes of the software the provider was producing and hosting for affiliated banks. In total, we detected 49 potential compliance violations that finally led the IT service provider to adapt her banking software running in more than 4000 affiliated banks [4].[6]

[6] Note that we cannot publish the name of the IT service provider due to a nondisclosure agreement.

The results are promising and suggest that GMQL and DMQL can be applied successfully in real-world scenarios.

6 GMQL, DMQL, and the Process Querying Framework

When relating GMQL and DMQL 2.0 (in this section referred to as G/DMQL) to the Process Querying Framework [23], we come to the following result: in the category *behavior model*, G/DMQL address *process models* only. Queries have to be *formalized* by persons or can be taken from existing query catalogues.[7] Depending on the used language, the query is either a *textual* or *visual* formalism. G/DMQL support the *modeling* component of the framework, i.e., they can be used to query any kind of process model. Although G/DMQL are not restrictive in what kinds of process model repositories should be queried, we regard it most reasonable to assign them to repositories containing *semi-formal models* as these are the kinds of models they operate on. G/DMQL's *query intention* is *read (retrieve)* and *read (project)*, as they answer both the questions whether a model fulfills a query and return corresponding details about the model (i.e., they pinpoint each query occurrence by highlighting it in the queried model). G/DMQL's *querying technique* is *graph matching* as both are based on adapted forms of subgraph homeomorphism.

G/DMQL operate over *semi-formal* models, i.e., over the formalized graph structure of the models, but not on their execution semantics like, for instance, token concepts or traces. G/DMQL perform *no kind of data preprocessing*, meaning that they operate on the sheer data of the models contained in a database, where the structure of such a database should preferably have a schema similar to the model shown in Fig. 3. No *filtering* takes place with the exception that G/DMQL queries operate over the subsections of the model repository that the user has selected. G/DMQL do not provide a dedicated *optimizing* mechanism. However, GMQL implements *caching*. It caches sub-queries that occur multiple times in a query structure [12]. After a query is executed, G/DMQL present *visualizations* to the user (cf. Sects. 3 and 4), and the results can be browsed to *inspect* where in the queried model the query occurrences can be found. There exists an option to *animate* the execution of a query, so the user can trace intermediate results of the matching algorithm, which are highlighted in sequence in the queried model.

Figure 12 shows the schematic view of the Process Querying Framework taken from [23], where we marked the components addressed by G/DMQL in black. Components that are only partly addressed by G/DMQL are marked in gray.

[7] See, e.g., http://www.conceptual-modeling.org.

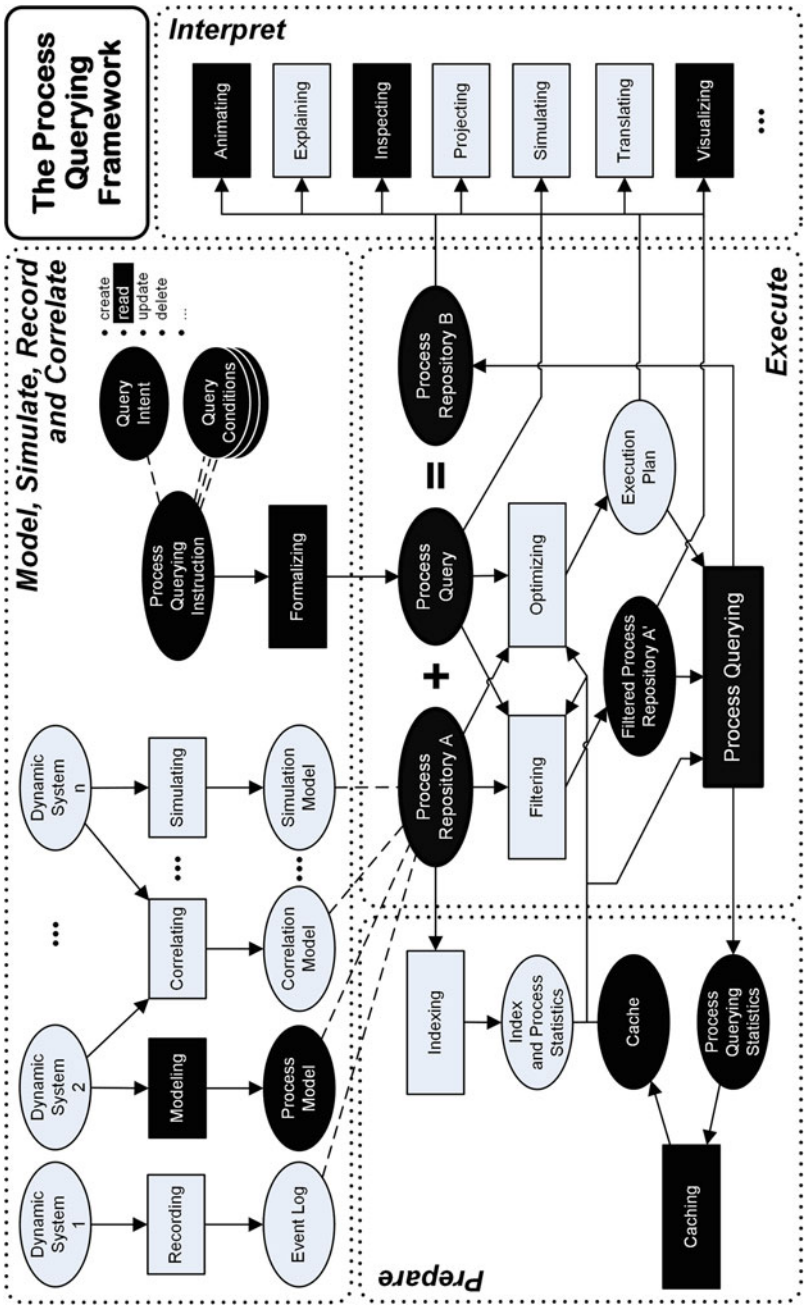

Fig. 12 Framework components addressed by G/DMQL [23]

7 Conclusion

In this chapter, we reported on the development and evaluation of the (process) model query languages GMQL and DMQL. The purpose was not to introduce new query languages—we have done that before—but rather to outline how GMQL and DMQL work and through how many and which iterations the development traversed. We have seen that evolving requirements, criticism, hints, and evaluation both from scholars and professionals have resulted in a research process of multiple iterations. With the latest adaptation of DMQL, we addressed the problem of previously missing negation and universal operators. For the future, we plan to evaluate DMQL 2.0 in further field experiments that might result in further adaptations. Through being able to directly compare several query languages with the help of this book, we also aim to promote discussions on GMQL, DMQL, and their related approaches.

References

1. Becker, J., Bergener, P., Räckers, M., Weiß, B., Winkelmann, A.: Pattern-based semi-automatic analysis of weaknesses in semantic business process models in the banking sector. In: Proceedings of the 18th European Conference on Information Systems (ECIS), Pretoria, South Africa (2010)
2. Becker, J., Delfmann, P., Eggert, M., Schwittay, S.: Generalizability and applicability of model-based business process compliance-checking approaches - A state-of-the-art analysis and research roadmap. BuR - Bus. Res. **5**(2), 221–247 (2012)
3. Becker, J., Clever, N., Holler, J., Shitkova, M.: Icebricks. In: vom Brocke, J., Hekkala, R., Ram, S., Rossi, M. (eds.) Design Science at the Intersection of Physical and Virtual Design, pp. 394–399. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
4. Becker, J., Delfmann, P., Dietrich, H.A., Steinhorst, M., Eggert, M.: Business process compliance checking - Applying and evaluating a generic pattern matching approach for conceptual models in the financial sector. Inf. Syst. Front. **18**(2), 359–405 (2016)
5. Bergener, P., Delfmann, P., Weiss, B., Winkelmann, A.: Detecting potential weaknesses in business processes. Bus. Process Manag. J. **21**(1), 25–54 (2015)
6. Chen, P.P.S.: The entity-relationship model–toward a unified view of data. ACM Trans. Database Syst. **1**(1), 1–36 (1976)
7. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT press (1999)
8. Delfmann, P., Breuker, D., Matzner, M., Becker, J.: Supporting information systems analysis through conceptual model query – The diagramed model query language (DMQL). Commun. Assoc. Inf. Syst. **37**(24) (2015)
9. Delfmann, P., Höhenberger, S.: Supporting business process improvement through business process weakness pattern collections. In: Proceedings of the 12. Internationale Tagung Wirtschaftsinformatik, pp. 378–392. Osnabrück, Germany (2015)
10. Delfmann, P., Steinhorst, M., Dietrich, H.A., Becker, J.: The generic model query language GMQL - conceptual specification, implementation, and runtime evaluation. Information Systems **47**, 129–177 (2015)
11. Diestel, R.: Graphentheorie. Springer, Berlin/Heidelberg, Germany (2010)
12. Dietrich, H.A., Steinhorst, M., Becker, J., Delfmann, P.: Fast pattern matching in conceptual models-evaluating and extending a generic approach. In: EMISA, pp. 79–92. Citeseer (2011)

13. El Kharbili, M., de Medeiros, A., Stein, S., van der Aalst, W.: Business process compliance checking: Current state and future challenges. In: Proceedings of the Conference Modellierung betrieblicher Informationssysteme (MobIS). Saarbrücken, Germany (2008)
14. Elgammal, A., Turetken, O., van den Heuvel, W.J., Papazoglou, M.: Formalizing and appling compliance patterns for business process compliance. Softw. Syst. Model. **15**(1), 119–146 (2016)
15. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) Proceedings of the 7th International Conference on Business Process Management (BPM), pp. 278–293. Ulm, Germany (2009)
16. Ghose, A., Koliadis, G.: Auditing business process compliance. In: Proceedings of the International Conference on Service-Oriented Computing (ICSOC), pp. 169–180. Vienna, Austria (2007)
17. Havel, J.M., Steinhorst, M., Dietrich, H.A., Delfmann, P.: Supporting terminological standardization in conceptual models – a plugin for a meta-modelling tool. In: Proceedings of the 22nd European Conference on Information Systems (ECIS 2014) (2014)
18. Höhenberger, S., Riehle, D.M., Delfmann, P.: From legislation to potential compliance violations in business processes — Simplicity matter. In: European Conference on Information Systems, ECIS '16, pp. 1–15. Istanbul, Turkey (2016)
19. La Rosa, M., Dumas, M., Uba, R., Dijkman, R.: Business process model merging. ACM Trans. Softw. Eng. Methodol. **22**(2), 1–42 (2013)
20. Lingas, A., Wahlen, M.: An exact algorithm for subgraph homeomorphism. J. Discrete Algorithms **7**(4), 464–468 (2009). https://doi.org/10.1016/j.jda.2008.10.003. http://www.sciencedirect.com/science/article/pii/S1570866708000968
21. Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A design science research methodology for information systems research. J. Manag. Inf. Syst. **24**(3), 45–77 (2007)
22. Pflanzl, N., Breuker, D., Dietrich, H., Steinhorst, M., Shitkova, M., Becker, J., Delfmann, P.: A framework for fast graph-based pattern matching in conceptual models. In: IEEE 15th Conference on Business Informatics, CBI 2013, Vienna, Austria, July 15–18, 2013, pp. 250–257 (2013). https://doi.org/10.1109/CBI.2013.42
23. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.: Process querying: Enabling business intelligence through query-based process analytics. Decis. Support Syst. **100**, 41–56 (2017). https://doi.org/10.1016/j.dss.2017.04.011. http://www.sciencedirect.com/science/article/pii/S0167923617300787. Smart Business Process Management
24. Reisig, W.: Petri nets and algebraic specifications. Theor. Comput. Sci. **80**(1), 1–34 (1991)
25. Riehle, D.M., Höhenberger, S., Cording, R., Delfmann, P.: Live query — visualized process analysis. In: Leimeister, J.M., Brenner, W. (eds.) Proceedings der 13. Internationalen Tagung Wirtschaftsinformatik (WI 2017), pp. 1295–1298. St. Gallen, Switzerland (2017)
26. Riehle, D.M., Höhenberger, S., Brunk, J., Delfmann, P., Becker, J.: [εm] — process analysis using a meta modeling tool. In: Cabanillas, C., España, S., Farshidi, S. (eds.) Proceedings of the ER Forum 2017 and the ER 2017 Demo Track, CEUR Workshop Proceedings, vol. 1979, pp. 334–337. Valencia, Spain (2017)
27. Rinderle-Ma, S., Ly, L.T., Dadam, P.: Business process compliance (Aktuelles Schlagwort). EMISA Forum **28**(2), 24–29 (2008)
28. Russell, N., ter Hofstede, A.H.M, van der Aalst, W.M.P, Mulyar, N.: Workflow control-flow patterns: A revised view. BPM Center Rep. **2**, 6–22 (2006)
29. Sadiq, S., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. In: Proceedings of the 5th International Conference on Business Process Management (BPM), pp. 149–164. Brisbane, Australia (2007)
30. Scheer, A.W.: ARIS—Business Process Modeling. Springer Science & Business Media (2012)
31. Schumm, D., Turetken, O., Kokash, N., Elgammal, A., Leymann, F., van den Heuvel, W.J.: Business process compliance through reusable units of compliant processes. In: Daniel, F., Facca, F.M. (eds.) Current Trends in Web Engineering, pp. 325–337. Springer, Berlin/Heidelberg, Germany (2010)

32. Steinhorst, M., Delfmann, P., Becker, J.: vGMQL - introducing a visual notation for the generic model query language GMQL. In: PoEM (2013)
33. Van der Aalst, W.M., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1), 5–51 (2003)
34. Winkelmann, A., Weiß, B.: Automatic identification of structural process weaknesses in flow chart diagrams. Bus. Process Manag. J. **17**(5), 787–807 (2011)