Communications of the Association for Information Systems

Volume 37 Article 24

9-2015

Supporting Information Systems Analysis Through Conceptual Model Query – The Diagramed Model Query Language (DMQL)

Patrick Delfmann
University of Koblenz-Landau, delfmann@ercis.de

Dominic Breuker
Hitfox Group

Martin Matzner
European Research Center for Information Systems, University of Münster

Jörg Becker
European Research Center for Information Systems, University of Münster

Recommended Citation
Delfmann, Patrick; Breuker, Dominic; Matzner, Martin; and Becker, Jörg (2015) "Supporting Information Systems Analysis Through Conceptual Model Query – The Diagramed Model Query Language (DMQL)," *Communications of the Association for Information Systems*: Vol. 37 , Article 24.
DOI: 10.17705/1CAIS.03724
Available at: https://aisel.aisnet.org/cais/vol37/iss1/24

Communications of the Association for Information Systems

Research Paper ISSN: 1529-3181

Supporting Information Systems Analysis Through Conceptual Model Query – The Diagramed Model Query Language (DMQL)

Patrick Delfmann
Institute for IS Research, University of Koblenz-Landau, Germany
delfmann@uni-koblenz.de

Dominic Breuker
Hitfox Group, Germany

Martin Matzner
European Research Center for Information Systems, University of Münster, Germany

Jörg Becker
European Research Center for Information Systems, University of Münster, Germany

Abstract:

Analyzing conceptual models such as process models, data models, or organizational charts is useful for several purposes in information systems engineering (e.g., for business process improvement, compliance management, model driven software development, and software alignment). To analyze conceptual models structurally and semantically, so-called model query languages have been put forth. Model query languages take a model pattern and conceptual models as input and return all subsections of the models that match this pattern. Existing model query languages typically focus on a single modeling language and/or application area (such as analysis of execution semantics of process models), are restricted in their expressive power of representing model structures, and/or abstain from graphical pattern specification. Because these restrictions may hamper query languages from propagating into practice, we close this gap by proposing a modeling language-spanning structural model query language based on flexible graph search that, hence, provides high structural expressive power. To address ease-of-use, it allows one to specify model queries using a diagram. In this paper, we present the syntax and the semantics of the diagramed model query language (DMQL), a corresponding search algorithm, an implementation as a modeling tool prototype, and a performance evaluation.

Keywords: Information Systems Engineering, Conceptual Modeling, Conceptual Model Analysis, Model Query, Query Language, Graph Search.

The manuscript was received 09/24/2014 and was with the authors 3 months for 2 revisions.

1 Introduction

In this paper, we introduce, implement, and evaluate a diagramed query language for conceptual models. Conceptual models are labeled and typed graphs that one can use to describe and subse¬quently analyze an organization from the perspective of its supportive information systems (Davies, Green, Rosemann, Indulska, & Gallo, 2006). Examples of conceptual models are process models that describe the order in which a set of busi-ness activities are executed to achieve a specific business goal (Weske, 2007), data models that capture the structure of data necessary to execute business tasks (Chen, 1976), or organizational charts (Haskell & Breaznell, 1922) represent¬ing the relationships of employees and departments in a company. Popular notations of conceptual models are event-driven process chains (EPCs (Scheer, 2000)), the business process model and notation (BPMN, (Object Management Group, 2014a)), and Petri nets (Desel & Juhás, 2001). Wide-spread notations for data modeling are the entity-relationship model (ERM, (Chen, 1976)) and UML class diagrams (Object Management Group, 2014b). Conceptual models do not only serve to document but also to analyze specific aspects of corporate reality in order to support economic decision making (Kottemann & Konsynski, 1984; Karimi, 1988). In this context, we understand "analysis" as the task of extracting relevant structural and semantic information out of conceptual models for a given task. In many cases, such an analysis results in querying a collection of models to identify model fragments with (partly) given structural characteris¬tics and (partly) given contents. Such an analysis may serve various business objectives, such as:

- To *check models for compliance* with laws and regulations (Awad, 2007; Becker, Delfmann, Eggert, & Schwittay, 2012). In the wake of recent financial crises, compliance management has become an im-portant management task that—if done incorrectly or not at all—can potentially be costly for an organi-zation (Ly, Rinderle-Ma, Göser, & Dadam, 2012). Compliance checking requires identifying model fragments, the structure and contents of which indicate a compliance violation (Knuplesch, Rinderle-Ma, Pfeifer, & Dadam, 2010) (see Section 3 for a particular example).
- To *identifying weaknesses in process models to improve business processes* according to efficiency, effectiveness, or quality, which supports, for instance, business process reengineering. Doing so aims to avoid shortcomings such as double work or unnecessary manual processing (Becker, Bergener, Räckers, Weiß, & Winkelmann, 2010; Bergener, Delfmann, Weiss, & Winkelmann, 2015). Identifying such weaknesses re¬quires finding process model fragments whose structure and contents typically indicate a shortcoming of a business process. (see Section 3 for a particular example).
- To *transform conceptual models* from one notation into another one; for instance, from a conceptual into an executable specification, which supports, for instance, software engineering. Notable examples discussed in the literature include translating BPMN models into code of the business process execution language (BPEL) (OASIS, 2007; García-Bañuelos, 2008; Ouyang, Dumas, ter Hofstede, & van der Aalst, 2008) and translating data models into relational schemas (Duddy, Gerber, Lawley, Raymond, & Steel, 2003; Batra, 2005). Translating models requires identifying model fragments of a source notation that are to be trans-lated to model or code fragments of a target notation (see Section 3 for a particular example).
- To check models for *structural or behavioral conflicts* to ensure their syntactical correctness and—in case of process models—their proper execution semantics (Mendling, Verbeek, van Dongen, van der Aalst, & Neumann, 2008). Syntax errors or improper execution semantics may disrupt runtime execution. Hence, identifying such problems can help assuring proper model execution in the case of automation. Identifying structural or behavioral conflicts requires identifying model fragments that indicate such conflicts (see Section 3 for a particular example).
- To *describe a fine-grained model* designed for a technical audience on a higher level of abstraction to render it more suitable for a managerial audience (which is referred to as *model abstraction* (Soffer, 2005; Polyvyanyy, Smirnov, & Weske, 2010; Reijers, Mendling, & Dijkman, 2011; Polyvyanyy, Weidlich, & Weske, 2012) and another kind of model analysis). Abstraction requires identifying frequently occurring model fragments that can be replaced by more abstract single objects (see Section 3 for a particular example).

The application areas mentioned above suggest that querying conceptual models to identify fragments with particular characteristics serves plenty of analysis tasks that are performed to design, re-design, or improve corporate information systems in different ways. Hence, many companies have started to create large collections of conceptual models (Dijkman, La Rosa, & Reijers, 2012). Such collections include different kinds of conceptual models, such as process models, data models, class diagrams, organizational charts, or ontologies. Prominent examples of model collections are the Suncorp collection (http://apromore.org/tag/suncorp) maintained in APROMORE (La Rosa et al., 2011), the SAP reference model (Keller & Teufel 1998), the BIT pro¬cess library (Fahland et al., 2009), or the process repository of Dutch municipalities (Dijkman, Dumas, van Dongen, Käärik, & Mendling, 2011). Such model collections may contain hundreds to thousands of models, and, in turn, each model may consist of hundreds to thousands of elements (Dijkman et al., 2011; Rosemann, 2006). Given the complexity of these collections, the task of analyzing conceptual models is becoming increasingly difficult (Dijkman et al., 2011). Analyzing conceptual models is even more complex considering that such an analysis may serve different business objectives as described above.

Given the necessity of analyzing conceptual models on the one hand and its complexity on the other hand, model querying is a currently intensively discussed discipline in information systems engineering and conceptual modeling (e.g., (Awad, 2007; Störrle, 2011; World Wide Web Consortium, 2014; Song et al., 2011; Delfmann, Steinhorst, Dietrich, & Becker, 2015). Several scholars have proposed model query languages to help analyze both the structure and the semantics of conceptual models (see Section 4 for an overview). Model querying's benefits include time and money savings because desired information can be obtained quickly and at least semi-automatically. Due to steadily growing corporate process, data, organizational, and IT models, manually analyzing such models is not an option anymore (Rosemann, 2006; Dijkman et al., 2012).

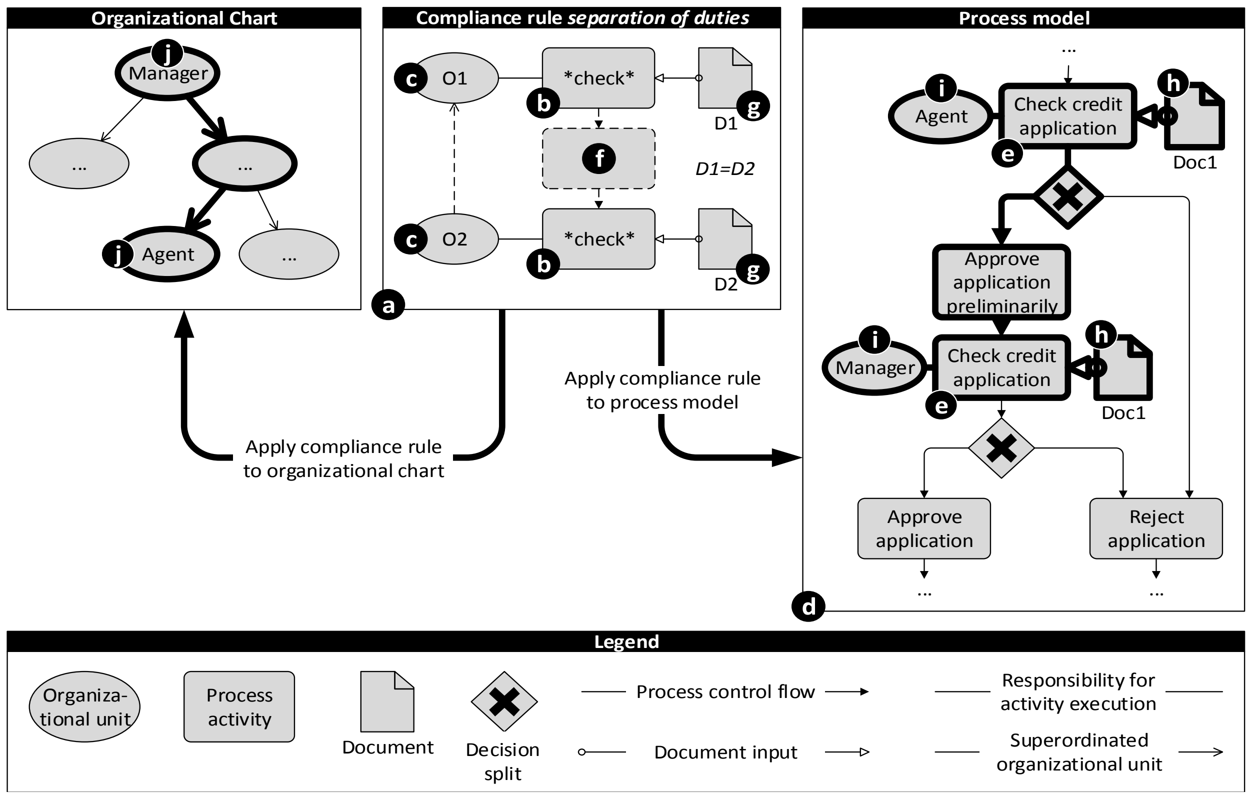


Figure 1. An Exemplary Model Query to Check for Compliance

As a particular model query example, consider the situation depicted in Figure 1 (cf. Delfmann et al., 2015): In the example, a compliance rule requires a business process to meet the *separation of duties principle* (a). It demands process participants to perform double checks (b) on critical documents (e.g., a credit application), where the second person performing the check must be superordinate to the first one (c). A process model being compliant with this rule (d) must contain two process activities that perform a check (e). Thus, both of the activities must be labeled with a phrase containing the term "check" (cf.

activity symbols in the pattern (b) and in the process model (e)). The activities must follow each other either directly or across an activity path of previously unknown length (indicated by the dashed symbols and arrows in the pattern (f)). Both activities have to act on the same document (i.e., the document that is to be checked) (cf. annotated documents in the pattern and the expression D1=D2 that requires that the activities must act on the same document (g), and the corresponding documents in the process model (h)). Furthermore, both activities are performed by different organizational units (i), of which the second is superordinate to the first one. The figure indicates this relationship through a directed path of disciplinary relationships leading from the second organizational unit to the first one (c). Hence, in the organizational chart of the company, we need to find a construct that expresses the following circumstance: the organizational unit who executes the second check activity (here: the manager) must be superordinate to the organizational unit who executes the first check activity (here: the agent) (j).

Validating a company's information systems against this compliance rule requires applying the rule to both the company's process models and its organizational charts. First, a process that complies with this rule must contain the check activities, the path between them, and their connection to the specified documents and organizational units. Second, at the same time, the company's organizational structure has to contain information about the second organizational unit's superordination to the first one. In the example, we highlight the shapes of the process model's and organizational chart's fractions matching the compliance rule in bold.

We also check the models for compliant subsections. If we wanted to check the models for non-compliant subsections (i.e., in order to find compliance violations), we could, for example, first search for the succeeding check activities and then for the superordinate organizational units. If one of the searches fails, we know that there is a compliance violation. Another possibility of searching for compliance violations would be to formulate anti-patterns. In the example, the anti-pattern would express that two succeeding check activities were not performed by superordinated organizational units but by other ones.

A common characteristic of many recent model query approaches is that they are specifically designed to support one particular business objective (e.g., compliance checking) (e.g., (Yan, Dijkman, & Grefen 2012; Polyvyanyy et al., 2012)). Some are even specialized to analyze one specific aspect of a specific kind of models (e.g., analyzing execution semantics of process models) (e.g., Song et al., 2011; Förster, Engels, Schattkowsky, & van der Straeten, 2007)). In addition, some of the query languages are designed for analyzing models developed in a particular modeling language (e.g., Awad, 2007; Beeri, Eyal, Kamenkovich, & Milo, 2008). Some even restrict queries in their structural expressive power; that is, the kinds of structures identified pattern matches can have are restricted, for instance, to linear structures (i.e., paths in a model) (Woerzberger, Kurpick, & Heer, 2008) or tree-like structures (i.e., paths in a model with annotations) (Elgammal, Turetken, van den Heuvel, & Papazoglou, 2010).

Instead of developing such customized query approaches, we follow van der Aalst (2013) who argues that, first, it is more beneficial to develop a query approach that can be used to support multiple business objectives. Second, we argue that it is beneficial to develop an approach that is by default *applicable to multiple types of conceptual models* (i.e., process models, data models, etc.) and models of any modeling language. Companies that wish to (semi-)automatically analyze their models may not be able to use a given specialized query approach if it does not fit their use case or their preferred modeling language. Furthermore, for reasons of economy and company-internal conventions, they may not be willing to change their preferred and established modeling language for every new analysis purpose to render some specialized analysis approach applicable (van der Aalst, 2013). Third, some model analysis tasks even require one to analyze multiple models of different modeling languages and different modeling views concurrently (i.e., to extract information that spans, for example, over a process model and an organizational chart; cf. example in Figure 1). Fourth, to be as flexible as possible in structurally specifying query patterns, we argue that a query language should allow one to specify query patterns and to identify pattern matches of any possible graph structure (i.e., not only tree-like or path-like structures but also those including, for example, loops and grids; cf. Figure 1). A more generalized approach to querying conceptual models realizing these four features could serve as an enabler for wide-spread applications of (semi-)automatic model analysis.

A previous case study-based utility evaluation of a related multi-purpose and language-independent model query language (Delfmann et al., 2015) revealed that stakeholders using conceptual models perceive such a mechanism to be highly beneficial to support model analysis (Becker, Delfmann, Dietrich, Eggert, & Steinhorst, 2015). The query language presented in that study, however, uses formal set operations represented by nested formulas to define pattern queries. The stakeholders in the study argued that the ease of use of the evaluated language was rather low because pattern specification was cumbersome (cf. application examples in (Becker et al., 2015)). Hence, fifth, we argue that a model query language is much easier to use if it allows one to graphically model a pattern in much the same way as a model is developed (thus, in a diagramed way).

In this paper, we present a multi-purpose modeling language-spanning and modeling view-spanning model query language that allows one to specify patterns of any graph structure in a diagramed way, which we call the diagramed model query language (DMQL). The query language comes with a corresponding algorithm that takes query patterns as input and matches them against conceptual models. As any conceptual model, no matter which modeling language was used to build it, is essentially a graph comprising vertices and edges of different kinds, we base the syntax and semantics of DMQL and the operations performed by the matching algorithm on graph theory.

In graph theory, model querying resembles the graph pattern matching problem of subgraph isomorphism (Ullmann, 1976). Corresponding algorithms are able to detect exact pattern matches. Exact in this context means that a pattern match is detected if and only if a given model fragment complies exactly with the structure of the pattern. Formally, this means that all vertices of a predefined pattern graph are mapped to a subset of vertices of the model graph such that the pattern structure defined by the edges connecting the vertices is preserved in the model subgraph. Consequently, each vertex and each edge in the pattern graph have exactly one corresponding vertex or edge in the mapped model subgraph. Furthermore, the attributes of the pattern vertices and edges (i.e., types and labels) have to match those of their mapped equivalents in the model graph.

In the context of analyzing conceptual models, however, it is often necessary to identify paths of model elements that are of previously unknown length (cf. compliance pattern introduced above and additional patterns described in Section 3). Consequently, a matching mechanism is required that allows one to identify subgraphs in a model that are not strictly identical to a predefined pattern but may contain paths of previously unknown length. In graph theory, this kind of pattern matching is known as the problem of subgraph homeomorphism (Lingas & Wahlen, 2009). Unfortunately, corresponding algorithms produce a huge number of pattern matches because, by default, all pattern edges are mapped to paths of all possible lengths in the model. However, this is often not necessary because only some particular pattern edges may need to be mapped to paths in the model. The resulting huge number of potentially irrelevant matching results leads to increased runtimes (Lingas & Wahlen, 2009). To address the problem of pattern matching in the context of conceptual model query, we include edge-path matching in DMQL, where a vertex in the pattern graph has exactly one equivalent node in the model, but an edge in the pattern graph may be—if so specified—mapped to a path of elements in the model. We call this way of combining parts of subgraph isomorphism and subgraph homeomorphism relaxed subgraph isomorphism.

However, we still have to cope with performance. Because many companies are maintaining large collections of models, the runtime performance of a query language is of paramount importance for its practical applicability. Subgraph isomorphism is known to be NP-complete (Garey & Johnson, 1979), and subgraph homeomorphism even exhibits a super-exponential runtime (Lingas & Wahlen, 2009). Hence, one can expect that runtimes of a relaxed subgraph isomorphism algorithm at least exhibit exponential runtime behavior. However, Cordella et al. (2004) have proposed a subgraph isomorphism algorithm that has proven to return results in satisfactory time despite its theoretical intractability (Cordella, Foggia, Sansone, & Vento, 2004). Runtime tests conducted in several scientific disciplines suggest that VF2 is able to return pattern matches with satisfactory runtime performance (Ferro et al., 2007). A study on the applicability of various subgraph isomorphism algorithms on conceptual models confirms this finding (Breuker, Delfmann, Dietrich, & Steinhorst, 2014). The idea of VF2 is to prune the search space by applying so-called feasibility checks; that is, to check if it makes sense to add a new node and edge to a partial mapping and, if not, discard the partial mapping immediately rather than create a complete mapping and check its feasibility afterwards. We adopt this idea to create a corresponding pattern matching algorithm for relaxed subgraph isomorphism as described above. To summarize, we make the following contributions to the literature:

- We present the novel model query language DMQL that supports multiple business objectives, is applicable to models of different modeling languages and of different modeling views, is able to identify pattern matches of any graph structure, and provides diagramed pattern specification. Although scholars have already articulated the need for such a language (van der Aalst, 2013), none of this kind exists so far.

- We introduce a new type of graph problem called relaxed subgraph isomorphism, which is designed to address the particular requirements of querying conceptual models. Furthermore, we present a novel algorithm to solve this problem that is specifically tailored to consider characteristics of conceptual models such as mixed directed/undirected edges and node and edge semantics (for details, see Section 3 on requirements). Up to now, only algorithms on subgraph isomorphism and subgraph homeomorphism exist, and they do not take configurable edge-path mappings into account as described above. Also, they are restricted to either undirected or directed edges and mostly ignore node and edge semantics, which are important when analyzing conceptual models.
- We provide a prototypical implementation of both the query language and the matching algorithm as a plug-in for a modeling tool to be able to evaluate DMQL in test settings and real-world evaluations.
- Based on the implementation, we provide a performance evaluation of DMQL's matching algorithm to demonstrate that the algorithm returns pattern matches with satisfactory runtime performance. Because a potentially large number of models need to be queried for some model analysis scenarios (Dijkman et al., 2012), the runtime performance of a matching approach is highly important for its practical applicability.

The remainder of this paper is structured as follows (and in a way that complies with a typical software engineering-related design science research process) (Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007): in Section 2, we explain the basics of conceptual models, conceptual modeling languages, and integrated, view-spanning modeling. Based on this understanding, in Section 3, we derive requirements that DMQL should comply with from particular examples of model querying taken from the literature. In Section 4, we conduct a literature review that evaluates existing model query approaches against the requirements to identify research gaps. In Section 5, we specify the abstract and concrete syntax and the formal semantics of the model query language and describe how a matching algorithm searches for DMQL patterns. In Section 6, we demonstrate the query language's applicability by describing a corresponding modeling tool prototype comprising the matching algorithm. Furthermore, we provide a performance evaluation. After discussing the particular contributions of this paper in Section 7, we close with limitations and an outlook to further research in Section 8.

2 Basic Specifications

The DMQL understands every conceptual model as a both directed and undirected graph, which allows multiple edges between the same pair of vertices. Furthermore, the graph is attributed, which means that each vertex and edge of the graph can be annotated with additional information (e.g., names, IDs, types, and so on) (see Diestel (2010) to learn more about graph theory notations we use in this paper). The meta model in Figure 2 depicts this concept, which we adapt from our previous work (Delfmann et al., 2015), as a UML class diagram (Object Management Group, 2014b). Model vertices are common modeling objects, such as particular process model activities, data model objects, or organizational units. We also regard attributes (i.e., properties of vertices such as name, description, duration, etc.) as being a special kind of vertices. Model edges connect vertices either through directed or undirected relationships (e.g., a directed edge for process flows or an undirected edge for data relations), which always start in a source vertex and end in a target vertex. Several kinds of models incorporate both directed and undirected edges (e.g., process models), so we consider both in the meta model through assigning every edge a type defining whether the edge is, for instance, a process flow, a data input, or a communication channel. Furthermore, the type defines whether or not an edge is directed. Similarly, we assign vertices to vertex types (e.g., organizational unit, entity type, name, duration, etc.). To define which edges can connect to which vertices, we establish source and target associations between edge type and vertex type and, thus, specify the syntax of a modeling language. To differentiate between common vertices and attributes, we use a one-by-one assignment of a vertex type to another vertex type (realized through an edge type), where the one, which is exclusively connected by the one-by-one assignment, is the attribute type. To differentiate between vertices and attributes visually, a corresponding modeling tool would have to display common vertices as graphical shapes, whereas attributes would be accessed through opening a context menu. Concerning graphical realization of vertices and edges in general, this framework also allows for displaying vertices and edges in any desired form (e.g., an edge does not have to be displayed as a line but can also be displayed implicitly; e.g., through positioning a vertex shape into a swim-lane).

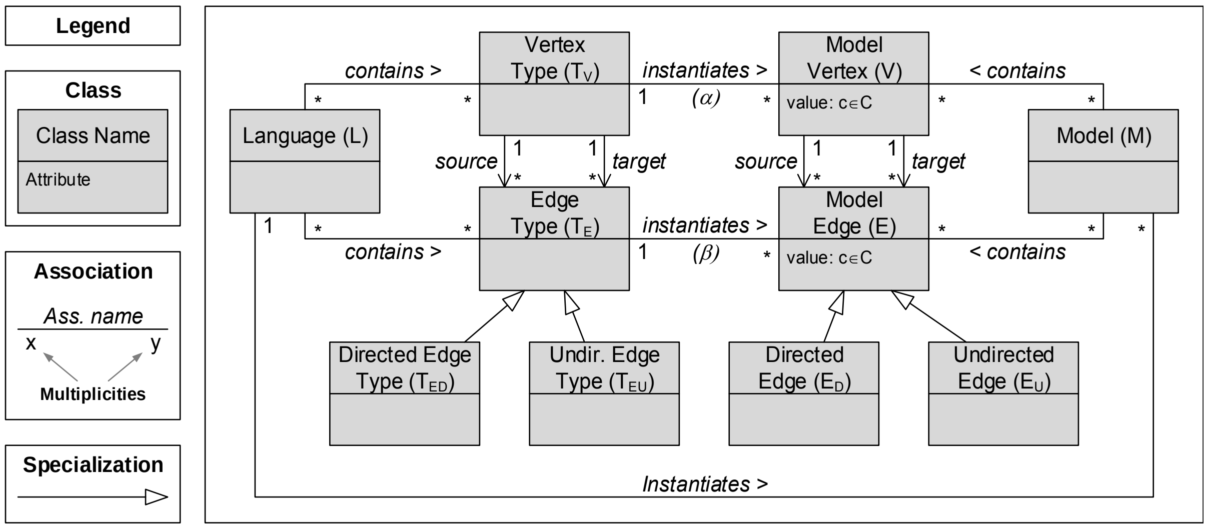


Figure 2. Integrated Meta Model for Modeling Languages and Models

Definition 1 (modeling language): based on the left-hand side of the meta-model, we define a modeling language as follows: a modeling language is a tuple $L=(T_V,T_E)$, where TV is a set of vertex types (e.g., “task” or “class”) including attribute types (e.g., “description”, “cost”, “duration”) and $T_E=T_{ED}\cup T_{EU}$ is a set of edge types. $T_{ED}\subseteq T_V\times T_V$ is the set of directed edge types (e.g., “control flow” or “input”), and $T_{EU}\subseteq\{\{t_{vx},t_{vy}\}\mid t_{vx},t_{vy}\in T_V,\ t_{vx}\neq t_{vy}\}$ is the set of undirected edge types (e.g., “association”, “attribute assignment”).

Definition 2 (conceptual model): we see any conceptual model as an instantiation of a modeling language (see above). Hence, based on the right-hand side of the meta-model, we define a conceptual model as follows: a conceptual model is a tuple $M=(V,E,C,L,T_V,T_E,\alpha,\beta,\chi)$, with V being a non-empty set of vertices and E being a non-empty set of edges. $E=E_D\cup E_U$ comprises directed and undirected edges. As in many modeling languages, multiple relationships are allowed between the same two nodes (similar to, e.g., hierarchy structures in ERMs), we use multi-edges to define relationships as follows: $ED\subseteq V\times V\times N$ is the set of directed edges between the vertices of a conceptual model. $E_U\subseteq\{\{v_x,v_y,n\}\mid v_x,v_y\in V,\ n\in N,\ v_x\neq v_y\}$ is the set of undirected edges between the vertices of a conceptual model. ***N*** is the set of natural numbers used to number multi-edges. Numbering of multi-edges always starts at $n=1$ and increases by one for each additional edge. We *write* $Z=V\cup E$ to denote the set of all model elements, including vertices and edges. α: $V\rightarrow T_V$ is a function assigning each vertex a type T_V. β: $E\rightarrow T_E$ is its counterpart for edges. L is the modeling language defining the interrelations of T_V and T_E; hence, T_V, $T_E\in L$. C is a set of captions serving as names for vertices and edges. Such captions can be assigned to vertices and edges using the function χ: $Z\rightarrow C$. In typical conceptual models, vertices have more attributes than a caption only. As Definition 1 already describes, further attributes are regarded the same as common vertices. They differ in their representation. While vertices are usually displayed as shapes, attributes are commonly not visualized but, in case of the presence of a modeling tool, are accessed via context menus or the like.

Integrated models are models of different modeling languages that use shared concepts (i.e., that have one or more vertex or edge types in common). For instance, some process models and organizational charts share the vertex type “organizational unit”. While these are used to describe the organizational structure of a company in organizational charts, in process models, they are used to explain responsibility for the execution of activities. Hence, the meta model allows one to assign multiple languages to vertex and edge types and vice versa. To really obtain integrated models, one needs to reuse particular vertices. This means that a particular vertex can occur in different models, even in models of different modeling languages. For instance, an organizational unit *CIO* that was originally defined in an organizational chart can be reused to be annotated to a process model activity. This way, it is possible to create integrated conceptual models because they are connected through their shared concepts. Whenever we connect models serving different purposes (such as organizational modeling, process modeling, and data modeling) this way, we speak of view-spanning integrated models (Scheer (2000) provides a very popular definition of integrated models and modeling views). In order to query integrated models, we have to create a unified model consisting of multiple single models. If these models share common vertices or edges, a union automatically creates an integrated, cohesive model that can be searched (see semantics subsection in Section 5 for details). If not, the union creates one model that comprises several model graphs that are not connected to each other (which is not a problem when searching the integrated model).

Definition 3 (pattern occurrence): because the goal of a model query is to receive all subsections of the model that match a query pattern, we define the notion of a pattern occurrence as a subsection of the conceptual model the query is applied to. Actually, pattern occurrences are all subsections of the conceptual model that match the query; hence, a pattern occurrence is a tuple $M'=(V',E',C',TV,TE,\alpha',\beta',\chi')$, where $V'\subseteq V$, $E'\subseteq E$, $C'\subseteq A$, $Z'=V'\cup E'$, $\alpha'=\alpha|_{V'}$, $\beta'=\beta|_{E'}$, and $\chi'=\chi|_{Z'}$.

With these basic specifications, we ensure that DMQL can be applied to preferably any thinkable conceptual model that comprises vertices and edges, regardless of which modeling language or notation is used. This is possible because we define the modeling language of the models DMQL should be applied to as a variable rather than a constant. In this aspect, DMQL differs from related work (see Section 3 and Section 4).

3 Requirements of DMQL

To support multiple tasks of conceptual model analysis, a model query language should comply with several requirements. Such requirements express that a model query language must be able to find substructures in conceptual models that exhibit specific characteristics (e.g., to find substructures that contain a path of which we did not know the exact length at the time we executed the query). The requirements come from different analysis scenarios (e.g., syntax checks, weakness detection, compliance checking, etc.); that is, different analysis scenarios typically require formulating queries of different expressive power. Many of these requirements have already been formulated in the model query literature. Hence, we only give an example for each requirement and refer to the literature for comprehensive collections of examples (cf. references given in Section 4. For an overview, see Delfmann et al. (2015)):

Requirement 1 (R1): the query language should be able to express that a pattern match must contain particular model vertices having particular properties. Properties of vertices are characteristics such as their type (e.g., an activity in a process model or a class in a class diagram) and their caption (e.g., an activity named “process order” or a class named “article”). Consider the example of transforming ERMs into relational schemas (Duddy et al., 2003; Batra, 2005) (cf. Delfmann et al., 2015). A transformation rule may state that a many-to-many relationship type is transformed to a relation, whereas a one-to-many relationship type is implemented using foreign keys. To properly apply this rule, ERM fragments need to be found that represent many-to-many relationship types like it is shown in Figure 3.

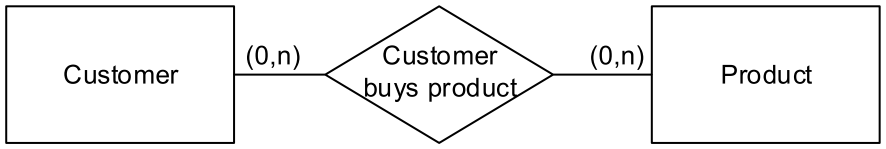


Figure 3. Pattern Requiring Properties of Vertices and Edges

To find such a pattern, DMQL consequently needs to be able to express pattern queries incorporating different types and labels of vertices and edges. More generally, we should be able to express that vertices must have attributes with a particular value in order to be matched. For instance, process model activities are often annotated with attributes such as “average cost”, “average duration”, and so on, which carry particular values. Matching the attributes of a vertex is necessary for almost any model analysis task (cf. for instance, the compliance example of Figure 1; see references given in Section 6 for further examples on vertex/attribute properties; for an overview, see Delfmann et al., 2015).

Requirement 2 (R2): furthermore, for many model analysis purposes, it is necessary to check whether a particular vertex represents the source or the target of a model. This check is especially necessary when models incorporate directed edges such as, for instance, process models. Because many modeling languages do not provide special types of vertices indicating the source or target of a model, we should be able to check how many edges are connected to a vertex. So, for instance, if a vertex is connected to outgoing directed edges but not to incoming ones, we can consider it as the source vertex of a model. Such edge counts are furthermore useful whenever a split or join in process models should be identified. A particular example incorporating edge counts comes from the area of checking models for structural and behavioral conflicts (Mendling et al., 2008):

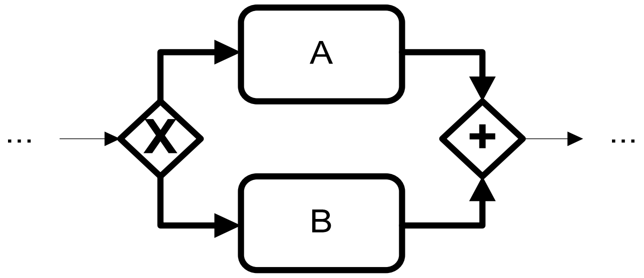


Figure 4. Deadlock Pattern Requiring to Know the Number of Edges

The model fraction of Figure 4 depicts a deadlock represented in a BPMN-like process modeling language (cf. (Mendling et al., 2008). The two parallel process branches are initiated through an XOR split; that is, a process instance takes exactly one of the two process branches. The join operator on the right-hand side, however, is an AND join that expects both process branches to be executed by the same process instance before it can fire. This, however, will never happen because the XOR split only allows one process branch to be executed by the same instance. Hence, the AND join will never fire, so the process will run into a deadlock. To identify such deadlocks, it is necessary to identify the involved joins and splits. Because DQML, as in most modeling languages, has no explicit join or split vertices, which are instead defined by their incoming and outgoing edges, DQML needs to be able to express pattern nodes that have a particular number of edges of particular directions. See references given in Section 4 for further examples on the number of edges. For an overview, see Delfmann et al. (2015).

Requirement 3 (R3): the query language should be able to express that vertices are connected by edges. More particularly, we should furthermore be able to specify what type of edge (e.g., a control flow, a data input, or an association) connects vertices and what direction it must have to be matched (i.e., directed in the one direction, in the other direction, or undirected). The distinction of edge types and their direction is necessary, for instance, to express patterns like they are needed in the area of improving business processes. Here, we search for particular process weaknesses, such as unnecessary switching of automatic and manual processing. A common process weakness is printing a document out of an application system and subsequently entering it into another application system (cf. Figure 5; here, the weakness is represented in a BPMN-like process modeling language).

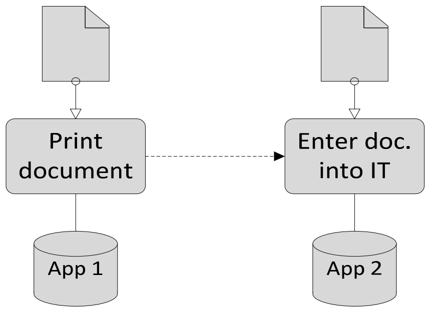


Figure 5. Print/enter Data Pattern Requiring to Know the Direction and Type of Edges (Adapted from Becker, Bergemer, Rackers, Weiss, & Winkelmann, 2010)

To identify such a technology switch in a process model, we need to identify the document outputs of process activities. Hence, we need to identify that a data object is connected to an activity object and, in addition, if it is an input or an output object. These properties can be identified through the type and/or the direction of the involved edge. See references given in Section 4 for further examples on the types and directions of edges. For an overview, see Delfmann et al. (2015).

Requirement 4 (R4): the query language should be able to specify that a pattern edge shall be mapped to a path in a model. This is necessary, for instance, to search for the sequence of activities in Figure 1.

Like for vertices and edges, it should be possible to specify further properties. One should be able to specify which directions the edges existing on a mapped path are allowed to have, vertices of which type are allowed/required to exist on a mapped path, the minimum/maximum length of the mapped path, and, finally, what sub-patterns are allowed to cut a path. These properties are relevant, for instance, for compliance rules that require that certain activities are "forbidden" on process paths (cf., e.g., Elgammal, Turetken, van den Heuvel, & Papazoglou, 2014). A particular example for a pattern requiring such paths comes from the business process compliance checking area. The pattern prescribes that, in a credit application process of banks, one needs to provide a customer with a detailed consulting documentation before the credit can be granted; that is, before the financial transaction can be performed. In order to check a violation of this rule, we could employ the exemplary compliance pattern depicted in Figure 6. It identifies all paths between a consultation activity and a transaction activity, which do not contain a further activity providing the necessary documents and, hence, indicates a compliance violation. See references given in Section 4 for further examples on paths, their properties and required/forbidden objects on paths. For an overview, see Delfmann et al. (2015).

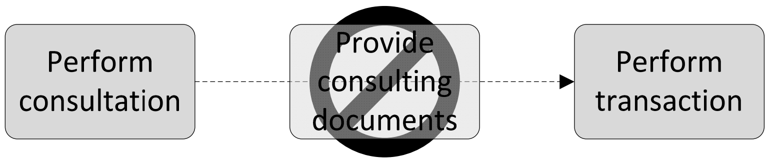


Figure 6. Forbidden Activity Pattern Requiring to Know Whether There is a particular Vertex on a Path

Requirement 5 (R5): because DMQL should be applicable to multiple types of modeling languages, it must consider their common characteristics. We exploit the common characteristic of conceptual models being graphs to be able to apply the query language to any type of modeling language comprising vertices and edges. However, we trade-off this generality for not being able to consider special characteristics of special kinds of modeling languages. For instance, we are not able to express execution semantics of process modeling languages explicitly. Because purely structural model query languages are oftentimes criticized for not being able to express execution semantics (Beeri et al., 2008), we require our query language to at least approximate the analysis of execution semantics. To make a structural query language able to do this, we need to express loops. Traversing a loop means that we search for a path in a model that may cut itself several times, either across one or more vertices or even across one or more edges. Such multiple traversals are needed to express several check operations of execution semantics (cf., e.g., Elgammal et al., 2014). Hence, we require DMQL to express minimum/maximum vertex/edge overlaps for paths. For instance, if we want to check in a process model if an activity "open account" is always preceded by an activity "perform legal customer identification", we need to search for paths from the beginning of the process model to the activity "open account" that does not contain any activity "perform legal customer identification". If we get a match, we know that the rule is violated. Even if the process model contains loops or other process branches that allow a compliant execution, we will find a violation if there is one on another path or loop as soon as we allow enough path overlaps (cf. Beeri et al. (2008) and Delfmann et al. (2015) for a further discussion on this issue).

Requirement 6 (R6): the query language should allow one to compare properties of a pattern's vertices. For instance, to check the compliance rule requiring separation of duties, we need to check whether the involved activities process the same document. That is, we need to express that the labels of the checked documents are the same (cf. Figure 1). See references given in Section 4 for further examples on comparison of properties. For an overview, see Delfmann et al. (2015).

Requirement 7 (R7): to allow for utmost flexibility, we also require that, if the pattern contains multiple paths, it should be possible to specify whether or not these paths are allowed to intersect each other; that is, if they share common vertices and/or edges.

In addition to these requirements, we argue that a query language for conceptual models should be applicable to not only process models, but also to any other kind of conceptual model such as data models, organizational charts, ontologies, and so on. It should also not be limited to analyzing models developed in a particular modeling language (van der Aalst, 2013). Furthermore, as we note in the introduction, we require the query language to also be able to express modeling view-spanning query patterns and the corresponding matching algorithms to apply these queries onto integrated models. Also, the approach should provide graphical query specification (cf. Becker et al. (2015) and Störrle and

Acretoaie (2013) for an additional discussion on the benefit of graphical pattern editors compared to text-based approaches). Therefore, we add three additional requirements.

Requirement 8 (R8): the query language should be applicable to conceptual models of multiple types and languages.

Requirement 9 (R9): the query language should be applicable to view-spanning, integrated conceptual models.

Requirement 10 (R10): the query language should provide a graphical pattern editor that allows modeling a diagramed pattern according to R1 to R9.

4 Related Work and Research Gap

To show that a query language that meets the requirements as derived above does not yet exist, we conducted a literature review on model query languages. In Table 1, we compare the ability of the existing languages to fulfill R1-R10. The table contains a "+" if the query language fulfills a given requirement and a "-" otherwise. The table contains a "0" if the given requirement is only partly fulfilled (see details below; for a detailed discussion of the properties of each query language, see Delfmann et al. (2015)).

While analyzing the different approaches, we could distinguish four classes of them. The first class comprises query languages specialized to query process models. The class comprises the languages Annotated Finite State Automata (aFSA) (Mahleko & Wombacher, 2008), A Process Query Language (APQL) (Song et al., 2011), BeehiveZ (Jin, Wang, Wu, La Rosa, & ter Hofstede, 2010), the BPMN Visual Query Language (BPMN VQL) (Di Francescomarino & Tonella, 2009), BPMN-Query (BPMN-Q) (Awad, 2007), the Business Process Query Language (BPQL) (Momotko & Subieta, 2004), BP-QL (Beeri et al., 2008), CompliancePatterns (Elgammal et al., 2014), the Integrated Process Management Process Query Language (IPM-PQL) (Choi, Kim, & Jang, 2007), the Process Pattern Specification Language (PPSL) (Förster et al., 2007), VisTrails (Scheidegger et al., 2008), and the Workflow Information Search Engine (Wise) (Shao, Sun, & Chen, 2009). Some of these query languages are specifically designed to query models of a specific modeling language. For instance, BPMN-Q is designed to query BPMN models only. APQL is able to query process models of different languages (i.e., R8="0").

The second class comprises query languages specifically designed to query UML models and comprises approaches such as Eclipse Modeling Framework Incremental Query (EMF-IncQuery) (Bergmann et al., 2010) and the Object Constraint Language (OCL) (Object Management Group, 2014c). These languages use declarative statements to specify query patterns.

The third class comprises general graph query languages such as Cypher (Neo4J, 2014) and the SPARQL Protocol And RDF Query Language (SPARQL) (World Wide Web Consortium, 2014). Although these languages could be customized to apply to different modeling languages, this has not been done yet (i.e., R8="0"). Furthermore, these languages do not provide graphical pattern specification; instead, they use a declarative programming language.

Table 1. Comparison of Our Work to Other Model Query Languages

Query language / requirement	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
aFSA	+	+	+	-	-	-	-	-	-	+
APQL	+	+	+	+	+	-	-	0	-	-
BeehiveZ	+	+	+	-	-	-	-	-	-	+
BPMN VQL	+	+	+	+	-	-	-	-	-	+
BPMN-Q	+	+	+	+	+	-	-	-	-	+
BPQL	+	+	+	-	-	+	-	-	-	-
BP-QL	+	+	+	+	+	-	-	-	-	+
CompliancePatterns	+	+	+	+	+	+	-	-	-	+
Cypher (Neo4j)	+	+	+	+	+	+	-	0	-	-
EMF-IncQuery	+	+	+	+	-	+	-	-	-	-
GMQL	+	+	+	+	-	+	-	+	-	-

Table 1. Comparison of Our Work to Other Model Query Languages

IPM-PQL	+	+	+	+	-	-	-	-	-	-
OCL	+	+	+	-	-	+	-	-	-	-
PPSL	+	+	+	+	+	-	-	-	-	+
SPARQL	+	+	+	+	+	+	-	0	-	-
VisTrails	+	+	+	-	-	-	-	-	-	-
VMQL	+	+	+	+	+	-	-	0	-	0
Wise	-	-	+	-	-	-	-	-	-	-
Developed in this paper	**+**	**+**	**+**	**+**	**+**	**+**	**+**	**+**	**+**	**+**

The fourth class comprises query languages with a similar goal as the query language we introduce in this paper. The Visual Model Query Language (VMQL) (Störrle, 2011) and the Generic Model Query Language (GMQL) (Delfmann et al., 2015) are both designed to operate on models of different modeling languages. However, with VMQL, one needs to build a new pattern editor every time a new modeling language should be supported (i.e., R8=R10="0"). The functionality of VMQL has only been shown for UML and BPMN models as yet. Furthermore, VMQL is not designed to handle view-spanning queries and models. GMQL has similar restrictions: it is applicable to arbitrary modeling languages without programming effort. However, it neither provides view-spanning queries nor diagramed pattern specification.

We learned from analyzing the literature that no existing query language yet fulfills the requirements formulated above. As lacking one or more of the requirements would hinder widespread adoption and application in practice (cf. Section 1), it would be beneficial to have a query language that meets all of the requirements. Particularly, it is not feasible for analysts to change their query language every time their currently used query language lacks one of the described requirements. As we can see from the introduction and the derivation of the requirements, this might happen frequently. Furthermore, some of the requirements we have formulated are not supported by any existing query language at all. Hence, we identify a research gap that we close with this paper. The challenge of closing the research gap is to combine the realization of R1-R10 in one single query language.

5 Syntax, Semantics and Matching Algorithm of DMQL

In this section, we introduce the abstract and concrete syntax of DMQL, its semantics, and its matching algorithm.

5.1 Abstract Syntax

Definition 4 (query pattern): every query that is applied to a conceptual model and that returns a set of pattern occurrences comprises a pattern that complies with the requirements derived in Section 3. Hence, a query pattern is a tuple $Q=(V_Q,E_Q,P_V,P_E,\delta,\varepsilon,\Gamma,G)$. R10 requires that every pattern can be drawn like a conceptual model; that is, like a visualized graph. Thus, a pattern comprises vertices V_Q and edges $E_Q=V_Q\times V_Q\times\mathbf{N}$. Each vertex V_Q can be assigned to vertex properties P_V, and each edge E_Q can be assigned to edge properties P_E. The assignment is done by two functions: δ: $V_Q\rightarrow P_V$ and ε: $E_Q\rightarrow P_E$. P_V and P_E are sets of tuples that, in turn, contain the properties for pattern vertices and edges, where $P_V\subseteq\{(vid, vcaption, vtypes)\}$ and $P_E\subseteq\{(eid, ecaption, dir, minl, maxl, minvo, maxvo, mineo, maxeo, vtypesr, vtypesf, etypesr, etypesf, \Theta)\}$. *vid* is a string that represents the ID each pattern vertex can be assigned. The vid is used to denote the vertex to both clarify its role in the pattern and to reuse the vertex' properties in specifying pattern-global rules (see below). *vcaption*$\in$C defines what captions a vertex in a model must have in order to be mapped to the pattern (cf. R1). The comparison of *vcaption* to the captions of the model vertices is conducted by an equivalence relation (see semantics section below). *vtypes* is a set of vertex types; that is $vtypes\subseteq TV\in\Gamma$, where $\Gamma\subseteq\bigcup_i L_i$. Γ is a set of modeling languages the pattern is valid for (cf. R8). This means that we assign every pattern to one or more modeling languages. This way, pattern occurrences that spread across models of different modeling views can be searched for (cf. R9). Hence, *vtypes* specifies what type a model vertex must belong to in order to be a candidate for mapping (cf. R1). *eid* and *ecaption* are handled analogously to *vid* and *vcaption*. *dir* is an enumtype; in particular, $dir\in\mathbf{P}(\{org,opp,none\})$ and $dir\neq\emptyset$. "org" means that a model edge to be mapped must have the same direction as the original direction of the pattern edge (see definition of E_Q). “Opp” means that it must have the opposite direction. “None” means that it must be undirected. As *dir* is an element of the powerset of the different directions, we also allow combinations. For instance, if *dir={org,opp}*, then an edge must either be of the first or the second direction in order to be mapped (cf. R3).

As R4 requires, an edge of a query pattern should not only be mappable to a model edge but also to a model path. Whether or not a pattern edge can be mapped to a path is defined by $minl \in \mathbf{N}$ and $maxl \in \mathbf{N_0}$. They define the minimum and maximum length of a model path that is mapped to the pattern edge. If *minl=maxl=1*, then the pattern edge is mapped to a model edge. If *maxl=0*, the matching algorithm (see algorithm section below) searches for paths of unlimited length. *minvo*,*maxvo*,*mineo* and *maxeo* define whether or not mapped paths are allowed to cut themselves (cf. R5). $minvo, mineo \in \mathbf{N_0}$ and $maxvo, maxeo \in \mathbf{N_0} \cup \{-1\}$. *minvo* and *maxvo* specify the number of minimum and maximum vertex overlaps. For the matching algorithm (see algorithm section below), this means that every time a path cuts itself (meaning that every time the path search reaches a vertex that was already visited), the count of vertex overlaps is incremented by one. A path is only mapped if the vertex overlap count is between *minvo* and *maxvo*. *mineo* and *maxeo* are the edge overlap counterparts. Setting *maxvo* and/or *maxeo* to *-1* means that we allow unlimited overlaps. *vtypesr* defines which vertices of which type are required to be part of a path to be mapped. vtypesf defines which vertices of which type are forbidden to be part of a path to be mapped $vtypesr, vtypesf \subseteq TV \in \Gamma$. $etypesr, etypesf \subseteq TE \in \Gamma$ are their counterparts for edge types. As R4 requires, we should also be able to either require an occurrence of another pattern to be part of a path or not to be part of it. Therefore, we use Θ that is a set of tuples $\Theta \subseteq \{(Q, constraint)\}$, where $constraint \in \{req, preq, forb, pforb\}$. To be mapped a path with $\Theta \neq \varnothing$ must comply with one of the following requirements:

- The path must contain at least one complete occurrence of $Q \in \theta \in \Theta$ if *constraint=req.*
- The path must contain at least one partial occurrence $Q \in \theta \in \Theta$ if *constraint=reqp.*
- The path must not contain any element of any occurrence $Q \in \theta \in \Theta$ if *constraint=forb.*
- The path must not contain a complete occurrence $Q \in \theta \in \Theta$, if *constraint=forbp* but can contain parts of it.

If the pattern edge should be mapped to only a model edge, then *etypesr* is handled analogously to *vtypes* for vertices, and *ecaption*, *vtypesr*, *vtypesf*, *etypesv*, *minvo*, *maxvo*, *mineo*, *maxeo*, and *Θ* are ignored. Vice versa, if the pattern edge should be mapped to a model path, then the directions given by *dir* are applicable to all edges of the path.

G is a set of global rules that are applied to the pattern as a whole. *G* is a tuple *G=(gminvo,gmaxvo,gmineo, gmaxeo,R)*. *gminvo,gmaxvo,gmineo*, and *gmaxeo* define whether not mapped paths of the pattern are allowed to cut each other (cf. R7). $gminvo, gmineo \in \mathbf{N_0}$ and $gmaxvo, gmaxeo \in \mathbf{N_0} \cup \{-1\}$. *gminvo* and *gmaxvo* specify the number of minimum and maximum vertex overlaps. *gmineo* and *gmaxeo* are the edge overlap counterparts. These values are handled similarly to *minvo,maxvo,mineo*, and *maxeo*. However, they specify the allowed overlaps between all different paths of a pattern match.

R is a set of rules that operates on the properties of the pattern’s components. This way, we can, for instance, compare the captions of a pattern’s vertices (cf. R6). A rule $r \in R$ consists of an equation, which has to obey the following syntax specified in EBNF:

```
rule        ::= {“NOT”} subrule | {“NOT”} “(“ subrule boolop subrule “)”;
subrule     ::= rule | calculation;
calculation ::= subcalc calcop subcalc compop subcalc;
subcalc     ::= {“-“} operand | {“-“} “(“operand calcop operand “)”;
operand     ::= subcalc | “[“vid”].” property | primitive;
```

In principle, global rules comprise two different kinds of equation: an outer equation and an inner equation. The outer equation combines one or more Boolean expressions that have to evaluate *TRUE* for a model subsection to be considered as a pattern occurrence (e.g., the name of a vertex should equal “*check*” or the number of edges attached to a vertex should be greater than 2). The inner equations perform *calculation* operations on properties of a prospective pattern occurrence and compare them to other ones or primitives (e.g., the name of a vertex should equal “*check*” *OR* the number of edges attached to a vertex should be greater than 2). Hence, a rule consists of *subrules* connected by logical operators *boolop*. A *subrule*, in turn, consists either of *another complete rule* or a *calculation*. A *calculation* consists of three sub-calculations *subcalc*, the first two of which are connected by a calculation operator *calcop* and which are compared to a third one by a comparison operator *compop*. A sub-calculation can, in turn, comprise a *single, optionally negated operand* or of *multiple operands*, again connected by a *calculation operator*. An operand consists either of *another complete sub-calculation* or of a *property of a pattern vertex* or of a *primitive*. Properties of vertices are accessed by using the *given vertex id* in square brackets followed by a dot and the respective *property*. Properties are the *caption*, the *number of {incoming | outgoing} edges* (cf. R2) of a vertex, and the *type* of the vertex. *Primitives* are either integer or string values that can freely be defined.

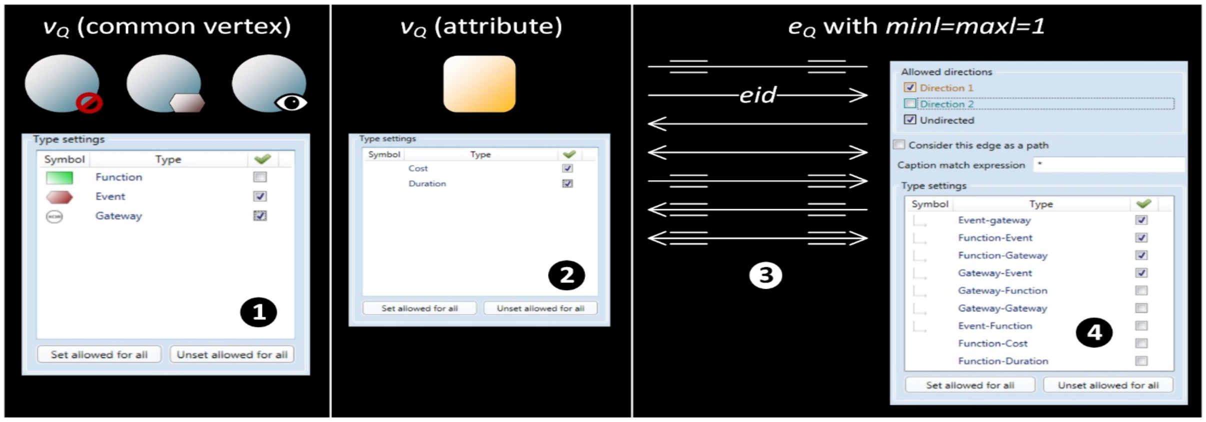
```
boolop    ::= “AND” | “OR” | “XOR”;
mop       ::= “+” | “-“ | “*” | “/” | “%”;
compop    ::= “==” | “!=” | “LIKE” | “<” | “<=” | “>=” | “>”;
property  ::= “caption” | “#inedges” | “#outedges” | “#undedges” | “vtype”;
primitive ::= integer | string;
```

This way, we can specify interrelations of the components of a pattern. For instance, to define that two vertices of a pattern (given the IDs “a” and “b”) should carry the same name or the number of incoming edges of the one vertex should be higher than that of the other one, we would specify the following rule: ([a].caption == [b].caption) OR ([a].#inedges < [b].#inedges).

5.2 Concrete Syntax

As we already state in the introduction and as R10 requires, we provide a diagramed concrete syntax for the query language. That is, a pattern should look like a visualized, graph-like, conceptual model. Hence, we represent part of the query language through graphical vertices and edges. Furthermore, many of the properties of a pattern require choosing particular values from a given set or defining particular numbers or strings. Hence, we represent such properties as selection lists and forms. Because selection lists and forms only make sense in a particular tool implementing the query language, we present the concrete syntax of the query language as it is also used in our implementation here (see Section 6).

We represent a pattern vertex $v_Q \in V_Q$ as a blue circle (see Figure 7, part 1). The representation differs according to the values of $p_V \in P_V$. *vid* is placed within the boundaries of the circle, and *vcaption* is placed at the upper right corner. If |*vtypes*|=*1*, then we add a symbol to the lower right corner of the circle, which equals the representation of the chosen vertex type in a model. If |*vtypes*|>*1*, then the symbol turns into an eye, indicating that a set of multiple vertex types exists. If |*vtypes*|=*0*, then the symbol turns into a no parking sign, indicating that no vertex type has yet been chosen. As Definition 1 already mentions, we regard attributes as common vertices; however, for the sake of clarity, we represent them differently as yellow squares (cf. Figure 7, part 2). We specify and represent all properties of attributes such as for common vertices.



Type settings

Symbol	Type	✓
🟩	Function	☐
⬣	Event	☑
○	Gateway	☑

Type settings

Symbol	Type	✓
	Cost	☑
	Duration	☑

Allowed directions
☑ Direction 1
☐ Direction 2
☑ Undirected

☐ Consider this edge as a path

Caption match expression: *

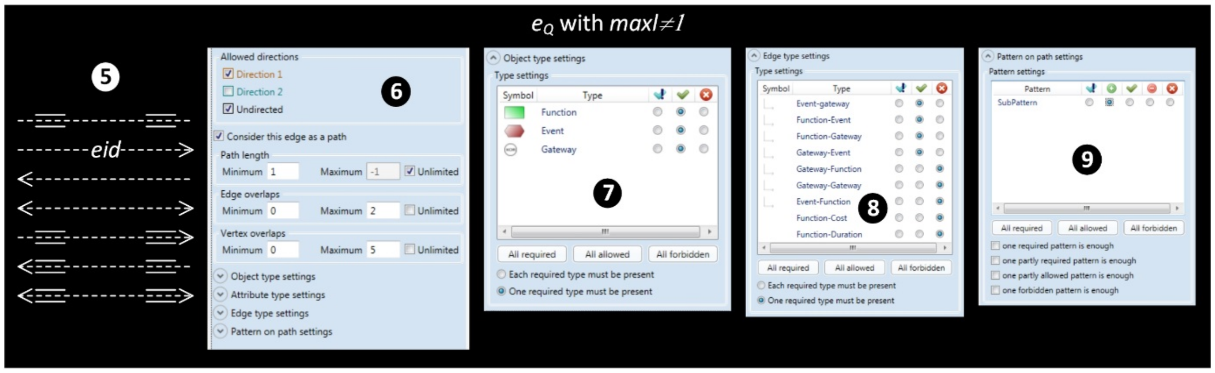
Type settings

Symbol	Type	✓
	Event-gateway	☑
	Function-Event	☑
	Function-Gateway	☑
	Gateway-Event	☑
	Gateway-Function	☐
	Gateway-Gateway	☐
	Event-Function	☐
	Function-Cost	☐
	Function-Duration	☐

Figure 7. The Concrete Syntax of DMQL (Vertices, Attributes, and Edges Mapped to Edges)

The representation of pattern edges differs depending on whether they are mapped to only model edges or to model paths. On the one hand, a pattern edge $e_Q \in E_Q$ is represented through a solid line if *minl=maxl=1* (cf. Figure 7, part 3). *eid* is placed onto the line, and *ecaption* is placed into a form. The visualization of the lines changes corresponding with *dir*. Suppose the original direction of e_Q was from left to right; in this case, e_Q, depending on *dir,* would be represented as follows(from top to bottom in the figure): *{none}*, *{org}*, *{opp}*, *{org,opp}*, *{none,org}*, *{none,opp}*, *{none,org,opp}*. The properties of *etypesr* are represented as a select list.

On the other hand, if *maxl≠1*, then we represent a pattern edge through a dashed line (cf. Figure 8, part 5). *eid* is placed onto the line. The visualization of *dir* is handled analogously to edges with *minl=maxl=1* (cf. Figure 8, part 6). As *minl*, *maxl*, *minvo*, *maxvo*, *mineo*, and *maxeo* are simple numbers, we represent them through a form. *vtypesr* and *vtypesf* are visualized through a select list, as are *etypesr* and *etypesf*. The select lists comprise all vertex types and edge types that are available corresponding with Γ (cf. Figure 8, parts 7 and 8). A further select list represents the settings of Θ. In that list, the query language provides all available patterns (e.g., those patterns that were specified in the past); that is, {Q}. For each pattern, we can specify whether it is required, partly required, partly forbidden, or completely forbidden on a path (cf. Figure 8, part 9). The path overlap rules of G are represented through a form similarly to the path overlap rules of single paths (cf. Figure 8, part 6). The property rules R of G are represented through a form.



Allowed directions
[✓] Direction 1
[ ] Direction 2
[✓] Undirected

[✓] Consider this edge as a path

Path length
Minimum 1 Maximum -1 [✓] Unlimited

Edge overlaps
Minimum 0 Maximum 2 [ ] Unlimited

Vertex overlaps
Minimum 0 Maximum 5 [ ] Unlimited

Object type settings
Attribute type settings
Edge type settings
Pattern on path settings

Object type settings
Type settings

Symbol	Type	!	✓	✗
🟩	Function	○	●	○
🟥	Event	○	●	○
○	Gateway	○	●	○

All required | All allowed | All forbidden
○ Each required type must be present
● One required type must be present

Edge type settings
Type settings

Symbol	Type	!	✓	✗
	Event-gateway	○	●	○
	Function-Event	○	●	○
	Function-Gateway	○	●	○
	Gateway-Event	○	●	○
	Gateway-Function	○	○	●
	Gateway-Gateway	○	○	●
	Event-Function	○	○	●
	Function-Cost	○	○	●
	Function-Duration	○	○	●

All required | All allowed | All forbidden
○ Each required type must be present
● One required type must be present

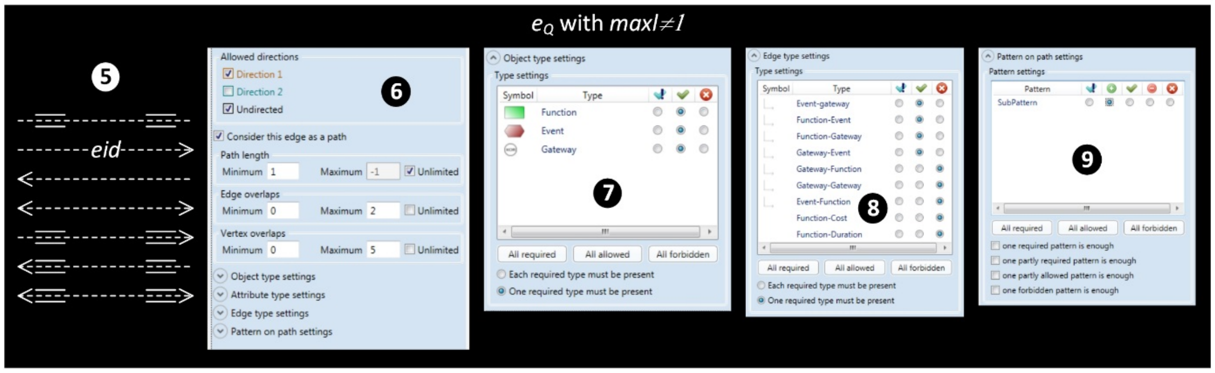
Pattern on path settings
Pattern settings

Pattern	!	⊕	✓	⊖	✗
SubPattern	○	●	○	○	○

All required | All allowed | All forbidden
[ ] one required pattern is enough
[ ] one partly required pattern is enough
[ ] one partly allowed pattern is enough
[ ] one forbidden pattern is enough

Figure 8. The Concrete Syntax of DMQL (Edges Mapped to Paths)

Figure 9 depicts an exemplary query pattern that may span across different modeling languages (in this example, we combined EPCs and organizational charts). It represents a process/organizational model subsection complying with a particular compliance rule. In this case, the rule realizes separation of duties. This principle is mandatory in, for instance, several processes of financial industries. It requires that, in critical business processes, documents are checked twice by two different persons, with the second person being organizationally superordinate to the first one. Hence, a corresponding pattern comprises two process model activities following each other over a directed control flow path of arbitrary length. Both activities are connected to document objects via simple, undirected edges. Furthermore, both activities are executed by organizational units. We express this by annotating organizational units through undirected edges. The condition that the second checking person must be superordinate to the first one is expressed by a directed path from the second organizational unit to the first one.



Allowed directions
[✓] Direction 1
[ ] Direction 2
[✓] Undirected

[✓] Consider this edge as a path

Path length
Minimum 1 Maximum -1 [✓] Unlimited

Edge overlaps
Minimum 0 Maximum 2 [ ] Unlimited

Vertex overlaps
Minimum 0 Maximum 5 [ ] Unlimited

Object type settings
Attribute type settings
Edge type settings
Pattern on path settings

Object type settings
Type settings
Symbol | Type: Function ● allowed; Event ● allowed; Gateway ● allowed
All required | All allowed | All forbidden
○ Each required type must be present
● One required type must be present

Edge type settings
Type settings
Event-gateway, Function-Event, Function-Gateway, Gateway-Event: ● allowed; Gateway-Function, Gateway-Gateway, Event-Function, Function-Cost, Function-Duration: ● forbidden
All required | All allowed | All forbidden
○ Each required type must be present
● One required type must be present

Pattern on path settings
Pattern settings
SubPattern ●
All required | All allowed | All forbidden
[ ] one required pattern is enough
[ ] one partly required pattern is enough
[ ] one partly allowed pattern is enough
[ ] one forbidden pattern is enough

Figure 9. An Exemplary Query Pattern

Formally, the pattern would be specified as follows:

- Γ={EPC,OrgChart}
- EPC=(T_{VEPC},T_{EEPC},P_{TVEPC},P_{TEEPC})
- T_{VEPC}={function,event,orgunit,document,connector}, T_{EEPC}={fe,ef,fo,fd,fc,ec,cf,ce}
- fe=(function,event), ef=(event,function), fo={function,orgunit}, fd={function,document}, fc=(function, connector), ec=(event,connector), cf=(connector,function), ce=(connector,event)
- OrgChart=($T_{VOrgChart}$,$T_{EOrgChart}$,$P_{TVOrgChart}$,$P_{TEOrgChart}$)
- $T_{VOrgChart}$={orgunit}, $T_{EOrgChart}$={superordinate}, superordinate=(orgunit,orgunit)
- Q=(V_Q,E_Q,P_V,P_E,δ,ε,Γ,G)
- V_Q={a,b,c,d,e,f}, δ(a)=(Check1,"*check*",{function}), δ(b)=(Check2,"*check*",{function}), δ(c)=(OU1,"*",{orgunit}), δ(d)=(OU2,"*",{orgunit}), δ(e)=(Doc1,"*",{document}), δ(f)=(Doc2,"*",{document})
- E_Q={g,h,i,j,k,l}, g=(a,b,1), h=(d,c,1), i=(a,c,1), j=(b,d,1), k=(a,e,1), l=(b,f,1)
- ε(g)=("g","*",{org},1,0,0,0,0,0,∅,∅,∅,{superordinate,fo,fd},∅)
- ε(h)=("h","*",{org},1,0,0,0,0,0,∅,∅,∅,T_{EEPC},∅)
- ε(i)=("i","*",{none},1,1,0,0,0,0,∅,∅,∅,∅,∅)
- ε(j)=("j","*",{none},1,1,0,0,0,0,∅,∅,∅,∅,∅)
- ε(k)=("k","*",{none},1,1,0,0,0,0,∅,∅,∅,∅,∅)
- ε(l)=("l","*",{none},1,1,0,0,0,0,∅,∅,∅,∅,∅)
- G=(0,0,0,0,{[Doc1].caption==[Doc2].caption})

Occurrences of this pattern can only be found when considering both the process models and the organizational charts because parts of the pattern use concepts that only occur in process models (e.g., control flow edges) and parts of the pattern use concepts that only occur in organizational charts (i.e., the superordination edge). Hence, the pattern will only produce occurrences when we search for it in multiple models of different modeling languages. A precondition for this is that the modeling languages share common concepts. In the example, the common concept is the vertex type *orgunit.* Another precondition is that the models we search in share common concepts as outlined in the CIO example in Section 2. The common concepts of integrated models then serve as unification points. This means that a model query like it is shown in Figure 9 is executed onto the union of multiple models and, hence, onto an integrated model.

5.3 Semantics

In this section, we describe how a query pattern is mapped onto a conceptual model or, more generally, onto a set of integrated conceptual models. Corresponding with Definitions 1-4, every query takes a set of models $\bar{M} = (\bar{V},\bar{E},\bar{C},\bar{T}_V,\bar{T}_E,\bar{\alpha},\bar{\beta},\bar{\gamma})$, $\bar{Z} = \bar{V} \cup \bar{E}$, where $\bar{V} = \cup_i V_i$, $\bar{E} = \cup_i E_i$, $\bar{C} = \cup_i C_i$, $\bar{T}_V = \cup_i T_{Vi}$, $\bar{T}_E = \cup_i T_{Ei}$, $\bar{\alpha}: \bar{V} \to \bar{T}_V$, $\bar{\beta}: \bar{E} \to \bar{T}_E$, $\bar{\gamma}: \bar{Z} \to \bar{C}$, and a query pattern *Q=(V_Q,E_Q,P_V,P_E,δ,ε,Γ,G)* as input and returns a set of pattern occurrences *M_i'(Q)=(V_i',E_i',C_i',T_{Vi},T_{Ei},α_i',β_i',χ_i')*, where $V'_i \subseteq \bar{V}$, $E'_i \subseteq \bar{E}$, $C'_i \subseteq \bar{C}$, $Z'_i \subseteq \bar{Z}$, $T_{Vi} \subseteq \bar{T}_V$, $T_{Ei} \subseteq \bar{T}_E$, $\alpha'_i = \bar{\alpha}|_{V'_i}$, $\beta'_i = \bar{\beta}|_{E'_i}$, and $\gamma'_i = \bar{\gamma}|_{C'_i}$.

We describe formally how patterns are mapped on to conceptual models in Definition 5 (see below). However, beforehand, to keep the formal definitions as short as possible, we introduce some auxiliary constructs: first, to describe how patterns and models are mapped, we have to access the properties of vertices and edges and, hence, particular components of tuples. Commonly, we could do this by multiplying a tuple with a unit vector. To keep things short, to access a particular component x_i of a tuple *t=(x_1,…,x_n)*, we write *t.x_i* instead, where *t.x_i=t*(y_1,…,y_i,…,y_n)*, *y_1,…,y_{i-1}=0*, *y_i=1*, *y_{i+1},…,y_n=0*.

Furthermore, we have to introduce the notion of a path to express formally that a pattern edge can be mapped to a model path. A path in a model *M* or, more generally, through a set of models $\bar{M}$, is determined through a sequence of vertices and edges *<v_x,e_x,v_{x+1},e_{x+1},…v_{y-1},e_{y-1},v_y>*, such that *e_i=(v_i,v_{i+1},z_{ij})*$\in\bar{E}$ $\vee$ *e_i=(v_{i+1},v_i,z_{ij})*$\in\bar{E}$ $\vee$ *e_i={v_i,v_{i+1},z_{ij}}*$\in\bar{E}$ $\forall$ *i$\in${x..y-1}*, *$z_{ij}\in$***N**. As defined in the syntax of the query language, we can map pattern edges to different kinds of paths, the edges of which have directions according to the property *dir*. Hence, we distinguish the following different kinds of *paths*:

- path(v_x,v_y)$_{\{org\}}$=<v_x,e_x,v_{x+1},e_{x+1},…v_{y-1},e_{y-1},v_y>, $v_i \in \bar{V}$, e_i=(v_i,v_{i+1},z_{ij})$\in\bar{E}$ $\forall$ i$\in${x..y-1}, $z_{ij}\in$**N**

- $path(v_x,v_y)_{\{opp\}}=<v_x,e_x,v_{x+1},e_{x+1},\ldots v_{y-1},e_{y-1},v_y>$, $v_i \in \bar{V}$, $e_i=(v_{i+1},v_i,z_{ij}) \in \bar{E}$ $\forall$ $i \in \{x..y-1\}$, $z_{ij} \in \mathbf{N}$
- $path(v_x,v_y)_{\{none\}}=<v_x,e_x,v_{x+1},e_{x+1},\ldots v_{y-1},e_{y-1},v_y>$, $v_i \in \bar{V}$, $e_i=\{v_i,v_{i+1},z_{ij}\} \in \bar{E}$ $\forall$ $i \in \{x..y-1\}$, $z_{ij} \in \mathbf{N}$
- $path(v_x,v_y)_{\{org,opp\}}=<v_x,e_x,v_{x+1},e_{x+1},\ldots v_{y-1},e_{y-1},v_y>$, $v_i \in \bar{V}$, $e_i=(v_i,v_{i+1},z_{ij}) \in \bar{E}$ $\vee$ $e_i=(v_{i+1},v_i,z_{ij}) \in \bar{E}$ $\forall$ $i \in \{x..y-1\}$, $z_{ij} \in \mathbf{N}$
- $path(v_x,v_y)_{\{org,none\}}=<v_x,e_x,v_{x+1},e_{x+1},\ldots v_{y-1},e_{y-1},v_y>$, $v_i \in \bar{V}$, $e_i=(v_i,v_{i+1},z_{ij}) \in \bar{E}$ $\vee$ $e_i=\{v_i,v_{i+1},z_{ij}\} \in \bar{E}$ $\forall$ $i \in \{x..y-1\}$, $z_{ij} \in \mathbf{N}$
- $path(v_x,v_y)_{\{opp,none\}}=<v_x,e_x,v_{x+1},e_{x+1},\ldots v_{y-1},e_{y-1},v_y>$, $v_i \in \bar{V}$, $e_i=(v_{i+1},v_i,z_{ij}) \in \bar{E}$ $\vee$ $e_i=\{v_i,v_{i+1},z_{ij}\} \in \bar{E}$ $\forall$ $i \in \{x..y-1\}$, $z_{ij} \in \mathbf{N}$
- $path(v_x,v_y)_{\{org,opp,none\}}=<v_x,e_x,v_{x+1},e_{x+1},\ldots v_{y-1},e_{y-1},v_y>$, $v_i \in \bar{V}$, $e_i=(v_i,v_{i+1},z_{ij}) \in \bar{E}$ $\vee$ $e_i=(v_{i+1},v_i,z_{ij}) \in \bar{E}$ $\vee$ $e_i=\{v_i,v_{i+1},z_{ij}\} \in \bar{E}$ $\forall$ $i \in \{x..y-1\}$, $z_{ij} \in \mathbf{N}$

Because we consider conceptual models as containing multi-edges (cf. Definition 2), we have to save both the vertices and the edges in the sequence to be able to trace a path correctly. Furthermore, as in a set of conceptual models, there can be more than one path from v_x to v_y, we introduce the index k and write $path_k(v_x,v_y)_{dir}$.

To map a pattern edge onto a path, we need to know which vertices and edges are participants of a path regardless of how often a path visits a vertex or edge. Thus, we further introduce the operations vertices and edges that transform the vertex and edge sequence of a path into sets simply comprising either vertices or edges. Correspondingly, $vertices(path_k(v_x,v_y)_{dir})=\{v \in \bar{V} \mid v \in path_k(v_x,v_y)_{dir}\}$ and $edges(path_k(v_x,v_y)_{dir})=\{e \in \bar{E} \mid e \in path_k(v_x,v_y)_{dir}\}$.

Furthermore, to map a path, we need to know its length and the number of places where it cuts itself. The length of a path is defined as follows: $length(path_k(v_x,v_y)_{dir})=|vertices(path_k(v_x,v_y)_{dir})|-1$. The number of vertex overlaps is defined as $vovl(path_k(v_x,v_y)_{dir})=\lceil |path_k(v_x,v_y)_{dir}|/2 \rceil - |vertices(path_k(v_x,v_y)_{dir})|$. Similarly, the number of edge overlaps is defined as $eovl(path_k(v_x,v_y)_{dir})=\lfloor |path_k(v_x,v_y)_{dir}|/2 \rfloor - |edges(path_k(v_x,v_y)_{dir})|$.

From here, we describe how the vertices and edges of a pattern are mapped to vertices, edges, and paths in a model or a set of models. The way the mapping is done depends on the structure of the pattern's vertices and edges, on their properties, and the global pattern rules. To compare the properties of a pattern to characteristics of a model's vertices and edges (e.g., their captions), we need to specify under which conditions a pair of mapping candidates should be considered equal. Clearly, this depends on the context of application. It may be a simple equality check of a caption's string or a full-fledged check applying complex similarity measures. To keep things general, we define only an equivalence relation on captions (see below). As an example, this equivalence relation could be implemented as a simple equality check on strings. More complicated equivalence relations considering multiple attribute dimensions and based on, for instance, string similarity or linguistic similarity (Delfmann, Herwig, & Lis, 2009) or ontological similarity (Thomas & Fellmann, 2009) measures are easily conceivable.

Definition 5 (mapping): given a query pattern $Q=(V_Q,E_Q,P_V,P_E,\delta,\varepsilon,\Gamma,G)$, one or more models $\bar{M}=(\bar{V},\bar{E},\bar{C},\bar{T}_V,\bar{T}_E,\bar{\alpha},\bar{\beta},\bar{\gamma})$, $\bar{Z}=\bar{V} \cup \bar{E}$, where $\bar{V}=\bigcup_i V_i$, $\bar{E}=\bigcup_i E_i$, $\bar{C}=\bigcup_i C_i$, $\bar{T}_V=\bigcup_i T_{Vi}$, $\bar{T}_E=\bigcup_i T_{Ei}$, $\bar{\alpha}: \bar{V} \to \bar{T}_V$, $\bar{\beta}: \bar{E} \to \bar{T}_E$, $\bar{\gamma}: \bar{Z} \to \bar{C}$, and an equivalence relation $\sim \subseteq \bar{C} \times \bar{C}$, a mapping of Q to $\bar{M}$ creates pattern occurrences $M_i'(Q)=(V_i',E_i',C_i',T_{Vi},T_{Ei},\alpha_i',\beta_i',\chi_i')$, where $V'_i \subseteq \bar{V}$, $E'_i \subseteq \bar{E}$, $C'_i \subseteq \bar{C}$, $Z'_i \subseteq \bar{Z}$, $T_{Vi} \subseteq \bar{T}_V$, $T_{Ei} \subseteq \bar{T}_E$, $\alpha'_i=\bar{\alpha}|_{V'_i}$, $\beta'_i=\bar{\beta}|_{E'_i}$, and $\gamma'_i=\bar{\gamma}|_{C'_i}$, such that:

1. $v_{Qx},v_{Qy} \in V_Q \wedge (v_{Qx},v_{Qy},n_Q) \in E_Q \wedge \varepsilon((v_{Qx},v_{Qy},n_Q)).minl=\varepsilon((v_{Qx},v_{Qy},n_Q)).maxl=1 \wedge dir=\varepsilon((v_{Qx},v_{Qy},n_Q)).dir \Leftrightarrow$
2. $\exists^{=1} v_{ix}' \in V_i'$, $\exists^{=1} v_{iy}' \in V_i'$: $\alpha_i'(v_{ix}') \in \delta(v_{Qx}).vtypes$, $\alpha_i'(v_{iy}') \in \delta(v_{Qy}).vtypes$, $\delta(v_{Qx}).vcaption \sim \chi_i'(v_{ix}')$, $\delta(v_{Qy}).vcaption \sim \chi_i'(v_{ix}') \wedge$
3. $(\exists^{=1}(v_{ix}',v_{iy}',n_M) \in E_i'$: $\varepsilon((v_{Qx},v_{Qy},n_Q)).ecaption \sim \chi_i'((v_{ix}',v_{iy}',n_M))$, $org \in \varepsilon((v_{Qx},v_{Qy},n_Q)).dir$, $\beta_i'((v_{ix}',v_{iy}',n_M)) \in \varepsilon((v_{Qx},v_{Qy},n_Q)).etypesr) \vee$
4. $(\exists^{=1}(v_{iy}',v_{ix}',n_M) \in E_i'$: $\varepsilon((v_{Qx},v_{Qy},n_Q)).ecaption \sim \chi_i'((v_{iy}',v_{ix}',n_M))$, $opp \in \varepsilon((v_{Qx},v_{Qy},n_Q)).dir$, $\beta_i'((v_{iy}',v_{ix}',n_M)) \in \varepsilon((v_{Qx},v_{Qy},n_Q)).etypesr) \vee$
5. $(\exists^{=1}\{v_{ix}',v_{iy}',n_M\} \in E_i'$: $\varepsilon((v_{Qx},v_{Qy},n_Q)).ecaption \sim \chi_i'(\{v_{ix}',v_{iy}',n_M\})$, $none \in \varepsilon((v_{Qx},v_{Qy},n_Q)).dir$, $\beta_i'(\{v_{ix}',v_{iy}',n_M\}) \in \varepsilon(\{v_{Qx},v_{Qy},n_Q\}).etypesr)$,
6. $v_{Qx},v_{Qy} \in V_Q \wedge (v_{Qx},v_{Qy},n_Q) \in E_Q \wedge \varepsilon((v_{Qx},v_{Qy},n_Q)).minl>1$, $\varepsilon((v_{Qx},v_{Qy},n_Q)).maxl \neq 1 \wedge dir=\varepsilon((v_{Qx},v_{Qy},n_Q)).dir \Leftrightarrow$
7. $\exists^{=1} v_{ix}' \in V_i'$, $\exists^{=1} v_{iy}' \in V_i'$: $\alpha_i'(v_{ix}') \in \delta(v_{Qx}).vtypes$, $\alpha_i'(v_{iy}') \in \delta(v_{Qy}).vtypes$, $\delta(v_{Qx}).vcaption \sim \chi_i'(v_{ix}')$, $\delta(v_{Qy}).vcaption \sim \chi_i'(v_{ix}') \wedge$

8. $\exists^{=1}$ $vertices(path_k(v_{xi}',v_{yi}')_{dir}) \in V_i'$, $\exists^{=1}$ $edges(path_k(v_{xi}',v_{yi}')_{dir}) \in E_i'$: $\varepsilon((v_{Qx},v_{Qy},n_Q)).minl \leq length(path_k(v_{xi}',v_{yi}')_{dir}) \leq \varepsilon((v_{Qx},v_{Qy},n_Q)).maxl \wedge$
9. $\varepsilon((v_{Qx},v_{Qy},n_Q)).minvo \leq vovl(path_k(v_{xi}',v_{yi}')_{dir}) \leq \varepsilon((v_{Qx},v_{Qy},n_Q)).maxvo \wedge$ $\varepsilon((v_{Qx},v_{Qy},n_Q)).mineo \leq eovl(path_k(v_{xi}',v_{yi}')_{dir}) \leq \varepsilon((v_{Qx},v_{Qy},n_Q)).maxeo \wedge$
10. $\forall t \in \varepsilon((v_{Qx},v_{Qy},n_Q)).vtypesr$ $\exists v \in vertices(path_k(v_{xi}',v_{yi}')_{dir})$ $|$ $\alpha_i'(v)=t \wedge$ $\nexists\ v \in vertices(path_k(v_{xi}',v_{yi}')_{dir})\ |\ \alpha_i'(v) \in \varepsilon((v_{Qx},v_{Qy},n_Q)).vtypesf \wedge$
11. $\forall t \in \varepsilon((v_{Qx},v_{Qy},n_Q)).etypesr$ $\exists e \in edges(path_k(v_{xi}',v_{yi}')_{dir})$ $|$ $\beta_i'(e)=t \wedge$ $\nexists\ e \in edges(path_k(v_{xi}',v_{yi}')_{dir})\ |\ \beta_i'(v) \in \varepsilon((v_{Qx},v_{Qy},n_Q)).etypesf \wedge$
12. $((vertices(path_k(v_{xi}',v_{yi}')_{dir}) \cup edges(path_k(v_{xi}',v_{yi}')_{dir})) \cap M_p'(\theta.Q)=M_p'(\theta.Q)$ $|$ $\theta.constraint=req \vee$ $(vertices(path_k(v_{xi}',v_{yi}')_{dir}) \cup edges(path_k(v_{xi}',v_{yi}')_{dir})) \cap M_p'(\theta.Q) \neq \emptyset$ $|$ $\theta.constraint=preq \vee$ $(vertices(path_k(v_{xi}',v_{yi}')_{dir}) \cup edges(path_k(v_{xi}',v_{yi}')_{dir})) \cap$ $M_p'(\theta.Q) \neq M_p'(\theta.Q)$ $|$ $\theta.constraint=pforb \vee$ $(vertices(path_k(v_{xi}',v_{yi}')_{dir}) \cup edges(path_k(v_{xi}',v_{yi}')_{dir})) \cap M_p'(\theta.Q)=\emptyset\ |\ \theta.constraint=forb\ \forall \theta \in \varepsilon((v_{Qx},v_{Qy},n_Q)).\Theta)$
13. $\forall i,n_Q,x,y,p,\ M_l' \neq M_m',\ l \neq m$
14. $\forall r \in Q.G.R$: $r=TRUE$
15. $\forall path_k(v_{xi}',v_{yi}')_{dir} \in M_i(Q)$: $G.gminvo \leq |\bigcap_k vertices(path_k\,(v'_{xi},v'_{yi})_{dir})| \leq G.gmaxvo \wedge$ $G.gmineo \leq |\bigcap_k edges(path_k\,(v'_{xi},v'_{yi})_{dir})| \leq G.gmaxeo$

The formal semantics above define how the components of a pattern are mapped to a set of conceptual models. In order to clarify the formalisms, we describe each part by referring to its line number:

1. A pair of pattern vertices that is connected by a pattern edge leading from the first to the second pattern vertex, the length of which has been set to 1, is mapped to a set of pattern occurrences as follows:
2. For both pattern vertices, there exists exactly one counterpart in each pattern occurrence. The vertex type of each of the mapped model vertices must match one of the vertex types assigned to its corresponding pattern vertex. Similarly, its label must be equivalent to the one of the corresponding pattern vertex.
3. For pattern edges as described in (1), there exists exactly one counterpart edge in each pattern occurrence. The direction of the mapped edge of the pattern occurrence is allowed to be the same as the original direction of the pattern edge if the set of allowed directions of the pattern edge contains {org}. Its label must be equivalent to the one of the corresponding pattern edge. Furthermore, its type must be part of the set of allowed edge types defined in the pattern edge.
4. For pattern edges as described in (1), there exists exactly one counterpart edge in each pattern occurrence. The direction of the mapped edge of the pattern occurrence is allowed to be the opposite of the original direction of the pattern edge if the set of allowed directions of the pattern edge contains {opp}. Its label must be equivalent to the one of the corresponding pattern edge. Furthermore, its type must be part of the set of allowed edge types defined in the pattern edge.
5. For pattern edges as described in (1), there exists exactly one counterpart edge in each pattern occurrence. The direction of the mapped edge of the pattern occurrence is allowed to be undirected if the set of allowed directions of the pattern edge contains {none}. Its label must be equivalent to the one of the corresponding pattern edge. Furthermore, its type must be part of the set of allowed edge types defined in the pattern edge.
6. A pair of pattern vertices that is connected by a pattern edge leading from the first to the second pattern vertex, the length of which is greater than 1, is mapped to a set of pattern occurrences as follows:
7. For both pattern vertices, there exists exactly one counterpart in each pattern occurrence. The vertex type of each of the mapped model vertices must match one of the vertex types assigned to its corresponding pattern vertex. Similarly, its label must be equivalent to the one of the corresponding pattern vertex.
8. For pattern edges as described in (6), there exists exactly one counterpart path in each pattern occurrence. The directions of the edges that are part of the path must comply with the direction properties of the corresponding pattern edge. In the formalism, this is expressed by the "dir" subscript of each path expression that refers to the path definition (cf. path definition). Thus, each pattern occurrence contains the vertices and edges that take part in the path. The path must have a length that is between the maximum and minimum length defined in the corresponding pattern edge.

9. Furthermore, the number of its vertex and edge overlaps must be between the maximum and minimum defined in the properties of the corresponding pattern edge.
10. Every vertex type that is defined as mandatory on the path in the corresponding pattern edge properties must occur on the path. Vice versa, none of the vertex types defined as forbidden is allowed to occur on the path.
11. Similarly, every edge type that is defined as mandatory on the path in the corresponding pattern edge properties must occur on the path. Vice versa, none of the edge types defined as forbidden is allowed to occur on the path.
12. This part of the formalism defines how other pattern occurrences of other patterns have to intertwine with a path in order to map it with the pattern edge. In particular, if another pattern occurrence is defined as required on the path, the intersection of the union of the path's edges and vertices with the other pattern occurrence has to equal the pattern occurrence itself. This means that every element of the pattern occurrence is part of the path. If another pattern occurrence is defined as partly required on the path, the intersection of the union of the path's edges and vertices with the other pattern occurrence has to result in a non-empty set. This means that at least one element of the pattern occurrence is part of the path. If another pattern occurrence is defined partly forbidden on the path, the intersection of the union of the path's edges and vertices with the other pattern occurrence must not equal the pattern occurrence itself. This means that at least one element of the pattern occurrence is missing on the path. If another pattern occurrence is defined as forbidden on the path, the intersection of the union of the path's edges and vertices with the other pattern occurrence has to equal the empty set. This means that no element of the pattern occurrence is part of the path. These definitions must be met by all patterns of Θ of the pattern edge.
13. All the definitions above assure that one pattern vertex is always mapped to one vertex of a pattern occurrence. Furthermore, one pattern edge is either mapped to one edge of a pattern occurrence or one path of a pattern occurrence. As mapping a vertex or an edge to a model may generally result in multiple vertex, edge, and path occurrences, the multiple mapping possibilities are saved in different pattern occurrences M_i. Therefore, the formalisms operate for every *i.* Furthermore, because we allow multi-edges, the formalisms also have to regard every n_Q. To identify the different vertices and edges of the pattern, we used the indices *x* and *y*. To map every vertex and every edge of the pattern, the formalisms also operate for every *x* and *y*. The same is true for the index *p*, which we use to denote the different pattern occurrences that may or may not intertwine with paths. Finally, we require that no two occurrences are allowed to be the same to avoid double results.
14. Each global rule out of *G.R* of the pattern has to be true for each of the pattern occurrences.
15. For all paths of a pattern occurrence, the following must be true: (a) the cardinality of the intersection of the vertices of all paths of a pattern occurrence must be greater than or equal to the global minimum vertex overlaps and less than or equal to the global maximum vertex overlaps, and (b) the cardinality of the intersection of the edges of all paths of a pattern occurrence must be greater than or equal to the global minimum edge overlaps and less than or equal to the global maximum edge overlaps. This must be true for all pattern occurrences.

5.4 Matching Algorithm

The matching algorithm takes a query pattern and a set of (possibly integrated) models as input and returns pattern occurrences (see "Semantics" section above). The algorithm adopts ideas from different disciplines of algorithmic graph theory. To match parts of a pattern that do not contain any edges to be mapped to paths, we adopt concepts from the VF2 algorithm for subgraph isomorphism (Cordella et al., 2004). The algorithm successively extends a candidate mapping area of a conceptual model by checking it against structural equality with the pattern. We adapted this algorithm to be able to account for multi-edges of different directions, captions, attributes, and global rules. Furthermore, we extended the algorithm with a path search component that takes into account all the properties specified by PE and the global rules G. The algorithm is a brute-force algorithm because, for model analysis, it is necessary to access all the pattern matches and to know their position in the models. Although both subgraph isomorphism and path search are computationally intractable, our algorithm exhibits a satisfactory runtime behavior (see performance evaluation). The reason for this runtime behavior is that model analysis oftentimes includes specifying particular values for a pattern's properties (e.g., captions, types, etc.), so the search space (i.e., the parts of a model that are actually of interest for an analysis) can be considerably reduced from the beginning. Here, we outline the algorithm through pseudo-code:

```
01:  PRCEDURE MATCH
02:     IF nothing has been mapped yet THEN
03:        Select a random pattern vertex $n_P$ to be mapped
04:        Map $n_P$ to all model vertices $n_M$ matching the properties of $n_P$ and the global rules
05:        Save possible mappings in distinct sets $D$
06:        FOR EACH $d$ IN $D$ CALL MATCH
07:     ELSE
08:        Select a random neighbor pattern vertex $v_P$ of $n_P$ to be mapped next
09:        Get all neighbors $N_P$ of the vertex $v_P$, which are already mapped
10:        FOR EACH $n_P$ IN $N_P$
11:           Get the edges $E_P$ connecting the vertices $n_P$ and $v_P$
12:        END FOR
13:        FOR EACH vertex $n_P$ IN $N_P$
14:           Get the vertex $n_M$ of the model, which was already mapped to $n_P$
15:           IF $length(e_P)=1$ THEN
16:              Get all unmapped neighbors of $U_M$ of $n_M$
17:              FOR EACH vertex $u_M$ IN $U_M$
18:                 IF properties $P_V$ of $v_P$ match $u_M$ AND global rules hold THEN
19:                    Get all edges $E_M$ connecting $n_M$ and $u_M$
20:                    FOR EACH edge $e_M$ IN $E_M$
21:                       IF properties $P_E$ of $e_P$ match $e_M$ THEN
22:                          Save possible combinations in distinct sets $D$
23:                       END IF
24:                    END FOR
25:                 END IF
26:              END FOR
27:           END IF
28:           ELSE IF $length(e_P)\neq 1$ THEN
29:              FOR EACH $u_M$ IN $U_M$
30:                 Perform a depth-first path search from $n_M$ to a vertex $u_M$ such that
31:                    $u_M$ complies with the properties $P_V$ of $v_P$, the path complies with
32:                    the properties $P_E$ of $e_P$, and global rules hold
33:                 Save possible combinations in distinct sets $D$
34:              END FOR
35:           END IF
36:        END FOR
37:        IF NOT whole pattern already mapped THEN
38:           FOR EACH $d$ IN $D$ CALL MATCH
39:        END IF
40:        RETURN $D$
41:     END IF
42:  END MATCH
```

The algorithm consists of a single procedure *Match* that starts the mapping with a single vertex and adds further neighbored vertices of the pattern successively. Each possible mapping is saved as an intermediate result. The procedure then calls itself for every possible mapping and extends them in the same manner. Non-feasible mappings are discarded. Hence, at the end, all possible mappings (i.e., all pattern occurrences) are returned.

The first step of the algorithm is to take a random vertex of the pattern and map it to all possible vertices of the models; that is, all vertices that comply with the pattern vertex' properties. These are saved as intermediate results. For each of the intermediate results, *Match* is called recursively (cf. Lines 1-6). Next, the algorithm gets a random pattern vertex that is adjacent to the last one (cf. Line 8). Then, all neighbors of that vertex that are already mapped are determined (cf. Line 9). This can be multiple vertices if the new vertex to be mapped has multiple connections to the already mapped area of the pattern. For all of the already mapped vertices, the algorithm then determines the edges that connect these vertices to the new vertex to be mapped (cf. Lines 10-12). For each of the already mapped vertices, the algorithm then performs the following actions: first, it gets the corresponding vertex of the model that was mapped to the pattern vertex (cf. Line 14). If the edge connecting the already mapped pattern vertex and the new pattern vertex to be mapped is of length one (cf. Line 15), then the algorithm gets all neighbors of the already mapped vertex in the model (cf. Line 16). Each such vertex is checked against the properties of the new pattern vertex to be mapped (cf. Line 17-18). If the properties are met, then the edges between the already mapped vertex of the model and the new ones are checked against the properties of their counterparts in the pattern. If the properties are met, the algorithm saves each mapping as an intermediate result (cf. Lines 19-27).

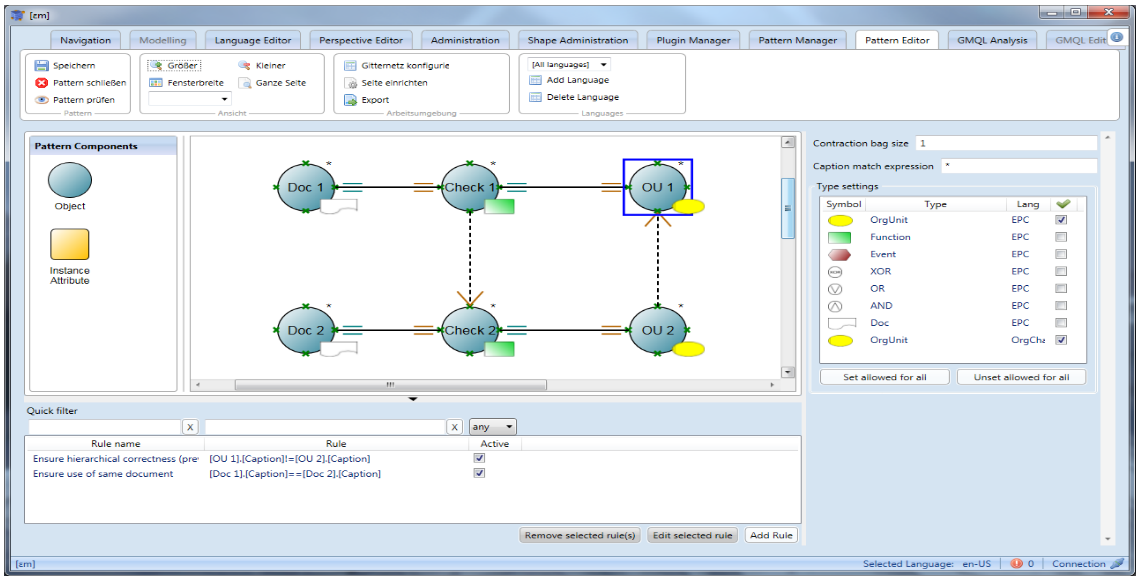
If the edge connecting the already mapped pattern vertex and the new pattern vertex to be mapped is of a length greater than one (cf. Line 28), then the algorithm performs a depth-first path search from the model vertex already mapped to a not yet mapped vertex that has to comply with the properties of the new pattern vertex to be mapped. During the path search, the algorithm checks all properties of the corresponding pattern edge (cf. Lines 30-32). As soon as one property is violated, the path search is stopped and the result is discarded. If the path search is successful, the path is added to the mapping and saved as an intermediate result (cf. Line 33). The intermediate results are then used to call *Match* again (cf. Lines 37.39). In the new recursion, the intermediate results are extended by adding new mapping candidates. If all vertices and edges of the pattern are mapped, the execution stops, and the algorithm returns all pattern occurrences.

6 Implementation

To demonstrate the applicability of DMQL and its matching algorithm, we implemented both as a plug-in for a modeling tool that was available from a former research project (Delfmann et al., 2015). The tool is a meta modeling tool; hence, we can specify modeling languages by mouse-click at runtime of the tool and without generating a new modeling tool for each new language. The tool is based on a similar concept as outlined in Figure 2. As such, we could easily implement DMQL and its matching algorithm as a plug-in. By accessing the language specifications of the modeling tool, the query language can load both the concepts of all implemented modeling languages (i.e., language, vertex type, and edge type; cf. Figure 2) and their visualizations and reuse them for pattern specification. For query execution, the matching algorithm can, in turn, access the model database of the tool and reuse the model data for the matching process. As the meta modeling tool is capable of defining integrated modeling languages and developing integrated models as described in Section 2, one can easily perform the preprocessing step of the matching algorithm (i.e., unifying the models to be searched through their shared vertices and edges) as well.

6.1 Query Pattern Editor

Figure 10 shows the pattern editor of the DMQL plug-in. It realizes the abstract and concrete syntax of the query language as introduced in Section 5.



Contraction bag size: 1

Caption match expression: *

Type settings

Symbol	Type	Lang	✓
●	OrgUnit	EPC	✓
🟩	Function	EPC	☐
⬢	Event	EPC	☐
⊗	XOR	EPC	☐
∨	OR	EPC	☐
∧	AND	EPC	☐
▭	Doc	EPC	☐
●	OrgUnit	OrgCha	✓

Quick filter: any

Rule name	Rule	Active
Ensure hierarchical correctness (pre	[OU 1].[Caption]!=[OU 2].[Caption]	✓
Ensure use of same document	[Doc 1].[Caption]==[Doc 2].[Caption]	✓

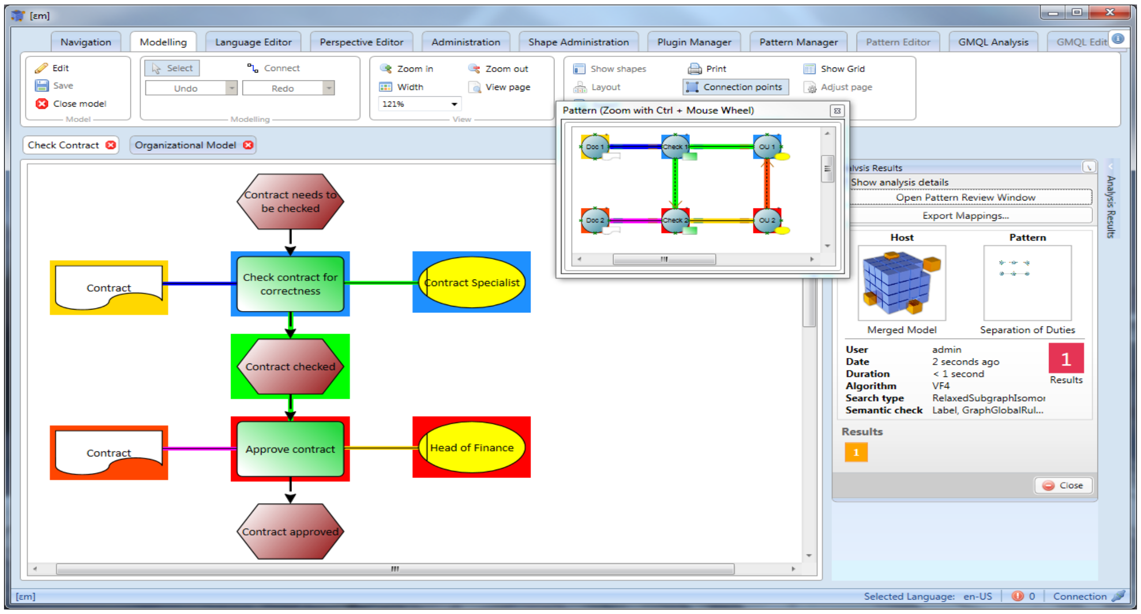
Figure 10. Specifying a Query Pattern

The pattern implements the compliance rule separation of duties (see Section 1) and combines EPCs and organizational charts. It comprises two vertices assigned a single allowed vertex type function of EPCs. These vertices are connected through a directed path of unlimited length with no path overlaps. Each function vertex is connected to another vertex of the type organizational unit by an undirected edge of length one. Between the organizational units, we define a directed path of unlimited length with no path overlaps that starts at the second organizational unit and ends at the first one. The allowed edge types on this path are supervisor edges. Hence, this path expresses that the second organizational unit is superordinate to the first one. In addition, both functions operate on a document. Further restrictions are specified in the global rules (at the bottom of the Figure). The first rule requires that the organizational units must not to the same (i.e., *[OU1].[Caption]!=[OU2].[Caption]*), and the second one requires that both functions have to operate on the same document (i.e., *[Doc1].[Caption]==[Doc2].[Caption]*).

Note that, in this pattern, the used vertex types function and document belong to only the EPC language, the organizational unit vertex type belongs to both, and the supervisor edge type belongs to only the organizational chart language.

6.2 Pattern Matching

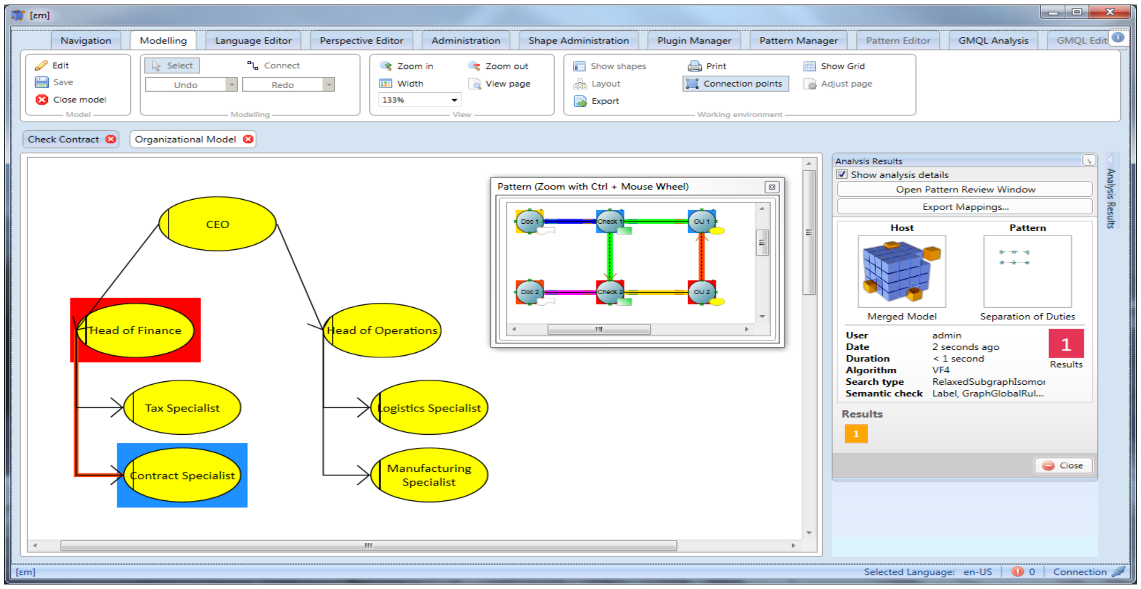
To apply this pattern to a set of models, a user has to select models to be examined from a list of existing ones. As a matter of course, only those models are provided that comply with the languages the pattern spans. The selected models are unified as described in Section 2 and in the semantics part of Section 5, and the matching algorithm is run onto the integrated models. The algorithm returns all matches by sets of mapped vertices and edges. As the algorithm is aware of the modeling tool-internal IDs of the vertices and edges, one can highlight and browse the each exact position of each match in the models.



User	admin
Date	2 seconds ago
Duration	< 1 second
Algorithm	VF4
Search type	RelaxedSubgraphIsomor
Semantic check	Label, GraphGlobalRul...

Figure 11. Matching Result (Process Model Part)

Figures 11 and 12 illustrate one particular match of the pattern we describe above. The tool highlights every match found. The match of the example spreads over a process model and an organizational chart. The user can switch between both models. One can open the pattern as it was modeled as a colored miniature graphic. The pattern match is colored in the same way to highlight which pattern vertices and edges were mapped to which model vertices, edges, and paths. For instance, the path edge between the function vertices of the pattern is colored green, as is the mapped path in the model. In the organizational chart, we see that the organizational units and the supervisor path edge of the pattern are mapped accordingly.



User admin
Date 2 seconds ago
Duration < 1 second
Algorithm VF4
Search type RelaxedSubgraphIsomor
Semantic check Label, GraphGlobalRul...
1 Results

Figure 12. Matching Result (Organizational Chart Part)

7 Performance Evaluation

To evaluate the performance of DMQL, we present a runtime evaluation as Georges, Buytaert, and Eeckhout (2007) propose. To that end, we introduce the patterns we queried, the model base we conducted our experiments on, technical details on the method of gathering runtime data and on the PC we used, and the final runtime measurements.

In total, we queried fourteen patterns, seven of which are EPC patterns and seven of which are patterns for ERMs. None of these patterns are modeling view-spanning; however, this does not affect performance. The way DMQL's matching algorithm searches for pattern occurrences is the same for models of single languages and for integrated ones because integrated models are unified before the search starts. All patterns can either be found in the literature or represent structures that, to our knowledge, frequently occur in conceptual models. Because the number of pattern matches influences runtime performance, one needs to search for patterns that are frequently contained in the models to obtain a conservative impression of runtime performance. Furthermore, to challenge the performance even more, we abstained from specifying too restrictive vertex and edge properties (especially captions) because they may considerably speed up the matching process.

The EPC patterns are based on Mendling (2007), who proposes a set of structural and behavioral conflicts in EPC models. We searched for AND control structures (AND) and for XOR splits that are not correctly joined (XOR split incorrectly; cf. Figure 4). Furthermore, we searched for the syntactically correct version of the latter structure, which means an XOR split that is indeed correctly joined (XOR split correctly). We searched for a decision split after an event, which comprises an event that is followed by an XOR split (DSAE), which is a syntax error in EPCs. In addition, we searched for the syntactically correct version of this pattern; namely, a decision split after a function (DSAF). Furthermore, we searched for two single-entry-single-exit (SESE) structures with XOR connectors as splitting and joining nodes. One SESE structure contains paths on all branches between the entry and exit nodes (SESE XOR), whereas, in the other structure, the entry and exit nodes are directly adjacent on one branch (XORS-XORJ).

The ERM patterns contain structures that frequently occur in the entire ER test models. We searched for all paths between an entity type and a relationship type (ER paths) and for all paths between entity types (EE paths). In addition, we searched for binary (BR) and ternary relationship types (TR), which means that a relationship type that is adjacent to either two or three entity types. We also searched for all paths between two entity types in which the start node is adjacent to a binary relationship type (BR + E paths). We furthermore searched for entity type loops (E loops), meaning paths that start and end in the same entity type, and hierarchy structures.

As a model base, we chose a repository of ERMs and EPCs that was available from a consulting project in trade companies. In total, the repository contained 53 EPCs and 42 ERMs. The largest EPC contained 264 elements and represented a material requirements planning process. The smallest EPC contained 15 elements and represented a deviation control process. The largest ERM contained 97 elements and represented an article data model. The smallest ERM contained 16 elements and represented an excerpt of a sales data model. We chose this model repository for two reasons. First, these process and data models were developed in a real-life consulting project. Consequently, they allow us to test DMQL in a realistic context. Second, the repository explicitly includes very small models that have a minimum of 15 elements and large models with up to 264 elements. This range of different model sizes allows us to evaluate the scaling characteristics of the matching algorithm because the size of the model is one factor (among others, such as pattern size) that influences runtime performance. Additionally, we argue that most conceptual models exhibit model sizes that fall into the range we cover here. Therefore, we conclude that the results we obtained for this repository are also comparable to other model collections.

To obtain the performance data in a statistically rigorous manner, we applied the steady-state performance method that (Georges et al. (2007) introduced. They developed the method to avoid measurement errors in Java programs caused by the non-deterministic execution in the virtual machine due to garbage collection, just-in-time (JIT) compilation and adaptive optimization. We implemented the matching algorithm as a plugin for a meta-modeling tool written in C#. This programming language suffers from the same measurement errors. Therefore, we adapted Georges et al.'s (2007) approach to C#. To compensate for the non-deterministic behavior of the virtual machine, we chose the so-called steady-state performance measurement determining program performance after JIT compilation and optimization. The measurement procedure comprised two phases. During a warm-up phase, the JIT compiler optimized the pattern search algorithm. This meant running the algorithm five times to reduce the probability of getting highly dispersed runtime measurements (Georges et al., 2007). Once this warm-up phase was completed, the measurement phase started. During this phase, we searched each pattern in each model ten times. We determined the mean execution times for each combination of pattern query and model.

We conducted the performance evaluation on an Intel® Core™ 2 Duo CPU E8400 3.0 GHZ with 4 GB RAM and Windows 7 (64-Bit edition). We disabled the energy saving settings in Windows and executed the application as a 32-bit real-time process to avoid any unnecessary hardware slow down or process switching. Prior to each search run, we forced a complete garbage collection. In doing so, we further eliminated systematic errors. For the time measurement, we used the high resolution QueryPerformanceCounter-API in Windows in concert with the adapted method.

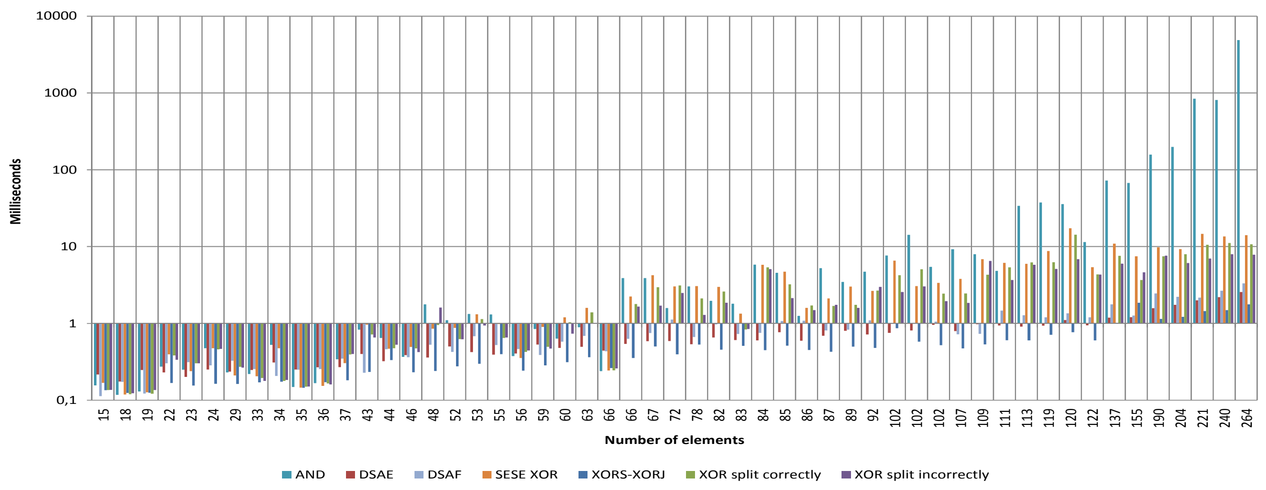


Figure 13. Runtime Measurements for EPCs

Figures 13 and 14 contain runtime measurements for EPCs (Figure 13) and ERMs (Figure 14). Each model is represented on the abscissa by its number of elements. The ordinate represents time measurements in milliseconds on a logarithmic scale. Each bar represents the mean execution time of one pattern in one model (possibly returning multiple occurrences) as described above. Figure 13 suggests that runtime performance of our algorithm can be considered acceptable. In 350 out of 371 (53 models times 7 patterns) cases, pattern matches could be identified within 10 milliseconds. In only 5 cases could we observe runtimes larger than 100 milliseconds. The maximum runtime measured for EPC models amounts to 4873 milliseconds. As one could expect, the figure demonstrates that the size of the model influences runtime performance. On rather small models with a size of up to about 60 elements, pattern matches could be identified within a fraction of a millisecond. Larger runtimes, in contrast, are only observed on larger models. The maximum runtime occurred in the largest model of the collection exhibiting a size of 264 elements.

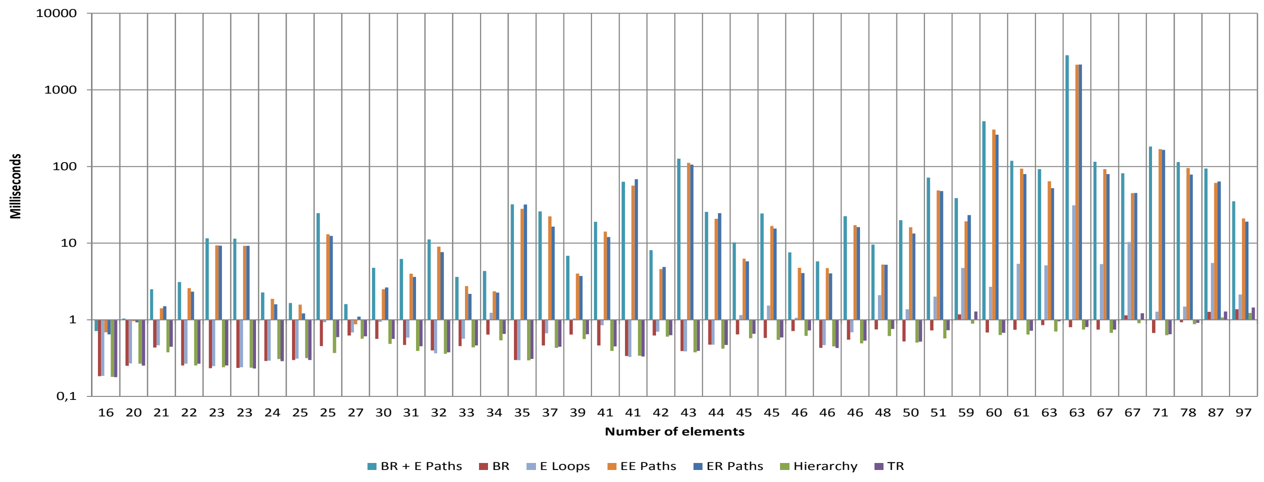


Figure 14. Runtime Measurements for ERMs

Figure 14 contains runtime measurements for ERMs. Compared to EPCs, runtimes were greater. In 279 out of 294 cases (42 models times 7 patterns), matches could be identified within 100 milliseconds. In three cases, we observed runtimes larger than one second. The maximum runtime for ERMs amounts to 2821 milliseconds. We can explain the fact that matching takes longer on ERMs on average with the edge direction type. Because ERMs are undirected, the algorithm has to consider a greater number of possible “ways to go”. EPCs, on the other hand, are directed, which allows the algorithm to follow the edge direction when calculating matches. However, runtime performance on ERMs can still be considered acceptable. In addition, the runtime measurements for ERMs also confirm the hypothesis that model size influences runtime performance, albeit not as clearly, because we did not observe the largest runtimes in the largest model.

In addition to runtime measurements, we also recorded the number of times each pattern was found in each model. Relating runtime performance to the number of pattern matches, one can see that large runtimes occurred whenever large amounts of pattern matches were recorded. Consider, for example, the case of ERMs. In three cases, runtimes larger than one second occurred. It took the algorithm 2821 milliseconds to determine paths between entity types that are directly adjacent to a binary relationship type (BR + E paths) in a model comprising 63 elements. This measurement corresponds to 4747 pattern matches. It took the algorithm 2130 milliseconds to determine all paths starting and ending in entity types and 2144 milliseconds to determine all paths starting in an entity type and ending in a relationship type. These measurements correspond to 22856 and 16875 pattern matches, respectively. The largest runtime measurement obtained in the EPC models corresponds to 377 pattern matches. Otherwise, if the matching procedure finished quickly, only a few matches could be found. We can explain this finding with the fact that the matching rules incorporated in the algorithm act as pruning techniques that allow the matching procedure to finish quickly if a candidate pair does not meet the required structural or semantic constraints. If, however, the matching rules do hold and a match is found, the algorithm has to run until all pattern nodes are correctly mapped to a subset of all model nodes. Furthermore, in case a match is found, it is copied to the result set, which requires additional “administrative” overhead. Runtimes consequently increase with the number of matches found.

8 Contributions and Challenges Met

In this paper, we introduce a diagramed query language for conceptual models that is applicable to graph-like models of any type or modeling language and even to integrated models of different views. It supports both mappings in the sense of subgraph isomorphism without any structural restriction and in the sense of subgraph homeomorphism through supporting configurable paths. Our work furthermore supports multiple business objectives of analyzing models because we do not allow only predefined queries. As the definition of the abstract and concrete syntax and the semantics of GMQL shows, we were able to meet all of the requirements specified at the beginning.

The runtime experiments we conducted demonstrate that DMQL returns matches with acceptable runtime performance. As such, it is suitable to be used on large collections of conceptual models that many companies have started to develop. Hence, we can summarize the paper's contributions as follows::

- DMQL includes a matching algorithm that is based on a new kind of graph search problem called relaxed subgraph isomorphism, which is particularly related to analyzing conceptual models. Therefore, we contribute to algorithmic graph theory by introducing a new class of graph search problem. With our work, we expect to trigger further research on this problem from a graph theoretical perspective. For instance, novel efficiency-enhanced algorithms would contribute both to graph theory and, in turn, to conceptual model analysis.
- No multi-purpose and language-independent model query languages that allow pattern occurrence structures of any complexity (and, hence, that meet all our requirements) have existed up to now. With our work, we contribute to a variety of different model analysis objectives requiring pattern matching. Furthermore, our approach can be used on models of any graph-based modeling language. Therefore, we expect wide-spread application because model analysis and model analysts are no longer restricted to particular modeling languages or few analysis objectives.
- As we demonstrate with our performance evaluation, our matching algorithm exhibits satisfactory runtime performance. Thus, we expect it to be suitable for application in real-world industry scenarios.
- Another important characteristic of DMQL is its diagramed way of specifying query patterns, which eases the usage of the query language.

To summarize, we provide a novel query language that meets all the requirements placed in Section 3. Up to now, no query language meeting all these requirements existed. To meet all the requirements in combination, we had to deal with some challenges that existing query languages thus far have not or only partially addressed: by analyzing related work, we show that most query languages meet R1-R4, that few meet R5, R6, and R10, and that virtually none meet R7-R9. Most query languages may support R1-R4 because they are the most obvious. We cannot find structures in conceptual models without considering vertices, edges, and their connections. Furthermore, there are well-known concepts coming from the areas of graph search and temporal logic that can be reused to realize R1-R4 (for instance, subgraph isomorphism (Cordella et al., 2004), subgraph homeomorphism (Lingas & Wahlen 2009), path search (Birmelé et al., 2013), Linear Temporal Logic (LTL) (Pnueli, 1977), and computation tree logic (CTL) (Clarke, Emerson, & Sistla, 1986)). In DMQL, we realized R1-R4 through a novel relaxed subgraph isomorphism algorithm (see "Semantics" section (Definition 5) and "Algorithm" section in Section 3).

To realize R5, related query languages transform process models into state machines (e.g., *PPSL*). These comprise either multiple linear graphs or a tree visualizing any possible execution of a process model, even when a loop in a process is executed multiple times. We could not use state machines because these would restrict our query language to process models. Hence, we meet R5 through introducing a modified depth-first search algorithm that uses count variables to remember vertices and edges already visited (cf. "Semantics" section in Section 3, Definition 5, line 9). We realized R7 in the same way (cf. "Semantics" section in Section 3, Definition 5, line 15). To realize R6, we had to introduce a mechanism that remembers properties of the visited vertices and edges in order to make their comparison possible (cf. "Abstract Syntax" section in Section 3; "Semantics" section in Section 3, Definition 5, line 14).

Many related query languages do not support R8-R9 because they were intentionally designed to support a specific modeling language. In most of the corresponding papers, the applicability to different modeling languages or even different modeling views (such as processes, data, organization) is not even discussed. Restricting a query language to one specific modeling language makes it easier to specify query components because the available language concepts and the syntax are known beforehand. Both can be exploited. For instance, many query languages are restricted to process models. All of them exploit the fact that control flows of process models are always directed edges. Hence, the challenge of meeting R8-R9 was to find an appropriate level of abstraction of the query language considering the commonalities of different modeling languages. We did this by using a very general definition of conceptual models that considers specificities of special modeling languages as variables rather than as constants (cf. Definitions 1-2).

The challenge to meet R10 was to align the formal concepts describing and executing a query with graphical representations. For some of the related query languages, a one-by-one representation of query concepts through graphical elements is not possible or at least not very helpful. For instance, *GMQL* uses complex set transformations to assemble pattern matches successively, such as ”find all activities and subtract all activities that are followed by a split connector from them”. Representing such a query graphically does not necessarily increase the readability or understandability of a query because the query does not “look like” the structures in the models that should be found. The same applies for approaches based on temporal logic and/or state machines (such as PPSL, aFSA, CompliancePatterns). Hence, in such cases, a sophisticated transformation mechanism of the underlying formal search concepts into intuitive graphical representations is necessary (as it is done, e.g., by Elgammal et al. (2014)). Thus, we use graph search as formal basis. We can easily turn our queries into a graphical representation, which we argue is intuitive as the queries look much like the model structures to be searched.

Finally, we realized the greatest challenge—creating a query language meeting *all* of R1-R10—by using a generalized definition of conceptual models that regards them as attributed, both directed and undirected graphs that comply with an arbitrary modeling language and, furthermore, by making use of the novel concept of relaxed subgraph isomorphism. Only through using these concepts could we provide operations realizing queries and query executions that meet R1-R10.

9 Limitations and Outlook

However, our work reveals some limitations, which result from its broad applicability:

- Although DMQL is applicable to different analysis purposes in principle, it does not come with any predefined query patterns. Hence, it is a pure construct and does not contain any business domain knowledge. Defining queries for a particular application scenario is the analyst’s responsibility. Furthermore, a query pattern has to be defined in accordance with the modeling languages of the models to be analyzed. Naturally, although searching for pure ERM patterns in, for instance, pure EPC models is possible using DMQL, it will not return any results.
- Another restriction of DMQL also results from its applicability to multiple modeling languages. As a general applicability does include not only process models but also data models, organizational charts, and so on, our approach is, by design, not able to analyze special characteristics of special kinds of models, especially execution semantics of process models. However, corresponding with R5, we included mechanisms to at least approximately analyze execution semantics.
- Our approach has not yet been subject to a comprehensive utility evaluation. Despite this, we expect analysts to appreciate it because related work has already proven to be highly relevant for model analysis purposes (Becker et al., 2015, Becker et al., 2011). Moreover, due to its satisfactory execution speed and the possibility to specify patterns graphically, we expect an even higher utility of DMQL. Comprehensive utility evaluations are subject of short-term research. In particular, we plan to apply our work in financial institutions to check business process models for regulatory compliance violations and weaknesses. Furthermore, we plan to apply DMQL to business process weakness detection. This may, furthermore, carve out additional functional requirements that have to be included in the query language.
- We are aware that choosing the right terms for labels to be searched for is difficult if an analyst does not know the entire set of terms used in the models. Hence, DMQL works only if its user has a deep knowledge of the natural language concepts used in the models to be analyzed or the user is supported by a corresponding natural language processing approach (note that this problem is always present when models should be analyzed regardless of the analysis approach). Natural language processing approaches either standardize the use of terms already during modeling (e.g., through enforcing naming conventions automatically (Delfmann et al., 2009)), or they connect model elements to ontology concepts containing a distinct definition of the element’s meaning (Thomas & Fellmann, 2009). As we outline in the “Semantics” section in Section 3, we provide a generic equivalence relation to compare label semantics. Hence, DMQL does not provide an own label processing approach but accounts for

docking existing ones. Once appropriate label processing is established, the user could be supported by label suggestions given automatically during pattern specification.

Medium-term research will focus on performing additional runtime experiments on extremely large models with the aim of further increasing execution speed and applying the algorithm to further model analysis scenarios in practice. At the moment, we investigate graph-theoretical structural characteristics of conceptual models that can speed up pattern matching. In particular, we expect bounded treewidth and planarity of conceptual model graphs to be very promising characteristics to increase matching performance significantly.

Acknowledgments

DMQL was partly developed in the course of the research project “Business Process Compliance Management in the Financial Sector”, which was funded by the German Research Association from September 2012 to August 2014 (reference number: DE1983/1-1).

References

Awad, A. (2007). BPMN-Q: A language to query business processes. In M. Reichert S. Strecker, & K. Turowski (Eds.), *Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures* (pp. 115-128).

Batra, D. (2005). Conceptual data modeling patterns: representation and validation. *Journal of Database Management, 16*(2), 84-106.

Becker, J., Bergener, P., Räckers, M., Weiß, B., & Winkelmann, A. (2010). Pattern-based semi-automatic analysis of weaknesses in semantic business process models in the banking sector. In *Proceedings of the European Conference on Information Systems.*

Becker, J., Bergener, P., Delfmann, P., & Weiß, B. (2011). Modeling and checking business process compliance rules in the financial sector. In D. F. Galletta & T.-P. Liang (Eds.), *Proceedings of the International Conference on Information Systems.*

Becker, J., Delfmann, P., Dietrich, H.-A., Eggert, M., & Steinhorst, M. (2015). Model-based business process compliance checking in financial industries—conceptualization, implementation, and evaluation. *Information Systems Frontiers*.

Becker, J., Delfmann, P., Eggert, M., & Schwittay, S. (2012). Generalizability and applicability of model-based business process compliance-checking approaches—a state-of-the-art analysis and research roadmap. *Business Research*, *5*(2), 221-247.

Beeri, C., Eyal, A., Kamenkovich, S., & Milo, T. (2008). Querying business processes with BP-QL. *Information Systems*, *33*(6), 477-507.

Bergener, P., Delfmann, P., Weiss, B., & Winkelmann, A. (2015). Detecting potential weaknesses in business processes—an exploration of semantic pattern matching in process models. *Business Process Management Journal*, *21*(1), 25-54.

Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., & Ökrös, A. (2010). Incremental evaluation of model queries over EMF models. In *Proceedings of the 13th International Conference MODELS* (pp. 76-90).

Birmelé, E. R., Ferreira, R., Grossi, A., Marino, N., Pisanti, R., Rizzi, G., & Sacomoto, G. (2013). Optimal listing of cycles and St-paths in undirected graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 1884-1896).

Breuker, D., Delfmann, P., Dietrich, H.-A., & Steinhorst, M. (2014). Graph theory and model collection management—conceptual framework and runtime analysis of selected graph algorithms. *Information Systems and e-Business Management*, *13*(1), 69-106.

Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, *1*(1), 9-36.

Choi, I., Kim, K., & Jang, M. (2007). An XML-based process repository and process query language for integrated process management. *Knowledge and Process Management*, *14*(4), 303-316.

Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems, 8*(2), 244-263.

Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (2004). A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *26*(10), 1367-1372.

Davies, I., Green, P., Rosemann, M., Indulska, M., & Gallo, S. (2006). How do practitioners use conceptual modeling in practice? *Data & Knowledge Engineering*, *58*(3), 358-380.

Delfmann, P., Steinhorst, M., Dietrich, H.-A., & Becker, J. (2015). The generic model query language GMQL—conceptual specification, implementation, and runtime evaluation. *Information Systems*, *47*(1), 129-177.

Delfmann, P., Herwig, S., Lis, L. (2009). Unified enterprise knowledge representation with conceptual models—capturing corporate language in naming conventions. In *Proceedings of the 30th International Conference on Information Systems.*

Desel, J., & Juhás, G. (2001). What is a Petri net? Informal answers for the informed reader. In H. Ehrig, J. Padberg, G. Juhás, & G. Rozenberg (Eds.), *Unifying Petri nets* (pp. 1-25). Berlin, Germany: Springer.

Di Francescomarino, C., & Tonella, P. (2009). Crosscutting concern documentation by visual query of business processes. In D. Ardagna, M. Mecella, & J. Yang (Eds.), *Business process management workshops, LNBIP 17* (pp. 18-31). Berlin, Germany: Springer.

Diestel, R. (2010). *Graph theory* (4th ed.). Berlin: Springer.

Dijkman, R. M., Dumas, M., van Dongen, B., Käärik, R., & Mendling, J. (2011). Similarity of business process models: Metrics and evaluation. *Information Systems*, *36*(2), 498-516.

Dijkman, R. M., La Rosa, M., & Reijers, H. A. (2012). Managing large collections of business process models—current techniques and challenges. *Computers in Industry*, *63*(2), 91-97.

Duddy, K., Gerber, A., Lawley, M., Raymond, K., & Steel, J. (2003). Model transformation: A declarative, reusable pattern approach. In *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference* (pp. 174-185). IEEE Computing Society.

Elgammal, A., Turetken, O., van den Heuvel, W., & Papazoglou, M. (2014). Formalizing and applying compliance patterns for business process compliance. *Software and Systems Modelling, 1-28.*

Elgammal, A., Turetken, O., van den Heuvel, W.-J., & Papazoglou, M. (2010). Root-cause analysis of design-time compliance violations on the basis of property patterns. In P. Maglio, M. Weske, J. Yang, & M. Fantinato (Eds.), *Service-oriented computing* (pp. 17-31). Berlin, Germany: Springer.

Fahland, D., Jobstmann, B., Koehler, J., Lohmann, N., Hagen, V., & Wolf, K. (2009). Instantaneous soundness checking of industrial business process models. In U. Dayal, J. Eder, J. Koehler, & H. A. Reijers (Eds.), *Proceedings of the 7th International Conference on Business Process Management* (pp. 278-293). Berlin, Germany: Springer.

Ferro, A., Giugno, R., Mongiovi, M., Pulvirenti, A., Skripin, D., & Shasha, D. (2007). GraphBlast: Multi-feature graphs database searching. In *Proceedings of the Workshop on Network Tools and Applications in Biology*.

Förster, A., Engels, G., Schattkowsky, T., van der Straeten, R. (2007). Verification of business process quality constraints based on visual process patterns. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering* (pp.197-208).

García-Bañuelos, L. (2008). Pattern identification and classification in the translation from BPMN to BPEL. In R. Meersman & Z. Tari (Eds.), *Proceedings of the International Conference On the Move to Meaningful Internet Systems* (pp. 436-444). Berlin, Germany: Springer.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness.* New York, NY: W. H. Freeman and Company.

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications* (pp. 57-76). New York, NY: ACM.

Haskell, A. C., & Breaznell, J. G. (1922). *Graphic charts in business: How to make and use them*. New York, NY: Codex Book Company.

Jin, T., Wang, J., Wu, N., La Rosa, M., & ter Hofstede, A. H. M. (2010). Efficient and accurate retrieval of business process models through indexing. In *Proceedings of the 2010 International Conference On the Move to Meaningful Internet Systems* (pp. 402-409).

Karimi, J. (1988). Strategic planning for information systems: Requirements and information engineering methods. *Journal of Management Information Systems*, *4*(4), 5-24.

Keller, G., & Teufel, T. (1998). *Sap R/3 process oriented implementation*. Boston, MA: Addison-Wesley.

Knuplesch, D., Ly, L., Rinderle-Ma, S., Pfeifer, H., & Dadam, P. (2010). On enabling data-aware compliance checking of business process models. In J. Parsons, M. Saeki, P. Shoval, C. Woo, & Y. Wand (Eds.), *Proceedings of the 29th International Conference on Conceptual Modeling* (pp. 332-346). Berlin, Germany: Springer.

Kottemann, J. E., & Konsynski, B. R. (1984). Information systems planning and development: Strategic Postures and methodologies. *Journal of Management Information Systems*, *1*(2), 45-63.

La Rosa, M., Reijers, H. A., van der Aalst, W. M. P., Dijkman, R. M., Mendling, J., Dumas, M., & García-Bañuelos, L. (2011). APROMORE: An advanced process model repository. *Expert Systems with Applications*, *38*(6), 7029-7040.

Lingas, A., & Wahlen, M. (2009). An exact algorithm for subgraph homeomorphism. *Journal of Discrete Algorithms*, *7*(4), 464-468.

Ly, L. T., Rinderle-Ma, S., Göser, K., & Dadam, P. (2012). On enabling integrated process compliance with semantic constraints in process management systems. *Information Systems Frontiers*, *14*(2), 195-219.

Mahleko, B., & Wombacher, A. (2006). Indexing business processes based on annotated finite state automata. In *Proceedings of the 2006 IEEE International Conference on Web Services* (pp. 303-311).

Mendling, J. (2007). *Detection and prediction of errors in EPC business process models*. Vienna, Austria: University of Economics and Business Administration.

Mendling, J., Verbeek, H. M. W., van Dongen, B. F., van der Aalst, W. M. P., & Neumann, G. (2008). Detection and prediction of errors in EPCs of the SAP reference model. *Data & Knowledge Engineering*, *64*(1), 312-329.

Momotko, M., & Subieta, K. (2004). Process query language: A way to make workflow processes more flexible. In *Proceedings of the 8th East European Conference on Advances in Databases and Information Systems* (pp. 306-321).

Neo4j. (2014). Cyper query language. Retrieved from http://docs.neo4j.org/chunked/stable/cypher-query-lang.html

OASIS. (2007). Web services business process execution language version 2.0. Retrieved from http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

Object Management Group. (2014a). BPMN 2.0 Specification. Retrieved from http://www.omg.org/spec/BPMN/2.0/PDF

Object Management Group. (2014b). *Unified modeling language 2.4.1 specification*. Retrieved from http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF

Object Management Group. (2014c). *Object constraint language*. Retrieved from http://www.omg.org/spec/OCL/2.4

Ouyang, C., Dumas, M., ter Hofstede, A. H. M., & van der Aalst, W. M. P. (2008). Pattern-based translation of BPMN process models to BPEL web services. *International Journal of Web Services Research*, *5*(1), 42-62.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45-77.

Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (pp. 46-57).

Polyvyanyy, A., Weidlich, M., & Weske, M. (2012). Isotactics as a foundation for alignment and abstraction of behavioral models. In A. Barros, A. Gal, & E. Kindler (Eds.), *Proceedings of the 7th International Conference on Business Process Management* (pp. 335-351). Berlin, Germany: Springer.

Polyvyanyy, A., Smirnov, S., & Weske, M. (2010). Business process model abstraction. In J. vom Brocke & M. Rosemann (Eds.), *Handbook on business process management 1* (pp. 149-166). Berlin, Germany: Springer.

Reijers, H. A., Mendling, J., & Dijkman, R. M. (2011). Human and automatic modularizations of process models to enhance their comprehension. *Information Systems*, *36*(5), 881-897.

Rosemann, M. (2006). Potential pitfalls of process modeling: Part A. *Business Process Management Journal*, *12*(2), 249-254.

Scheer, A.-W. (2000). *ARIS—business process modeling* (3rd ed.). Berlin, Germany: Springer.

Scheidegger, C. E., Vo, H. T., Koop, D., Freire, J., Silva, C. T., & Silva, T. (2008). Querying and re-using workflows with VisTrails. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (pp. 1251-1254).

Shao, Q., Sun, P., & Chen, Y. (2009). WISE: A workflow information search engine. In *Proceedings of the 25th International Conference on Data Engineering* (pp. 1491-1494).

Soffer, P. (2005). Refinement equivalence in model-based reuse. *Journal of Database Management*, *16*(3), 21-39.

Song, L., Jianmin, W., ter Hofstede, A. H. M., La Rosa, M., Ouyang, C., & Wen, L. (2011). *A semantics-based approach to querying process model repositories* (Technical Report). Brisbane, Australia: School for Information Systems of the Queensland University of Technology.

Störrle, H. (2011). VMQL: A visual language for ad-hoc model querying. *Journal of Visual Languages & Computing*, *22*(1), 3-29.

Störrle, H., & Acretoaie, V. (2013). Querying business process models with VMQL. In *Proceedings of the 5th ACM SIGCHI Annual International Workshop on Behaviour Modelling—Foundations and Applications* (pp. 1-10).

Thomas, O., & Fellmann, M. (2009). Semantic process modeling—design and implementation of an ontology-based representation of business processes. *Business & Information Systems Engineering*, *1*(6), 438-451.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM*, *23*(1), 31-42.

van der Aalst, W. M. P. (2013). Business process management: A comprehensive survey. *ISRN Software Engineering*, *19*(1), 1-37.

Weske, M. (2007). *Business process management: Concepts, languages, architectures*. Berlin, Germany: Springer.

Woerzberger, R., Kurpick, T., & Heer, T. (2008). On correctness, compliance and consistency of process models. In *17th IEEE International Workshops on Enabling Technologies* (pp. 251-252). Rome, Italy: IEEE Computer Society.

World Wide Web Consortium. (2014). SPARQL query language for RDF. Retrieved from http://www.w3.org/ TR/rdf-sparql-query

Yan, Z., Dijkman, R. M., & Grefen, P. (2012). Fast business process similarity search. *Distributed and Parallel Databases*, *30*(2), 105-144.

Glossary

Abstract syntax

The abstract syntax of a conceptual modeling language defines which concepts the language consists of and how these concepts are allowed to be related. For instance, the abstract syntax of event-driven process chains (EPC) prescribes that process functions and process events have to alternate in a process model.

Attributed graph

An attributed graph is a graph the vertices and edges of which can be assigned additional information, such as names, descriptions, numbers, and so on.

Conceptual model

A conceptual model is an abstract representation of a part of reality; for instance, an information system. In this paper, we regard conceptual models that can be represented through only graphs. Examples for such models are business process models, data models, and organizational charts.

Concrete syntax

The concrete syntax of a conceptual modeling language describes how the elements of the abstract syntax are represented. For instance, the concrete syntax of event-driven process chains (EPC) prescribes that process functions are represented by green rectangles, whereas the process flow is represented by arrows.

Directed edge

A directed edge of a graph is an edge that has a defined direction. Graphically, directed edges are mostly represented by arrows.

Edge

An edge of a graph connects the graph's vertices, where every edge always connects two vertices. Graphically, edges are mostly represented by lines.

Element

An element denotes any atomic part of a conceptual model. Hence, elements comprise objects, relationships, and attributes.

Element type

An element type denotes any atomic part of a conceptual modeling language. Hence, element types comprise object types, relationship types, and attribute types.

Execution semantics

The execution semantics of a process model describes the rules that define how process instances (i.e., particular executions of the model) are handled. For instance, it describes the possible paths a process instance can take "through" the process model.

Graph

A graph is a set-theoretical construct consisting of vertices and edges. Except for hyper-graphs (which are not of interest here), edges always connect two vertices. Graphs are oftentimes represented graphically through networks of icons and lines connecting them.

Integrated model

Integrated models are conceptual models of different modeling views that are interrelated by shared concepts. For instance, process models oftentimes reuse organizational units that were originally defined in an organizational chart.

Label semantics

Label semantics denotes the meaning of conceptual model elements' labels. For instance, the label "check invoice" of a process model's activity tells us what actually happen when the process activity is executed.

Loop

A loop in a conceptual model or, generally, in a graph, is a path that starts and ends in the same vertex.

Mapping

A mapping describes the possibilities to assign the elements of a pattern to the elements of a model in the context of model querying. For instance, a pattern consisting of two vertices and an edge between them can have multiple counterparts in a conceptual model. All of these counterparts must have the structure that is described in the pattern. This means that all of the counterparts must consist of two vertices that are connected by one edge.

Meta model

A meta model describes the abstract syntax of a modeling language using, in turn, a conceptual model. Meta models are mostly represented through data models.

Model analysis

Model analysis describes the task of extracting information out of a conceptual model that is particularly relevant for a given task. The information that is extracted can consist of the whole or parts of the conceptual model and can be aggregated arbitrarily. One possibility to support model analysis is model querying.

Model querying

Model querying means retrieving fragments of conceptual models that comply with given structural and semantic characteristics. Hence, model querying resembles graph pattern matching.

Modeling view

To handle the complexity of modeling information systems, common modeling frameworks separate the subject matter (i.e., the information system) into different modeling views. Hence, within different modeling views, we model different aspects of information systems. Commonly, modeling frameworks distinguish a data view, a process view, and an organizational view, amongst others.

Object

An object is an atomic part of a conceptual model that usually describes an entity of reality, for instance, a person "Mr. Bond", a document "invoice", or an activity "grant credit". Objects of graph-like conceptual models resemble graph vertices.

Object type

An object type is an atomic part of a conceptual modeling language that subsumes objects of a similar kind, for instance, persons, documents, and activities. Every object of a conceptual model thus belongs to a distinct object type.

Path

A path denotes a sequence of vertices in a graph, where each pair of vertices that follow each other in the sequence must be connected by an edge.

Pattern

A pattern acts as a template for a set of cohesive vertices and edges in a graph. In this context, the pattern describes a set of vertices, how they have to be connected by edges, and what attributes they should have. Patterns are used to find structures in graphs that match the pattern's properties.

Pattern matching

Pattern matching is the process of identifying substructures in graphs that match the properties of a pattern.

Pattern occurrence

A pattern occurrence is a subsection of a conceptual model that matches the properties of a pattern.

Query

A query of a model query language holds a conceptual model pattern that serves as input for a pattern matching process. The pattern matching process is executed on one or more conceptual models.

Relationship

A relationship is part of a conceptual model that usually describes relations of objects, for instance, the responsibility of a person "Mr. Bond" to execute an activity "grant credit". Relationships of graph-like conceptual models resemble graph edges.

Relationship type

A relationship type is part of a conceptual modeling language that subsumes relationships of a similar kind, for instance, responsibilities, process flows, or associations. Every relationship of a conceptual model, thus, belongs to a distinct relationship type.

Relaxed subgraph isomorphism

Relaxed subgraph isomorphism resembles subgraph homeomorphism. The difference is that for every edge of the graph to be contained in the other one, we have to define whether or not the edge can be replaced by a path and what minimum and maximum length that path may have.

Subgraph

A subgraph is a cohesive subsection of a graph.

Subgraph homeomorphism

Given two graphs, where we find a subdivision of one of the graphs or the graph itself as a subgraph of the other one, the two graphs are called subgraph-homeomorphic. A subdivision of a graph can be created by replacing edges of the graph by paths of arbitrary length.

Subgraph isomorphism

Given two graphs, where we find one of the graphs as a subgraph of the other one, the two graphs are called subgraph-isomorphic.

Sub-pattern

A sub-pattern is a pattern that can be reused by other patterns. For instance, a pattern may contain a path for which we defined that it must not contain specific elements. Those specific elements are called sub-pattern.

Tree

A tree is a structure in a graph that does not contain any loops.

Undirected edge

An undirected edge has no defined direction. Graphically, undirected edges are mostly represented as simple lines rather than arrows.

Vertex

A vertex is an atomic part of a graph. Vertices can be connected by edges.

About the Authors

Patrick Delfmann is an Associate Professor at the Department of Information Systems at the University of Münster, Germany and acting professor at the Department for IS Research at the University of Koblenz-Landau, Germany. He was the coordinator of several research projects, funded by national and international funding organizations. He teaches at the Universities of Koblenz-Landau and Münster and he was a visiting professor in Moscow (RU), Vienna (A), Biel (CH), and Osnabrück (D). His work comprises more than 90 scientific papers, many of which have appeared in international journals (e.g., *Information Systems*, *ISF*, *BISE*, *BPMJ*) and conference proceedings (e.g., *ICIS*, *ECIS*, *ER*). His main research interests are conceptual modeling, model analysis, semantic process modeling, business process compliance management, business process improvement, and process mining.

Dominic Breuker is a tech developer at Hitfox group, Berlin, Germany. He studied Information Systems at the University of Münster, and he holds a Master of Science and a PhD in Information Systems. Before he changed to industry, Dominic was a PhD student at the University of Münster, Germany and visiting researcher at the Ulsan National Institute of Science and Technology, South Korea, and Queensland University of Technology, Australia. His research interests include business process management, business intelligence, process mining, and machine learning. He published his work in peer-reviewed academic journals and presented at major international conferences.

Martin Matzner is an assistant professor at the Department of Information Systems at the University of Münster, Germany. In 2012, he received the PhD in Information Systems from the University of Münster for his work on the management of networked service business processes. His main research interests include service management, business process management, and business process analytics. In these areas, he concluded and currently manages a number of research projects funded by the European Union, the German Federal Government and industry. His work was published in peer-reviewed academic journals and presented at major international conferences.

Jörg Becker is a full professor and head of the Department of Information Systems at the University of Münster, Germany and the managing director of the European Research Center for Information Systems (ERCIS). He is Editor in Chief of the international journal of *Information Systems and e-BusinessManagement* (*ISeB*) and serves on various editorial boards. His work has appeared in several journals (e.g., *Communications of the Association for Information Systems*, *ISJ*, *SJIS*, *EJIS*) and been presented at international conferences (e.g., *ICIS*, *AMCIS*, *ECIS*). His research interests include information management, process management, conceptual modeling, retail IS, and strategic IT management consulting.