

The generic model query language GMQL – Conceptual specification, implementation, and runtime evaluation



Patrick Delfmann*, Matthias Steinhorst, Hanns-Alexander Dietrich, Jörg Becker

University of Münster – ERCIS, Institut für Wirtschaftsinformatik, Leonardo-Campus 3, 48149 Münster, Germany

ARTICLE INFO

Article history:

Received 20 December 2012

Received in revised form

16 April 2014

Accepted 16 June 2014

Recommended by M. Weske

Available online 26 June 2014

Keywords:

Business Process Management

Conceptual model repository

Conceptual model analysis

Generic model query language

Model querying

ABSTRACT

The generic model query language GMQL is designed to query collections of conceptual models created in arbitrary graph-based modelling languages. Querying conceptual models means searching for particular model subgraphs that comply with a predefined pattern query. Such a query specifies the structural and semantic properties of the model fragment to be returned. In this paper, we derive requirements for a generic model query language from the literature and formally specify the language's syntax and semantics. We conduct an analysis of GMQL's theoretical and practical runtime performance concluding that it returns query results within satisfactory time. Given its generic nature, GMQL contributes to a broad range of different model analysis scenarios ranging from business process compliance management to model translation and business process weakness detection. As GMQL returns results with acceptable runtime performance, it can be used to query large collections of hundreds or thousands of conceptual models containing not only process models, but also data models or organizational charts. In this paper, we furthermore evaluate GMQL against the backdrop of existing query approaches thereby carving out its advantages and limitations as well as pointing toward future research.

© 2014 Elsevier Ltd. All rights reserved.

1. Introduction

With the advancement of Business Process Management (BPM) technologies, many companies have started to develop and maintain large collections of conceptual models [1]. These collections include process models, data models as well as organizational charts and many other model types [2]. A task that frequently occurs when analysing large model collections is querying models to detect particular patterns in them. A pattern represents a subgraph of the overall model graph that complies with the structural and semantic properties defined in the predefined pattern query. Such a query is executed on a set of models to determine all pattern occurrences contained in that collection. Querying models serves a variety of different analysis purposes (see examples in Section 2 for more details). For instance, in order to improve a business process, the corresponding conceptual process model needs to be checked for weakness patterns [3]. In this context, a weakness pattern represents a subgraph of the overall model graph that points to a potentially inefficient part of a business process (e.g., subsequent switching of manual and automatic processing). Pattern matching also plays a role in design time process compliance checking. Many compliance rules place particular restrictions on the control flow of process models [4]. These restrictions can be represented as patterns that need to be identified in a model collection. A third application scenario of

* Corresponding author. Tel.: +49 251 83 38083.

E-mail address: patrick.delfmann@ercis.uni-muenster.de (P. Delfmann).

pattern matching is model translation, in which a model of a given language is transferred into a model of a different notation [5]. This can be achieved by finding patterns that are translated to predefined model fragments of the target language.

All of these model analysis scenarios have in common that a collection of conceptual models needs to be queried in order to find occurrences of a particular pattern within the model graphs. These model collections typically contain models developed in many different modelling languages [2]. Process models can, for instance, be developed using Business Process Model and Notation (BPMN) [6], Event-driven Process Chains (EPC) [7], or Petri Nets [8]. Data models can be developed using Entity-Relationship (ER) models [9] or Unified Modelling Language (UML) class diagrams [10], to name only a few. To support these model analysis scenarios, the literature has put forth the concept of model query languages [11]. A model query language essentially consists of two main components. First, it provides a set of *constructs to define a pattern query*. A pattern query is a representation of a model subgraph that needs to be found within the overall model graph for various analysis purposes (see above). Second, a query language contains a *pattern matching algorithm* that takes the query specification and a model (or a set of models) as input and returns all occurrences of the subgraph represented by the query within the input model(s). Model query languages therefore enable pattern matching in conceptual models.

A number of such query languages have been proposed in recent years (see [11] for a comprehensive survey as well as Section 7.3 in this paper). However, these query languages are designed to support pattern matching in a particular type of model (e.g., process models [12,13]) or in models developed in a particular modelling language [14]. We follow the argument of van der Aalst [15] claiming that model analysis approaches put forth in the scientific community need to support analysing models of any graph-based modelling language if they are ever to disseminate into corporate reality. This is due to the fact that different organizations typically use different modelling languages [16]. After all, specialized approaches that work only on particular types of models developed in a particular language are not useful for an organization that uses different modelling languages. For example, an organization that uses EPCs for process modelling cannot use a query language like BPMN-Q that is solely designed for querying BPMN models. To this end, the generic model query language GMQL has recently been proposed [17,18]. It is based on the idea that any model is essentially an attributed graph that can be represented by the set of its objects (i.e., graph vertices) and the set of its relationships (i.e., graph edges). Each object and each relationship has a set of attributes that further specifies the semantics of the element (e.g., type, label, description, etc.). As *constructs to define a pattern query* GMQL provides set-altering functions and operators that take these two basic sets as input and perform various operations on them. These functions and operators can be nested to construct tree-like pattern queries (see details below). The GMQL *pattern matching algorithm* walks through this query tree calculating its leaf node first and returning the results to the next higher level. In doing so, the result of one function or operator call serves as input for the next. At the end, GMQL returns every pattern occurrence, meaning the model fragments that comply with the query definition and are thus of interest for the analyst.

GMQL treats any conceptual model as an attributed graph. It is thus closely related to graph theory. In graph theory, the problem of pattern matching is known as the problem of subgraph isomorphism (SGI) [19]. SGI, however, is concerned with identifying one-to-one mappings between a pattern graph and a subsection of a search graph. This means, that all nodes and all edges of a pattern graph are mapped to a subset of nodes and edges in the model graph. As we will see in the examples section below, this is too restrictive for many model analysis scenarios, because SGI requires the exact pattern structure to be known a priori. Pattern matching in model analysis scenarios rather requires identifying subsections of a model that contain element paths of previously unknown length. In graph theory, this can be achieved with algorithms for subgraph homeomorphism (SGH) [20]. However, corresponding algorithms by default map all edges in the pattern graph on all possible paths in the model graph leading to a huge number of (mostly not suitable) pattern matches. Consequently, SGI on the one hand is too restrictive and SGH on the other hand is too unrestrictive for the purpose of pattern matching in conceptual models. GMQL therefore is a combination of both. It allows for finding model subgraphs that are partly isomorphic and partly homeomorphic to a predefined pattern query. In doing so, particular edges in the pattern query can be mapped to edges in the model, whereas other pattern edges can be mapped to paths in the model.

The paper at hand extends two previous papers presenting the initial concept of GMQL [17] and a preliminary performance evaluation for EPC models [18]. This paper extends these findings as follows:

- We analyse common patterns proposed in the literature in terms of their graph structure. In doing so, we derive a refined set of functional requirements for GMQL.
- We present a formal specification of the GMQL syntax using Extended Backus-Naur Form (EBNF) statements.
- We present a formal specification of the new GMQL semantics using set operations.
- We present a detailed description of GMQL's matching algorithm.
- We present exemplary pattern queries for each of the model analysis scenarios discussed in the literature. Thereby, we demonstrate the applicability of GMQL.
- We analyse the theoretical worst-case complexity of the GMQL matching algorithm.
- We extend the performance analysis presented in [18] to include runtime measurements for ER models.
- As the number and size of conceptual models contained in a collection is steadily increasing [21], runtime performance of a pattern matching approach is of paramount importance. The measurements presented in [18] suggest that runtimes increase significantly whenever the query contains path functions. We therefore changed the implementation of these functions to include an efficient depth first search that is based on the idea of tagging already visited elements instead of

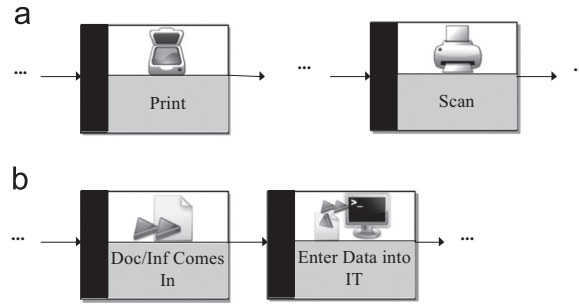


Fig. 1. Patterns representing a switch between manual and automatic processing as discussed in [3].

reiterating a corresponding list. The collected runtime data indicate that GMQL is able to execute most pattern queries within fractions of a millisecond. In rare cases, if large amounts of pattern matches are found runtimes can increase to a few hundred milliseconds depending on the size and type of the model.

- We conduct a multiple regression analysis to explain the percentage of runtime variability that can be attributed to the size of the model and the number of returned pattern matches. We demonstrate that these two factors account for more than ninety per cent of runtime variability and can thus be used to estimate GMQL's runtime performance in practice.
- We provide a feature-based comparison of GMQL and related model query languages proposed in the literature.

The remainder of this paper is structured as follows. In the following section, we elaborate on model analysis scenarios discussed in the literature that involve pattern matching (Sections 2.1–2.5). On the basis of these exemplary patterns, we deduce requirements for a generic model query language (Section 2.6). We then introduce GMQL by specifying its semantics and syntax (Section 3). In Section 4, the broad applicability of GMQL is demonstrated by providing an exemplary pattern query for each of the model analysis scenarios discussed in Section 2. Details on the implementation of GMQL are provided in Section 5. We conduct an analysis of GMQL's theoretical worst-case complexity (Section 6.1) and its practical runtime performance (Section 6.2). We then elaborate on limitations of GMQL (Section 7.1) and the previously described performance analysis (Section 7.2) as well as perform a feature-based comparison of GMQL and related work (Section 7.3). The paper closes with a summary of its contributions and an outlook to future research in Section 7.4.

2. Motivating examples

The purpose of a model query language is to enable pattern matching in conceptual models (see Section 1). Detecting patterns in conceptual models serves a variety of different purposes. In this section, we will elaborate on examples of patterns in conceptual models discussed in the literature. On the basis of these examples, we carve out requirements for a generic model query language. We identify five areas in which pattern detection plays an important role, namely the areas of business process weakness detection, business process compliance management, model transformation, syntactical correctness checking, and model comparison. In all five areas, we identify typical patterns, explain them in terms of their underlying model graph structure and derive requirements for a generic model query language designed to express corresponding pattern queries.

2.1. Business process weakness detection

Many organizations have started to describe their process landscape with the help of process models [1]. These models are used as a means of analysing the underlying business processes. Such an analysis may serve to identify process weaknesses. A process weakness translates to a particular model fragment (i.e., a pattern) that represents a part of the process that is handled inefficiently [3]. In the domain of public administration, [3,22] propose a set of such weakness patterns. The authors express these patterns using the semantic business process modelling language (SBPML) providing a standardized set of activities that describe frequently occurring tasks in public administrations. One such pattern describes a switching between manual and automatic processing. This weakness occurs when information is printed and once again entered manually into an IT system (e.g., by scanning a document). Fig. 1 depicts two possible pattern occurrences representing such a weakness.

In terms of graph structure, the pattern in Fig. 1(a) represents a path of activities starting in a scan task and ending in a print task. The pattern in Fig. 1(b), however, represents two directly related activities. A generic model query language should therefore be able to express pattern queries consisting of path structures (see R1.5 in Section 2.6) and adjacent model elements (see R1.4). A query language should also be able to express pattern queries consisting of model elements having particular labels (see R1.1). Fig. A1 in the appendix lists a number of further weakness patterns, their graph structure and subsequent query language (QL) requirements. All patterns were reported in [3,22].

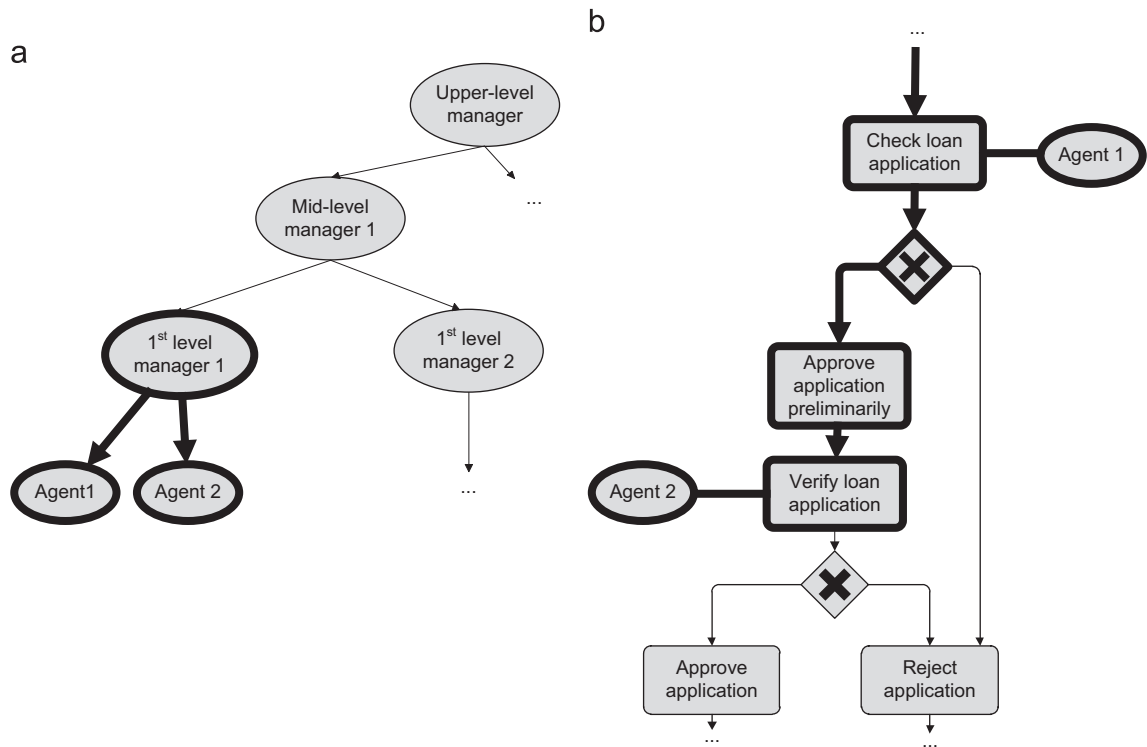


Fig. 2. Violation of separation of duties pattern.

2.2. Business process compliance management

Design time business process compliance checking is concerned with detecting violations of internal or external rules and regulations in conceptual models [4]. A compliance rule imposes particular restrictions on the control flow of process models. Therefore, a compliance violation represents a model fragment that does not adhere to these restrictions. It thus translates to a pattern in the model graph. A frequently discussed example is the separation of duties pattern claiming that two particularly critical business activities need to be executed by different organizational units [23]. Fig. 2 depicts an excerpt of an organizational chart (a) and an excerpt of a BPMN-like process model (b) containing a corresponding compliance violation. The process model represents a loan application process during which the application is first checked by an agent and later needs to be verified by the supervising employee of that agent. In this example the verification task is also performed by an agent. The organizational chart indicates that this second agent is not the supervising employee of the first agent. Instead, both agents are on the same organizational level. The models thus contain a violation of the separation of duties rule. The fragments of the respective models are highlighted in bold. This compliance rule thus extends over two different models.

In terms of graph structure, the compliance rule translates to a path in the process model that starts in a check activity and leads to a verification activity. Both start and target elements are furthermore adjacent to an organizational unit. A generic model query language thus has to be able to express pattern queries consisting of paths of model elements (see R1.5) having particular labels and types (see R1.1). In addition, the pattern consists of two substructures, namely the element path and two activities and their adjacent organizational units. A query language thus should be able to express pattern queries consisting of a combination of various substructures (see R2). Also, a query language should be able to express pattern queries consisting of neighbouring elements and the relationships between these elements (see R1.4). In Fig. 2, the organizational units are directly related to a task object. A query language should also be able to distinguish directed and undirected edges. In this example, the organizational units are connected to their activities via undirected edges, while the control flow is directed.

To find the highlighted structure in the organizational chart, a query language has to be able to express pattern queries consisting of adjacent elements (see R1.4). It also needs to be able to compare the values of attributes to one another (see R1.3). The label of the first organizational unit in the process model (i.e., the one attached to the check activity) needs to be identical to one of the labels of the units contained in the highlighted part of the chart. At the same time, the label of the second organizational unit in the process model needs to be identical to the correspondingly other label. Fig. A2 in the appendix contains further examples of compliance violation patterns discussed in the literature [23,24].

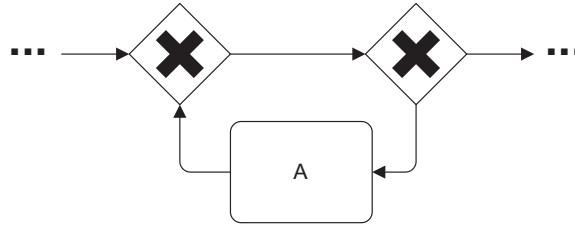


Fig. 3. WHILE-pattern [25].

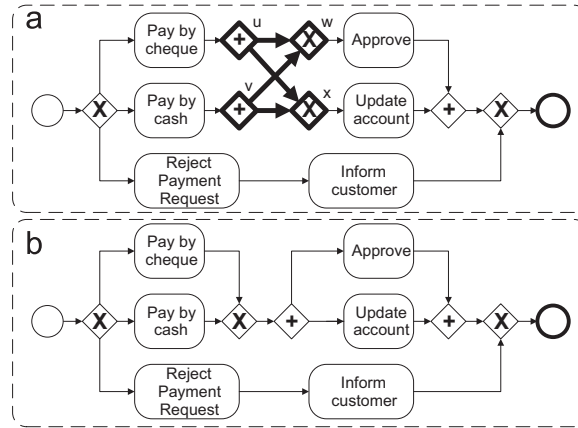


Fig. 4. An unstructured process model (a) and its structured counterpart (b) (adapted from [31]).

2.3. Model translation

Model translation deals with transforming a conceptual model into a model of a different language. Notable examples discussed in the literature include translations from BPMN into BPEL [5,25]. Model translation requires identifying model fragments that are then translated to predefined fragments in the target notation. Such a translation therefore first requires detecting typical patterns in the source model. Consider for example the WHILE-pattern discussed in [25] (cf. Fig. 3). The authors propose to translate this pattern into predefined BPEL code. In terms of graph structure, this pattern translates to a path that starts and ends in the same element. This element needs to be of a particular type, in this case “XOR”. A pattern query language thus has to be able to express pattern queries consisting of loops (see R1.5) of elements having a particular label (see R1.1). Fig. A3 in the appendix contains additional patterns relevant for model translation that we extracted from the literature [5,25]. The appendix also contains typical patterns relevant in the domain of database management.

2.4. Syntactical correctness checking

A fourth application area of pattern matching is syntactical correctness checking. Corresponding patterns represent model fragments that explicitly violate the syntactical rules of a modelling language. In addition, we also include structures in this category that represent violations of modelling guidelines [26] or may lead to execution problems at runtime [27,28]. Process models, for instance, should be (well-) structured [29], meaning that for each node having several outgoing arcs (a split) there has to be a corresponding node having several ingoing arcs (a join) such that the set of nodes encapsulated by the split and join form a so called single-entry-single-exit (SESE) fragment [30]. Fig. 4(a) depicts a process model that is unstructured, because there are no corresponding nodes for the AND splits *u* and *v* as well as for the XOR joins *w* and *x*. Fig. 4(b) depicts the structured version of that model. The problem of transforming an unstructured process model into a structured one has been intensively studied by Polyvyanyy et al. [31]. The algorithm presented in that work takes an unstructured fragment of a process model as input and returns its structured counterpart. As a pre-processing step of this algorithm, unstructured fragments first have to be identified. This means that patterns representing unstructured fragments have to be found within the process model graph. In Fig. 4(a), the nodes of such a subgraph are highlighted in bold. In terms of requirements for a model query language it has to be able to express joining and splitting nodes (i.e., nodes that have at least two outgoing or two ingoing arcs) (see R1.2). Fig. A5 in the appendix contains additional examples of patterns related to a syntactical correctness checking that we identified in the literature [27,28].

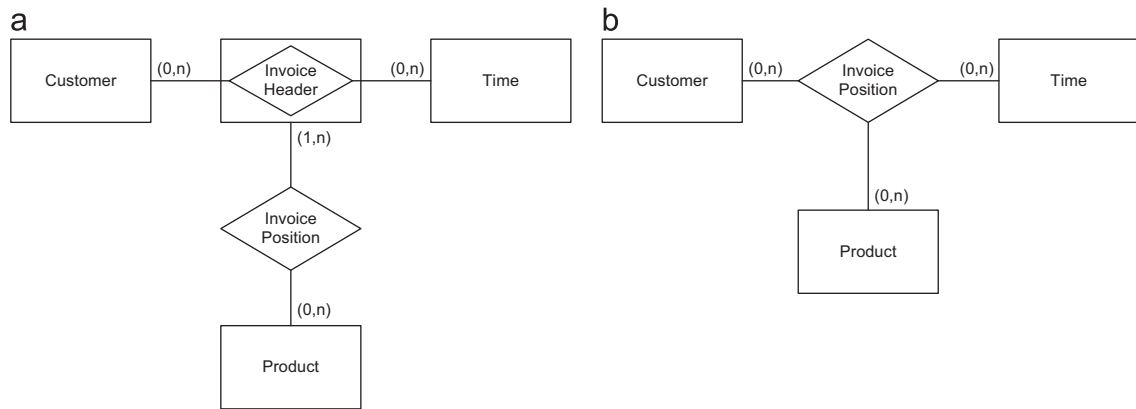


Fig. 5. Different receipt structures.

2.5. Model comparison

Pattern matching is a pre-processing step to compare and subsequently integrate different models. The literature proposes a great number of different model comparison metrics [32]. These metrics measure the similarity of two models on a scale from zero to one. A similarity score of zero means that the models are completely dissimilar and a score of one indicates they are identical. Pattern matching can be used to build up these similarity metrics [33]. Intuitively, models each containing a number of particular fragments are similar to one another and can thus more easily be integrated to one consolidated model. Pattern matching is thus one step in the integration process. Fig. 5 contains two similar receipt structures depicted as ER models. Fig. 5(a) represents the receipt structure implemented in the SAP ERP System [34] containing a relational entity type. Fig. 5(b) represents a similar structure using a ternary relationship type. Conceptual data models containing either of these structures can possibly be integrated to one consolidated model. Again, pattern matching is required to identify corresponding structures within the conceptual data models. In terms of graph structure, the pattern in Fig. 5(a) requires a query language to be able to express pattern queries consisting of a combination of various substructures (see R2). The element path from entity type “Customer” to entity type “Time”, for instance, needs to be joined with the element paths from entity type “Product” to relational entity type “Invoice Header”. The pattern in Fig. 5(b) requires the query language to express queries consisting of elements having a particular type and label (see R1.1) and a particular number of edges (see R1.2). In this example, a relationship type labelled “Invoice Position” is required to have three edges connecting the relationship type to three entity types.

2.6. Requirements for a generic model query language

In the previous sections we have demonstrated that pattern matching plays an important role in many model analysis scenarios. These scenarios may involve different types of models like process models, organizational charts or data models. In addition, organizations may use multiple modelling languages to describe various aspects of their corporate reality (i.e., process models, data models and organizational models at the same time, which are sometimes even integrated). The literature that addresses these model analysis scenarios is reflecting this range of model types and languages. Other than patterns for process models created in several process modelling languages, the literature also includes patterns for ER models (see analysis scenarios “model translation” and “model comparison and integration”) or organizational charts (see analysis scenario “business process compliance management”). We argue that it is thus beneficial to develop a generic model query language that can be applied to models developed in any graph-based modelling language and that can be used in all of the mentioned model analysis scenarios. This can be achieved by treating any model as an attributed graph (see formal definitions in the following section). Analysing the proposed patterns from a graph-theoretical point of view leads to a set of reoccurring requirements for a generic model query language (see cross references in Sections 2.1–2.5). These requirements reflect the structural properties of the patterns proposed in the literature for various analysis purposes. Many requirements occur multiple times in many different scenarios. In addition, many requirements occur in patterns for different model types. Consider for example requirement R1.4 stating that a generic model query language has to be able to express queries consisting of adjacent model elements. This requirement is relevant for process models and organizational charts (see analysis scenario “business process compliance management”) as well as for data models (see analysis scenario “model translation”). We therefore conclude that this set of requirements is representative of the model analysis scenarios that are currently being discussed in the literature. These requirements can be summarized as follows.

- **R1:** A generic query language should be able to express pattern queries for models of any graph-based modelling language. R1 can be understood as a superior requirement for a generic model query language, whereas requirements R1.1–R1.5 address the particular graph-structural properties of the model fragments discussed above.

- **R1.1:** A generic query language should be able to express pattern queries consisting of single model elements that have a particular type or label. As such information can be stored in respective attributes the query language should offer functionality to consider attribute information in its matching process.
- **R1.2:** A generic query language should be able to express pattern queries consisting of single elements and all of their arcs (i.e., edges of the model graph). In addition, the query language should be able to express pattern queries consisting of elements having particular arcs (e.g., ingoing, outgoing, directed or undirected) and elements having a particular number of arcs.
- **R1.3:** A generic query language should be able to express pattern queries that allow for comparing the values of particular attributes to one another (cf. separation of duties pattern).
- **R1.4:** A generic query language should be able to express pattern queries consisting of adjacent elements and the directed or undirected arcs between them.
- **R1.5:** A generic query language should be able to express pattern queries consisting of directed and undirected element paths of arbitrary length. In addition, particular paths that, for instance, do or do not contain certain elements need to be distinguished. Furthermore, a generic query language should be able to express pattern queries consisting of paths starting and ending in the same element (i.e., loops).
- **R2:** A generic query language has to be able to express pattern queries that combine pattern substructures.
- **R3:** A generic query language should contain a pattern matching algorithm that takes one model or a set of models and a pattern query as input and returns all pattern occurrences contained in the input model(s).

In the following sections we will reference these requirements while detailing which GMQL feature fulfils which requirement.

3. GMQL specification

3.1. Preliminaries

To specify a query language that can be used to detect patterns in conceptual models of any modelling language, we regard any conceptual model graph, be it a process model, data model, or organizational chart as a tuple, as captured by the following definition.

Definition 1. (Conceptual model): A conceptual model is a tuple $M=(O,R,T,V,W,\alpha,\beta,\gamma)$, where O is a non-empty set of objects (vertices) and R is a non-empty set of relationships (edges). We define $E=O\cup R$ as the set of all model elements and $R=R_D\cup R_U$ as the set of all directed and undirected relationships. $R_D\subseteq E\times E$ defines the set of all directed relationships between the elements of a conceptual model (i.e., directed edges of the model graph). $R_U=\{\{e_1,e_2\}|e_1,e_2\in E\}$ defines the set of all undirected relationships between the elements of a conceptual model (i.e., undirected edges of the model graph). T is a non-empty-set of element types. It is used to assign every object and every relationship a type (e.g., for objects: “activity”, “event”, “entity type”, “class”, etc., and for relationships: “control flow”, “data input”, “association”, etc.). $\alpha: E\rightarrow T$ is a function that performs that assignment. As model elements can have various properties like “description”, “cost”, “name”, “transition probability”, etc., both elements and relationships can be assigned to attributes. To be as flexible as possible in the specification and assignment of attributes, we regard them as objects $A\subseteq O$, which are assigned to a special kind of element type $T_a\subseteq T$, that is, $\alpha(a)\in T_a$, $a\in A$. Attributes are assigned to elements via undirected relationships. Attributes are furthermore distinguished from other objects through the fact that they can carry values V . The function $\beta: A\rightarrow V$ assigns every attribute a value (e.g., “article”, “check invoice”, “CEO”, etc.). In contrast, both non-attribute objects and relationships do not carry values. However, they can be assigned attributes with values. W is a non-empty set of attribute data types (e.g., string, integer, etc.). The function $\gamma: A\rightarrow W$ assigns each attribute a data type.

We define a pattern occurrence as a subgraph of a conceptual model, as captured by the following definition.

Definition 2. (Pattern occurrence): Given a conceptual model M as defined above, a pattern occurrence is a tuple $P=(O',R',T',V',W',\alpha',\beta',\gamma')$ with $O'\subseteq O$, $R'\subseteq R$, $T'\subseteq T$, $V'\subseteq V$, $W'\subseteq W$, $\alpha'=\alpha|_{E'}$, and $\beta'=\beta|_{A'}$, $\gamma'=\gamma|_{E'}$, $E'=O'\cup R'$, $A'\subseteq O'$.

Definition 3. (Pattern match): A pattern match is a function $\delta: M\rightarrow\{P\}$ that maps a model M to the set of all pattern occurrences $\{P\}$ contained in the model.

In order to specify GMQL, we first have to determine similarities of all conceptual modelling languages. As depicted in Fig. 6 any such language can be described in terms of the set T of element types it offers. These element types constitute any atomic part of a modelling language. Element types are further subdivided into its sets T_O of object types and T_R of relationship types. Each relationship type has a source element type, from which it originates, and a target element type, to which it leads. Furthermore, each relationship type exhibits an attribute “Directed” specifying if the relationship type is directed or not. In the latter case, the direction determined by “IsSourceOf” and “IsDestinationOf” is ignored. The EPC language can, for instance, be described in terms its object types function, event, and {AND|OR|XOR}-gateway [7]. The set of EPC relationship types defines what object types can be connected to what other object types. The same holds true for the

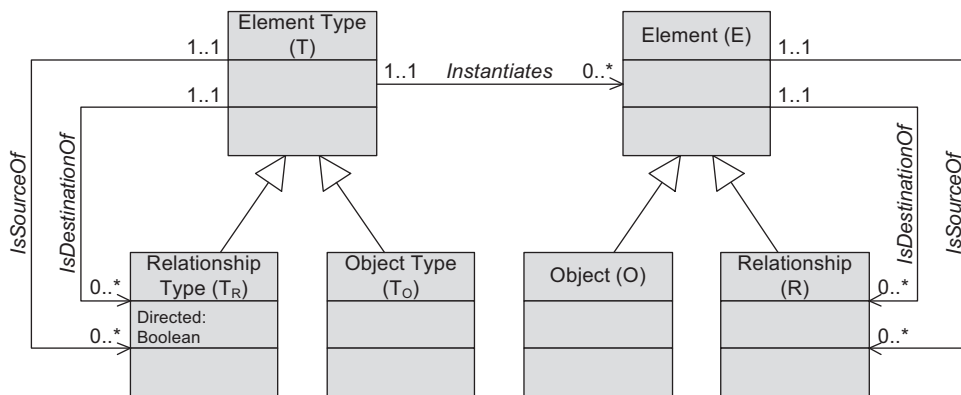


Fig. 6. Generic pattern specification environment (adapted from [17]).

ERM language, which consists, amongst others, of entity types and relationship types [9]. Developing a conceptual model means instantiating the element types of the particular language. In line with Definition 1, a conceptual model can thus be described in terms of the set E of its elements. This set of elements then consists of the set O of model objects (i.e., vertices) and the set R of model relationships (i.e., edges). Note that in line with Definition 1, the source and target elements of a given relationship may be other relationships, because some modelling languages allow for edge-to-edge relationships (e.g., association class in UML class diagrams).

3.2. GMQL semantics

GMQL is based on the idea that essentially any conceptual model can be represented as the set O of its objects and the set R of its relationships. The objects represent the vertices of the model graph, whereas the relationships constitute its edges. In doing so, GMQL can be used to define and execute pattern queries for models of any graph-based modelling language (R1). GMQL provides set-altering functions and operators that take these two sets as input and perform particular operations on them, returning all nodes and edges of a model that comply with the query definition and hence are of interest for the analyst. In the following, we will specify the semantics of each GMQL function and operator using set operations. The semantics of a given query can be derived from the semantics of all of its components. Each of the functions and operators returns an output set that serves as input for a further call to another function or operator. In doing so, a pattern query is built up successively. Hence, a pattern query exhibits a tree structure that is traversed in post order. Therefore, the left subtree is calculated first followed by the right subtree. The root node of a given subtree is calculated last. The result of any given level of the tree thus serves as input for the next level. The overall result of the matching algorithm is a set of sets. Each inner set of this output set contains all model elements that represent one pattern occurrence. Fig. 7 demonstrates the matching approach. The exemplary pattern that is searched for consists of three objects and their relationships. The matching algorithm returns a set of pattern occurrences. Each pattern occurrence consists of a set of model elements. Hence, the matching algorithm returns a subset of the power set of all model elements. Each pattern occurrence is built up successively according to the semantics of the given query.

Other than the sets E , O , and R , we define the following sets to be the basic input sets of GMQL. Note that the sets X , Y and Z serve as input for functions and operators and represent subsets of the corresponding sets E , O and R , because X , Y and Z may contain element subsets that have been created by other functions and operators.

- E : the set of all elements available; $e \in E$ is a particular element.
- $P(E)$: the power set of E .
- O : the set of all objects available; $O \subseteq E$; $o \in O$ is a particular object.
- R : the set of all relationships available; $R \subseteq E$; $r \in R$ is a particular relationship.
- T : the set of all element types available; $t \in T$ is a particular element type.
- T_O : the set of all object types available; $T_O \subseteq T$; $t_O \in T_O$ is a particular object type.
- T_R : the set of all relationship types available; $T_R \subseteq T$; $t_R \in T_R$ is a particular relationship type.
- I : the set of all instantiations available; $I \subseteq T \times E$; $(t, e) \in I$ is a particular instantiation.
- D : the set of all relationship destinations available; $D \subseteq E \times R$; $(e, r) \in D$ is a particular destination.
- S : the set of all relationship sources available; $S \subseteq E \times R$; $(e, r) \in S$ is a particular source.
- X : a set of elements with $x \in X \subseteq E$.
- X_k : sets of elements with $X_k \subseteq E$ and $k \in N_0$.

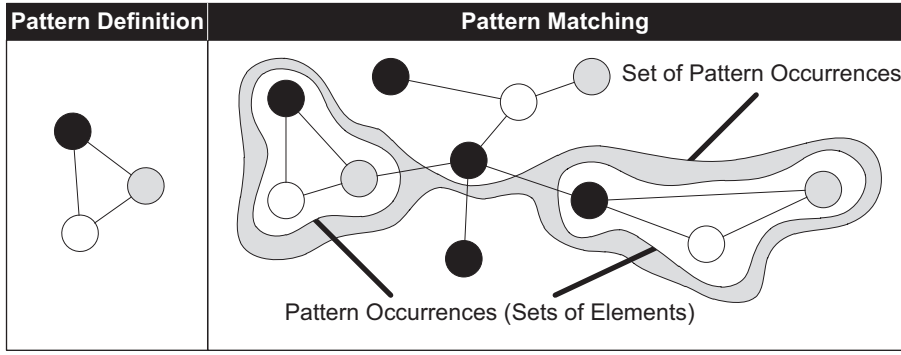


Fig. 7. Matching process (adapted from [17]).

- x_i : distinct elements with $x_i \in E$ and $i \in N_0$
- Y : a set of objects with $y \in Y \subseteq O$.
- Z : a set of relationships with $z \in Z \subseteq R$.
- n_x : a positive natural number $n_x \in N_I$
- R_d : the set of all directed relationships available; $R_d \subseteq R$, $((t_R, r_d) \in I \wedge t_R.directed = TRUE \wedge t_R \in T_R) \forall r_d \in R_d$
- D_d : the set of all directed relationship destinations available; $D_d \subseteq D$, $(r_d \in R_d) \forall (e, r_d) \in D$
- S_d : the set of all directed relationship sources available; $S_d \subseteq S$, $(r_d \in R_d) \forall (e, r_d) \in S$
- D_u and S_u are undirected counterparts; $D_u = D \setminus D_d$ and $S_u = S \setminus S_d$

The GMQL functions fall into four classes in order to meet requirements R1.1–R1.5. The first class of functions takes a simple set of elements as input and returns those elements that have particular characteristics (R1.1). All functions of this first class return a simple set of elements. Consider, for example, the separation of duties pattern outlined in Fig. 2. This pattern requires the matching algorithm to determine objects that have a particular type. In this case, the query language has to express objects that are of types “task” and “organizational unit”. These element types are defined in the language specification. In this example, they are thus defined in the BPMN language specification. Furthermore, it has to determine objects that have a particular label (i.e., objects assigned to a particular attribute carrying a particular value).

- $ElementsOfType(X, t) \subseteq E$ is provided with a set of elements X and a distinct element type t . It returns a set containing all elements of X that belong to the given type t . This function, for instance, allows for finding all event objects in an EPC model.
 - $ElementsOfType(X, t) = \{x \in X \mid (t, x) \in I\}$
- $ElementsWithAttributeOfValue(X, t_a, v) \subseteq E$ is provided with a set of elements X , an attribute of type $t_a \in T_a$, and a value $v \in V$. It returns a set of elements that have an attribute of type t_a carrying the value v . This function, for instance, allows for finding all model objects that are assigned to an attribute of type “label” having a value “Check invoice”.
 - $ElementsWithAttributeOfValue(X, t_a, v) = \{x \in X \mid (x, r) \in S_u \wedge (a, r) \in D_u \vee (a, r) \in S_u \wedge (x, r) \in D_u, (t_a, a) \in I, \beta(a) = v, a \in A, r \in R\}$
- $ElementsWithAttributeOfDatatype(X, w) \subseteq E$ is provided with a set of elements X and an attribute data type w . It returns all elements that have attributes of the given data type. Valid data types are String, Integer, Double, Boolean and Enum.
 - $ElementsWithAttributeOfDatatype(X, w) = \{x \in X \mid (x, r) \in S_u \wedge (a, r) \in D_u \vee (a, r) \in S_u \wedge (x, r) \in D_u, \gamma(a) = w, a \in A, r \in R\}$

A second class of functions determines elements having (a particular number or type (of outgoing, ingoing, directed or undirected)) relationships (R1.2). Consider the example of the deadlock pattern given in Fig. A5. This pattern contains splitting and joining nodes, meaning nodes that have at least two outgoing or ingoing relationships. A query language should therefore be able to express elements that have a predefined number of outgoing or ingoing relationships. Building on this rationale, GMQL also provides functions that determine elements having relationships of a certain type. Consider the receipt structure patterns depicted in Fig. 5. In Fig. 5(b) the relationship type is connected to three entity types via relationships of type $EntityType \rightarrow RelationshipType$. In Fig. 5(a), however, the relationship type is connected to relationships of different types (i.e., $EntityType \rightarrow RelationshipType$ and $RelationalEntityType \rightarrow RelationshipType$). Ergo, information about the type of a relationship needs to be included in the matching algorithm to identify such structures. The functions introduced below provide corresponding functionality. They all return a set of sets with each inner set containing one element and all of the relationships that meet the specific criteria set by the function.

- $ElementsWithRelations(X, Z) \subseteq P(E)$ is provided with a set of elements X and a set of relationships Z . Each inner set of the result set contains one element of X and all of its relationships of Z .
 - $ElementsWithRelations(X, Z) = \{EWR(x, Z)\}$ with $x \in X$ and $EWR(x, Z) = \{z \in Z \mid (x, z) \in D \vee (x, z) \in S\} \cup \{x\}$.

- $ElementsWithSuccRelations(X, Z_d) \subseteq P(E)$ is provided with a set of elements X and a set of directed relationships $Z_d \subseteq R_d$. Each inner set of the result set contains one element of X and its succeeding (outgoing) relationships of Z_d .
 - $ElementsWithSuccRelations(X, Z_d) = \{EWSR(x, Z_d)\}$ with $x \in X$ and $EWSR(x, Z_d) = \{z_d \in Z_d \mid (x, z_d) \in S_d\} \cup \{x\}$.
- $ElementsWithPredRelations(X, Z_d) \subseteq P(E)$ is provided with a set of elements X and a set of directed relationships $Z_d \subseteq R_d$. Each inner set of the result set contains one element of X and its preceding (ingoing) relationships of Z_d .
 - $ElementsWithPredRelations(X, Z_d) = \{EWPR(x, Z_d)\}$ with $x \in X$ and $EWPR(x, Z_d) = \{z_d \in Z_d \mid (x, z_d) \in D_d\} \cup \{x\}$.
- $ElementsWithRelationsOfType(X, Z, t_R) \subseteq P(E)$ is provided with a set of elements X , a set of relationships Z and a distinct relationship type t_R . Each inner set of the result set contains one element of X and all its relationships of Z that are of type t_R .
 - $ElementsWithRelationsOfType(X, Z, t_R) = ElementsWithRelations(X, ElementsOfType(Z, t_R))$
- $ElementsWithSuccRelationsOfType(X, Z_d, t_R) \subseteq P(E)$ is provided with a set of elements X , a set of directed relationships Z_d and a distinct relationship type t_R . Each inner set of the result set contains one element of X and all its outgoing relationships of Z_d that are of type t_R .
 - $ElementsWithSuccRelationsOfType(X, Z_d, t_R) = ElementsWithSuccRelations(X, ElementsOfType(Z_d, t_R))$
- $ElementsWithPredRelationsOfType(X, Z_d, t_R) \subseteq P(E)$ is provided with a set of elements X , a set of directed relationships Z_d and a distinct relationship type t_R . Each inner set of the result set contains one element of X and all its ingoing relationships of Z_d that are of type t_R .
 - $ElementsWithPredRelationsOfType(X, Z_d, t_R) = ElementsWithPredRelations(X, ElementsOfType(Z_d, t_R))$
- $ElementsWithNumberOfRelations(X, n_x) \subseteq P(E)$ is provided with a set of elements X and a distinct number n_x . Each inner set of the result set contains one element of X and its n_x relationships of R .
 - $ElementsWithNumberOfRelations(X, n_x) = \{EWN(x) \mid |EWN(x)| = n_x + 1\}$ with $EWN(x) = \{r \in R \mid (x, r) \in D \vee (x, r) \in S\} \cup \{x\}$
- $ElementsWithNumberOfSuccRelations(X, n_x) \subseteq P(E)$ is provided with a set of elements X and a distinct number n_x . Each inner set of the result set contains one element of X and all its n_x outgoing relationships of R .
 - $ElementsWithNumberOfSuccRelations(X, n_x) = \{EWSNR(x) \mid |EWSNR(x)| = n_x + 1\}$ with $EWSNR(x) = \{r_d \in R_d \mid (x, r_d) \in S_d\} \cup \{x\}$
- $ElementsWithNumberOfPredRelations(X, n_x) \subseteq P(E)$ is provided with a set of elements X and a distinct number n_x . Each inner set of the result set contains one element of X and all its n_x ingoing relationships of R .
 - $ElementsWithNumberOfPredRelations(X, n_x) = \{EWNPR(x) \mid |EWNPR(x)| = n_x + 1\}$ with $EWNPR(x) = \{r_d \in R_d \mid (x, r_d) \in D_d\} \cup \{x\}$
- $ElementsWithNumberOfRelationsOfType(X, t_R, n_x) \subseteq P(E)$ is provided with a set of elements X , a distinct relationship type t_R and a distinct number n_x . Each inner set contains one element of X and all its n_x relationships of R that are of type t_R .
 - $ElementsWithNumberOfRelationsOfType(X, t_R, n_x) = \{EWNRT(x, t_R) \mid |EWNRT(x, t_R)| = n_x + 1\}$ with $x \in X$ and $EWNRT(x, t_R) = \{r \in R \mid (t_R, r) \in I \wedge ((x, r) \in D \vee (x, r) \in S)\} \cup \{x\}$
- $ElementsWithNumberOfSuccRelationsOfType(X, t_R, n_x) \subseteq P(E)$ is provided with a set of elements X , a distinct relationship type t_R and a distinct number n_x . Each inner set contains one element of X and all its n_x outgoing relationships of R that are of type t_R .
 - $ElementsWithNumberOfSuccRelationsOfType(X, t_R, n_x) = \{EWSNRT(x, t_R) \mid |EWSNRT(x, t_R)| = n_x + 1\}$ with $x \in X$ and $EWSNRT(x, t_R) = \{r_d \in R_d \mid (t_R, r_d) \in I \wedge (x, r_d) \in S_d\} \cup \{x\}$
- $ElementsWithNumberOfPredRelationsOfType(X, t_R, n_x) \subseteq P(E)$ is provided with a set of elements X , a distinct relationship type t_R and a distinct number n_x . Each inner set contains one element of X and all its n_x ingoing relationships of R that are of type t_R .
 - $ElementsWithNumberOfPredRelationsOfType(X, t_R, n_x) = \{EWNPRT(x, t_R) \mid |EWNPRT(x, t_R)| = n_x + 1\}$ with $x \in X$ and $EWNPRT(x, t_R) = \{r_d \in R_d \mid (t_R, r_d) \in I \wedge (x, r_d) \in D_d\} \cup \{x\}$

Instead of concrete element types, attributes, or attribute values, all functions contained in the first two classes can also take an input of type *variable*. These variables can be compared to one another using variable equations. In doing so, elements having equal or unequal attribute values can be detected (R1.3). A third class of functions determines adjacent elements and the relationships connecting them (R1.4). Consider the example of the XOR-control pattern depicted in Fig. A5. The joining AND node is directly succeeding a start event. This substructure therefore contains two adjacent elements (the AND join and the event) and the relationship connecting them. The functions introduced below identify such structures. Again, they will return a set of sets with each inner set containing one element, its neighbouring element(s) and the relationship(s) connecting them.

- $ElementsDirectlyRelated(X_1, X_2) \subseteq P(E)$ is provided with two sets of elements X_1 and X_2 . Each inner set of the result set contains one element of X_1 , its adjacent elements of X_2 , and the connecting undirected relationships of R .
 - $ElementsDirectlyRelated(X_1, X_2) = \{EDR(x_1, X_2)\}$ with $x_1 \in X_1$ and $EDR(x_1, X_2) = \{x_2 \in X_2, z \in R_u \mid (x_1, z) \in S_u \wedge (x_2, z) \in D_u \vee (x_2, z) \in S_u \wedge (x_1, z) \in D_u\} \cup \{x_1\}$
- $AdjacentSuccessors(X_1, X_2) \subseteq P(E)$ is provided with two sets of elements X_1 and X_2 . Each inner set of the result set contains one element of X_1 , its adjacent, succeeding elements of X_2 , and the connecting directed relationships of R .
 - $AdjacentSuccessors(X_1, X_2) = \{AS(x_1, X_2)\}$ with $x_1 \in X_1$ and $AS(x_1, X_2) = \{x_2 \in X_2, z \in R_d \mid (x_2, z) \in D_d \wedge (x_1, z) \in S_d\} \cup \{x_1\}$

The fourth class contains functions that allow for defining recursive structures representing paths of arbitrary length (R1.5). Element paths are common substructures in many patterns. Consider, for example, the separation of duties pattern in Fig. 2 or the deadlock pattern in Fig. A5. In all of these cases, paths that start and end in particular elements have to be determined. Furthermore, GMQL offers functions that determine paths containing or not containing particular elements. Consider, for example, the mandatory/forbidden activities pattern in Fig. A2 that requires particular elements to be contained or not contained on the paths. In addition, loop functions return paths that start and end in the same element. Again, functions of the fourth class return a set of sets with each inner set containing one element path. For the specification of path structures, we make use of sequences of the form $(x_i) = (x_1, x_2, \dots, x_n)$, $x_i \in E$. However, as our functions generally operate on sets, we need a way to transform sequences into sets. Therefore, we define an auxiliary function $\text{Set}((x_i))$ taking a sequence as an input and returning the set containing all members of this sequence: $\text{Set}((x_i)) = \{x_i \in E \mid x_i \in (x_i)\} \subseteq E$.

- $\text{Paths}(X_1, X_n) \subseteq P(E)$ takes two sets of elements as input. It calculates all undirected paths from all elements of X_1 to all elements of X_n . Each inner set contains one path from one element of X_1 to one element of X_n .
 - $\text{Paths}(X_1, X_n) = U_{x_1 \in X_1, x_n \in X_n} \text{PX}(x_1, x_n)$ with $\text{PX}(x_1, x_n) = \{\text{Set}((x_1, x_2, \dots, x_n)) \mid x_2, \dots, x_{n-1} \in E \wedge ((x_i, x_{i+1}) \in S_u \vee (x_i, x_{i+1}) \in D_u) \forall 1 \leq i < n\}$

We first define a function PX returning all undirected paths, each as a set, starting with a single, particular element $x_1 \in X_1$ and leading to a single, particular element $x_n \in X_n$. Every sequence (x_1, \dots, x_n) containing elements x_1, \dots, x_n being pairwise related (i.e., x_i and x_{i+1} have a source or a target relation) is recognized as a path. Every sequence representing a path is transformed into a set. The result of PX is a set containing sets, which contain all paths leading from x_1 to x_n . Finally, the function PX is executed for every combination of every $x_1 \in X_1$ and $x_n \in X_n$. The resulting outer sets are unified, so the result is a set containing sets of all paths found.

- $\text{DirectedPaths}(X_1, X_n) \subseteq P(E)$ is the directed counterpart of Paths. It returns only paths containing directed relationships of the same direction leading from each element of X_1 to each element of X_n . Each such path found is represented by an inner set.
 - $\text{DirectedPaths}(X_1, X_n) = U_{x_1 \in X_1, x_n \in X_n} \text{DPX}(x_1, x_n)$ with
 - $\text{DPX}(x_1, x_n) = \{\text{Set}((x_1, x_2, \dots, x_n)) \mid x_2, \dots, x_{n-1} \in E \wedge$
 - $((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i+2}) \in D_d \forall 1 \leq i \leq \lfloor n/2 \rfloor) \quad (a)$
 - $\vee ((x_{2i}, x_{2i+1}) \in D_d \wedge (x_{2i+2}, x_{2i+3}) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor) \quad (b)$
 - $\vee ((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i+2}) \in D_d \wedge (x_{n-1}, x_n) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1) \quad (c)$
 - $\vee ((x_{2i}, x_{2i+1}) \in D_d \wedge (x_{2i+2}, x_{2i+3}) \in S_d \wedge (x_n, x_{n-1}) \in D_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1) \quad (d)$
 - $\forall 1 \leq i < n\}$

To specify directed paths, we first define a function DPX returning all directed paths, each as a set, starting with a single, particular element $x_1 \in X_1$ and leading to a single, particular element $x_n \in X_n$. Every sequence (x_1, \dots, x_n) containing objects that are related to relationships having the same direction (i.e., leading from x_1 to x_n), is recognized as a directed path. The formal definition of DPX is divided into four sections (a–d). This is necessary as the DPX function takes elements (not only objects) as inputs. Consequently, a path can either start with an object or with a relationship. Alternatively, a path can end with an object or with a relationship (cf. Fig. 8). For example, a path starting with an object and ending with an object requires a “source” relationship between its first two elements, followed by alternating “source” and “destination” relationships and ending with a “destination” relationship between its last two elements (cf. part (a) of the definition; parts (b)–(d) are defined analogously). Every sequence representing a directed path is transformed into a set. The result of DPX is a set containing sets, which in turn contain the directed paths leading from x_1 to x_n . Finally, the function DPX is

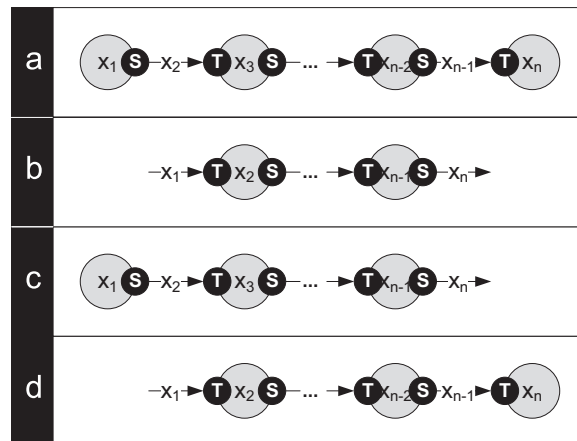


Fig. 8. Path configurations with respect to their start and target elements (adapted from [17]).

executed for every combination of every $x_1 \in X_1$ and $x_n \in X_n$. The resulting outer sets are unified, so the result is a set containing sets of all directed paths found.

- $Loops(X) \subseteq P(E)$ takes a set of elements as input and returns a set of sets containing all paths, which lead from any element of X back to itself. The direction of relations is handled analogously to Paths. Each inner set contains one loop.
 - $Loops(X) = U_{x \in X} PX(x, x)$
- $DirectedLoops(X) \subseteq P(E)$ takes a set of elements as input and returns a set of sets containing all directed paths, which lead from any element of X back to itself. The direction of relations is handled analogously to DirectedPaths. Each inner set contains one loop.
 - $DirectedLoops(X) = U_{x \in X} DPX(x, x)$
- $ShortestPaths(X_1, X_n)$ performs a path search as described above, but will only return the shortest paths found.
 - $ShortestPaths(X_1, X_n) = \min_{x \in Paths} |x|$
- $LongestPaths(X_1, X_n)$ performs a path search as described above, but will only return the longest paths found.
 - $LongestPaths(X_1, X_n) = \max_{x \in Paths} |x|$
- $ShortestDirectedPaths(X_1, X_n)$ performs a path search as described above, but will only return the shortest directed paths found.
 - $ShortestDirectedPaths(X_1, X_n) = \min_{x \in DirectedPaths} |x|$
- $LongestDirectedPaths(X_1, X_n)$ performs a path search as described above, but will only return the longest directed paths found.
 - $LongestDirectedPaths(X_1, X_n) = \max_{x \in DirectedPaths} |x|$
- $PathsContainingElements(X_1, X_n, X_c) \subseteq P(E)$ is provided with three sets of elements X_1, X_n and X_c . It returns a set of sets. Each inner set contains one path from one element of X_1 to one element of X_n . The path has to contain at least one element of X_c . The direction of relations is handled analogously to Paths.
 - $PathsContainingElements(X_1, X_n, X_c) = U_{x_1 \in X_1, x_n \in X_n} PCE(x_1, x_n, X_c)$ with
 $PCE(x_1, x_n, X_c) = \{Set((x_1, x_2, \dots, x_n)) | x_2, \dots, x_{n-1} \in E \wedge \exists x_c \in \{x_2, \dots, x_{n-1}\} \wedge ((x_i, x_{i+1}) \in S_u \vee (x_i, x_{i+1}) \in D_u) \forall 1 \leq i < n\}$

The definition of PCE is similar to that of PX. In addition, it forces the sequences returned to contain at least one element $x_c \in X_c$ (i.e., $\exists x_c \in \{x_2, \dots, x_{n-1}\}$).

- $DirectedPathsContainingElements(X_1, X_n, X_c) \subseteq P(E)$ is provided with three sets of elements X_1, X_n and X_c . It returns a set of sets. Each inner set contains one directed path from one element of X_1 to one element of X_n . The path has to contain at least one element of X_c . The direction of relations is handled analogously to DirectedPaths.
 - $DirectedPathsContainingElements(X_1, X_n, X_c) = U_{x_1 \in X_1, x_n \in X_n} DPCE(x_1, x_n, X_c)$ with
 $DPCE(x_1, x_n, X_c) = \{Set((x_1, x_2, \dots, x_n)) | x_2, \dots, x_{n-1} \in E \wedge \exists x_c \in \{x_2, \dots, x_{n-1}\} \wedge$
 $((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in D_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$
 $\vee ((x_{2i}, x_{2i-1}) \in D_d \wedge (x_{2i}, x_{2i+1}) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$
 $\vee ((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in D_d \wedge (x_{n-1}, x_n) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1)$
 $\vee ((x_{2i}, x_{2i-1}) \in D_d \wedge (x_{2i}, x_{2i+1}) \in S_d \wedge (x_n, x_{n-1}) \in D_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1) \forall 1 \leq i < n\}$

The definition of DPCE is similar to that of DPX. In addition, it forces the sequences returned to contain at least one element $x_c \in X_c$ (i.e., $\exists x_c \in \{x_2, \dots, x_{n-1}\}$).

- $PathsNotContainingElements(X_1, X_n, X_c) \subseteq P(E)$ is provided with three sets of elements X_1, X_n and X_c . It returns a set of sets. Each inner set contains one path from one element of X_1 to one element of X_n . The path must not contain any element of X_c . The direction of relations is handled analogously to Paths.
 - $PathsNotContainingElements(X_1, X_n, X_c) = U_{x_1 \in X_1, x_n \in X_n} PNCE(x_1, x_n, X_c)$ with $PNCE(x_1, x_n, X_c) = \{Set((x_1, x_2, \dots, x_n)) | x_2, \dots, x_{n-1} \in E \setminus X_c \wedge ((x_i, x_{i+1}) \in S_u \vee (x_i, x_{i+1}) \in D_u) \forall 1 \leq i < n\}$

The definition of PNCE is similar to that of PX. In addition, it forces the sequences returned not to contain any element $x_c \in X_c$ (i.e., $x_2, \dots, x_{n-1} \in E \setminus X_c$).

- $DirectedPathsNotContainingElements(X_1, X_n, X_c) \subseteq P(E)$ is provided with three sets of elements X_1, X_n and X_c . It returns a set of sets. Each inner set contains one directed path from one element of X_1 to one element of X_n . The path must not contain any element of X_c . The direction of relations is handled analogously to DirectedPaths.
 - $DirectedPathsNotContainingElements(X_1, X_n, X_c) = U_{x_1 \in X_1, x_n \in X_n} DPNCE(x_1, x_n, X_c)$ with
 $DPNCE(x_1, x_n, X_c) = \{Set((x_1, x_2, \dots, x_n)) | x_2, \dots, x_{n-1} \in E \setminus X_c$
 $\wedge (((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in D_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$
 $\vee ((x_{2i}, x_{2i-1}) \in D_d \wedge (x_{2i}, x_{2i+1}) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$
 $\vee ((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in D_d \wedge (x_{n-1}, x_n) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1)$
 $\vee ((x_{2i}, x_{2i-1}) \in D_d \wedge (x_{2i}, x_{2i+1}) \in S_d \wedge (x_n, x_{n-1}) \in D_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1) \forall 1 \leq i < n\}$

The definition of DPNCE is similar to that of DPX. In addition, it forces the sequences returned not to contain any element $x_c \in X_c$ (i.e., $x_2, \dots, x_{n-1} \in E \setminus X_c$).

- $\text{LoopsContainingElements}(X, X_c) \subseteq P(E)$ takes two sets of elements as input. It returns a set of sets. Each inner set contains one loop from one element of X back to itself. The loop must contain at least one element of X_c . The direction of relations is handled analogously to Paths.
 - $\text{LoopsContainingElements}(X, X_c) = U_{x \in X} \text{PCE}(x, x, X_c)$
- $\text{DirectedLoopsContainingElements}(X, X_c) \subseteq P(E)$ takes two sets of elements as input. It returns a set of sets. Each inner set contains one directed loop from one element of X back to itself. The loop must contain at least one element of X_c . The direction of relations is handled analogously to DirectedPaths.
 - $\text{DirectedLoopsContainingElements}(X, X_c) = U_{x \in X} \text{DPCE}(x, x, X_c)$
- $\text{LoopsNotContainingElements}(X, X_c) \subseteq P(E)$ takes two sets of elements as input. It returns a set of sets. Each inner set contains one loop from one element of X back to itself. The loop must not contain any element of X_c . The direction of relations is handled analogously to Paths.
 - $\text{LoopsNotContainingElements}(X, X_c) = U_{x \in X} \text{PNCE}(x, x, X_c)$
- $\text{DirectedLoopsNotContainingElements}(X, X_c) \subseteq P(E)$ takes two sets of elements as input. It returns a set of sets. Each inner set contains one directed loop from one element of X back to itself. The loop must not contain any element of X_c . The direction of relations is handled analogously to DirectedPaths.
 - $\text{DirectedLoopsNotContainingElements}(X, X_c) = U_{x \in X} \text{DPNCE}(x, x, X_c)$

By nesting the functions introduced above, it is possible to build up pattern queries successively. The results of each function can be reused by adopting them as an input for other functions. In order to combine intermediary results, GMQL additionally offers a set of operators (R2). Consider, again, the example of the two receipt structures in Fig. 5. In order to search for occurrences of either of these structures in a set of models, sub-queries have to be defined for each structure. These sub-queries then need to be combined to represent a query that will return occurrences of either structure. The operators provided by GMQL fall into three classes. The first class contains the common set operators UNION (\cup), INTERSECTION (\cap) and COMPLEMENT (\setminus). It furthermore includes the JOIN operator performing a particular union operation. They all expect two input parameters. These parameters are either both simple sets or both sets of sets. The JOIN-operator performs a union operation on the two input sets if these sets have at least one element in common. In case the input parameters are sets of sets, each inner set of the first parameter is compared to each inner set of the second parameter to determine if the respective sets have elements in common. Note that in the following a simple set of elements will be noted X_i , whereas a set of sets will be denoted \bar{X}_i . Each element of such a set of sets is again a set of elements and will be denoted \bar{x}_i . These constructs allow for defining the semantics of the operators as follows:

- $\text{JOIN}(X_1, X_2) = X_1 \cup X_2 \mid X_1 \cap X_2 \neq \emptyset$
- $\text{JOIN}(\bar{X}_1, \bar{X}_2) = \{\bar{x}_1 \cup \bar{x}_2 \mid \bar{x}_1 \cap \bar{x}_2 \neq \emptyset\}$

The second class contains operators that perform operations on all combinations of inner sets. This allows for further refining the result sets returned by the functions (see application example in the following section). The operators of this class can only be called with at least one set of sets. This means that only one of the two input parameters may be a simple set. The class contains the INNERINTERSECTION and INNERCOMPLEMENT operators. As the JOIN operator already takes into account the elements of inner sets, an INNERJOIN would be redundant and is thus not included in GMQL. Following the logic of the INNER-operators, an INNERUNION would calculate all unified sets of all combinations of inner sets. As we cannot think of a sensible application scenario for such an operation, it is not included in GMQL. The INNERCOMPLEMENT operator performs a minus operation on each combination of inner sets. As the complement operation is not commutative the INNERCOMPLEMENT requires the first parameter to be a set of sets. In case the second parameter is a simple set, the elements of that second parameter will be subtracted from all inner sets of the first parameter. The INNERINTERSECTION operator performs an intersection on all combinations of inner sets.

- $\text{INNERINTERSECTION}(X_1, \bar{X}_2) = \{X_1 \cap \bar{x}_2\}$
- $\text{INNERINTERSECTION}(\bar{X}_1, \bar{X}_2) = \{\bar{x}_1 \cap \bar{x}_2\}$
- $\text{INNERCOMPLEMENT}(\bar{X}_1, X_2) = \{\bar{x}_1 \setminus X_2\}$
- $\text{INNERCOMPLEMENT}(\bar{X}_1, \bar{X}_2) = \{\bar{x}_1 \setminus \bar{x}_2\}$

The third class contains operators that only take one set of sets as input and transform it into a simple set. These operators are necessary, because most functions expect simple sets as input. The class provides the SELFUNION and SELFINTERSECTION operators. As the input set may contain more than two inner sets, a SELFCOMPLEMENT is not feasible, because a minus operation expects exactly two input parameters and there is no sensible way of deciding which inner sets should be taken into account. The same argument can be extended to a SELFJOIN operation. The SELFUNION operator, however, unifies all inner sets into one simple set. The SELFINTERSECTION operator is defined analogously: It performs an

INTERSECTION operation on all inner sets of a set of sets successively. The result is a set containing elements that occur in all inner sets of the original set.

- $\text{SELFUNION}(\bar{X}_i) = \bigcup_{i=1}^n \bar{X}_i, \bar{X}_i \in \bar{X}$
- $\text{SELFINTERSECTION}(\bar{X}_i) = \bigcap_{i=1}^n \bar{X}_i, \bar{X}_i \in \bar{X}$

3.3. GMQL syntax

Having introduced the GMQL functions and operators in the previous section, we can now describe the GMQL syntax using EBNF statements as specified in the corresponding ISO standard [35]. As it will be demonstrated in Section 4, the GMQL functions and operators can be nested to construct a pattern query. In doing so, a query exhibits a tree structure with the result of one function or operator call serving as input for the next higher level. On root level a `queryExpression` consists of two parts. The first part is a `subQueryExpression` which can either be a `functionExpression`, an `operatorExpression` or a `setExpression` as further specified in the EBNF statements below. The second part contains all variable equations. If functions contained in a query take variables as input instead of concrete values for element types, numbers, etc., these variable equations can be defined to compare variable occurrences to one another (see separation of duties query in Section 2.2 and R1.3). A query may contain zero to n equations that are separated using a comma.

```
queryExpression=subQueryExpression{" " equationExpression};
subQueryExpression=(functionExpression|operatorExpression|setExpression);
```

A `functionExpression` consists of a `functionIdentifier` followed by an opening bracket. As explained above a function has one, two or three inputs. The first input is always another set specification, thus another `subQueryExpression`. The second and third inputs are either `parameterExpressions` as defined in the EBNF statement below or further sub-query expressions. Inputs are separated using commas. A `functionExpression` ends with a closing bracket.

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

An `operatorExpression` starts with an `operatorIdentifier` followed by an opening bracket. Operators can have either one or two inputs. These inputs need to be set specifications. Therefore, they can be represented by sub-query expressions. Inputs are separated using commas. An `operatorExpression` ends with a closing bracket.

```
operatorExpression=operatorIdentifier "(" subQueryExpression [" " subQueryExpression] ")";
```

A `setExpression` represents the basic input sets of the GMQL matching algorithm. A `setExpression` can be the terminal expressions “E”, “O”, or “R” representing the set of all model elements, the set of all model objects, or the set of all model relationships.

```
setExpression=("E"|"O"|"R");
```

A `parameterExpression` represents all inputs for functions that are not set specifications. Such inputs represent numbers, element type specifications, attribute data types and values as well as variables. The former are represented using Integer values, the latter three are represented as String values. Note that we did not further decompose the non-terminals `Integer` and `String`. They can be instantiated as any number or any string representing the name of an element type (e.g., “EPC-function”, “BPMN-task”, or “ERM-entity type”) or the value of an attribute (e.g., “Check invoice”). If the string contains an asterisk (“*”), it is interpreted as a wildcard that can be replaced by any substring (see application examples in Section 4). An attribute may be of a data type as indicated by the terminal values given below. A `parameterExpression` can furthermore be a `Variable` as captured by the EBNF statement below.

```
parameterExpression=(Integer | ElementType | AttributeDataType | AttributeValue | Variable);
ElementType=String;
AttributeDataType=("INTEGER" | "STRING" | "BOOLEAN" | "ENUM" | "DOUBLE");
AttributeValue=String;
```

An `equationExpression` consists of a first `Variable`, one of the comparison operators given below and a second `Variable`.

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

A `functionIdentifier` consists of one of the terminal values given below. A `functionIdentifier` represents the name of a particular function. The terminal values correspond to all function names introduced in Section 3.2.

```
functionIdentifier=("ElementsOfType" |
"ElementsWithAttributeOfValue" | "ElementsWithRelations" |
"ElementsWithSuccRelations" | "ElementsWithPredRelations" |
"ElementsWithRelationsOfType" | "ElementsWithSuccRelationsOfType" | "ElementsWithPredRelationsOfType" |
"ElementsWithNumberOfRelations" | "ElementsWithNumberOfSuccRelations" |
"ElementsWithNumberOfPredRelations" |
"ElementsWithNumberOfRelationsOfType")
```



```

"ElementsWithNumberOfSuccRelationsOfType" |
"ElementsWithNumberOfPredRelationsOfType" |
"ElementsDirectlyRelated" | "AdjacentSuccessors" | "Paths" |
"DirectedPaths" | "Loops" | "DirectedLoops" | "ShortestPaths" | "LongestPaths" | "ShortestDirectedPaths" |
"LongestDirectedPaths" | "PathsContainingElements" | "DirectedPathsContainingElements" |
"PathsNotContainingElements" | "DirectedPathsNotContainingElements" | "LoopsContainingElements" |
"DirectedLoopsContainingElements" | "LoopsNotContainingElements" |
"DirectedLoopsNotContainingElements");

```

The same holds for an `operatorIdentifier`. It represents the name of an operator.

```

operatorIdentifier=( "UNION" | "INTERSECTION" | "COMPLEMENT" | "JOIN" | "INNERINTERSECTION" | "INNERCOMPLEMENT" |
"SELFUNION" | "SELFINTERSECTION");

```

Lastly, a `Variable` has an identifier which can be any capital letter or combination of capital letters and a type. The type defines for which input this variable is used. An `"Integer"` can, for instance, be used to represent a number input as needed by the function `"ElementsWithNumberOfRelations"`. Variable name and type are separated using a comma and encapsulated by brackets.

```

Variable="( (" "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
"U" | "V" | "W" | "X" | "Y" | "Z" ) + ", " ("Integer" | "ElementType" | "AttributeDataType" | "AttributeValue" ) )";

```

4. Application examples

In this section we present one exemplary pattern query for each of the model analysis scenarios presented in [Section 2](#).

4.1. Business process weakness detection

To demonstrate the applicability of GMQL in the context of business process weakness detection, consider the process weakness of switching between manual and automatic activities presented in [Section 2.1](#). [Fig. 1](#) contains exemplary pattern occurrences. In informal English, the pattern in [Fig. 1\(a\)](#) can be described as follows: “Find paths of activities that start in a building block labelled ‘Print’ and end in a building block labelled ‘Scan’”. The pattern query given below returns corresponding model fragments. The call to the `DirectedPaths` function takes two sets of elements as input. These sets are built up analogously. They represent objects of type “BuildingBlock” that are labelled either “Print” (line 03) or “Scan” (line 05). Building blocks are the activity types specified in the SBPML process modelling language (see [\[3,22\]](#)). Labels are represented as attributes that need to have a particular value. Therefore, the `ElementsWithAttributeOfValue` function is called (lines 02 and 04). The two resulting sets are fed to the `DirectedPaths` function in order to determine all paths that start in a print activity and end in a scan activity (line 01).

```

01: DirectedPaths(
02:   ElementsWithAttributeOfValue(
03:     ElementsOfType(O,BuildingBlock), Label, 'Print'),
04:   ElementsWithAttributeOfValue(
05:     ElementsOfType(O,BuildingBlock), Label, 'Scan'))

```

4.2. Business process compliance checking

In informal English, the structure of the separation of duties pattern (cf. [Section 2.2](#)) can be described as follows: “Find an element path in the process model that starts in a task object labelled like ‘check*’ and ends in a task object labelled like ‘verify*’”. The start and target objects of the path have to be directly adjacent to an organizational unit. In the chart these units must be on the same level.” In order for the query to work not just on a loan application process, but on any process that requires a check activity to be performed by a different organizational unit than the succeeding verification task, the label values include an asterisk that can be replaced by any substring (e.g., “loan application”, “inventory list”, etc.).

The GMQL query below returns the structures highlighted in [Fig. 2](#). It consists of two components, namely the structure contained in the organizational chart (lines 02–10) and the structure contained in the process model (lines 11–26). Both structures are joined (line 01). The structure contained in the organizational chart consists of two substructures. Both contain two directly related organizational units (lines 03–06 and lines 07–10). These two substructures are joined (line 02) in order to get the overall model fragment. The first unit contained in the two substructures furthermore carries a label whose value in turn carries a variable (lines 05 and 09).

As far as the structure in the process model is concerned we are interested in finding particular paths. The two input parameters (lines 12–18 and 19–25) of the call to the `DirectedPaths` function (line 11) are built up analogously. They both return task objects that have a particular label and are directly related to objects of type organizational unit. Let us consider the first parameter (lines 12 and 18). In particular, we are interested in elements of type “OrgaUnit” that have an attribute called “Label” that is of value C (lines 14 and 15). Note that C represents a variable that holds a string value in this case. This means that we are not interested in a particular attribute value; rather we are interested

in comparing the label values of different objects of type “OrgaUnit”. The organizational units need to have adjacent elements that are of type “Task” and have the label “check*” (lines 16 and 17). The set of organizational units and the set of task objects are fed to the *ElementsDirectlyRelated* function (line 13) in order to determine all task objects that are adjacent to organizational units. The result set is fed to the INNERINTERSECTION operator (line 12) along with the set of all task objects (line 18) in order to cut away the organizational units and the relationship between the task node and the organizational unit. The resulting set of sets contains inner sets each holding one task object labelled “check*”. As the *DirectedPaths* function expects simple sets as input, this set of sets is fed to the SELFUNION operator in order to turn it into a simple set. The second input parameter for *DirectedPaths* (line 11) is calculated analogously (lines 19–25). The only difference is that we are interested in (a) tasks that have a label of value “verify*” (lines 23 and 24) and (b) organizational units with a label *D*. Again, *D* represents a variable for a string value. The resulting two sets of activities are finally fed to the *DirectedPaths* function in order to determine all possible paths between them (line 11).

Lastly, we are interested in comparing the label values of the organizational units in the process model (Fig. 2(b)) to the label values of the units contained in the chart (Fig. 2(a)). To that end, the query contains two equations (lines 26 and 27) claiming that variable *A* has to have the same value as variable *C*. In addition, variable *B* has to have the same value as variable *D*. These equations allow for creating a relationship between the organizational units contained in the process model and those units contained in the corresponding chart. In summary, this query thus returns paths from task objects having the label “check*” to task objects having the label “verify*”. The start and target elements of the paths are furthermore directly related to organizational units that are on the same level in the organizational hierarchy as depicted by the corresponding chart.

```

01: JOIN(
02:   JOIN(
03:     AdjacentSuccessors(
04:       ElementsWithAttributeOfValue(
05:         ElementsOfType(O,OrgaUnit), Label, (A,AttributeValue)),
06:       ElementsOfType(O,OrgaUnit)),
07:     AdjacentSuccessors(
08:       ElementsWithAttributeOfValue(
09:         ElementsOfType(O,OrgaUnit), Label, (B,AttributeValue)),
10:       ElementsOfType(O,OrgaUnit))),
11:   DirectedPaths(
12:     SELFUNION(INNERINTERSECTION(
13:       ElementsDirectlyRelated(
14:         ElementsWithAttributeOfValue(
15:           ElementsOfType(O,OrgaUnit), Label, (C,AttributeValue)),
16:         ElementsWithAttributeOfValue(
17:           ElementsOfType(O,Task), Label, 'check*')),
18:       ElementsOfType(O,Task))),
19:     SELFUNION(INNERINTERSECTION(
20:       ElementsDirectlyRelated(
21:         ElementsWithAttributeOfValue(
22:           ElementsOfType(O,OrgaUnit), Label, (D,AttributeValue)),
23:         ElementsWithAttributeOfValue(
24:           ElementsOfType(O,Task), Label, 'verify*')),
25:       ElementsOfType(O,Task))))),
26:   (A,AttributeValue)=(C,AttributeValue),
27:   (B,AttributeValue)=(D,AttributeValue)

```

4.3. Model translation

Fig. 3 contains the WHILE pattern reported in [25] representing a BPMN structure that can easily be translated to BPEL code. In informal English, a corresponding pattern query can be described as follows: “Find all element paths of arbitrary length that start and end in the same joining XOR gateway.” The pattern query given below returns such a structure. It makes use of the *DirectedLoops* function (line 01). This function is called with one set of elements that represent the start and target elements of the loop. The input parameter of this call represents the set of all joining XOR gateways. This set is calculated by subtracting all XOR objects that have exactly one or exactly zero ingoing edges from the set of all XOR objects. What is left is the set of all XOR objects that have at least two ingoing edges, hence all XOR joins. This is achieved by first determining all XOR objects that have no ingoing edges (lines 10 and 11). The result is a set of sets with each inner set containing one XOR object. It is fed to the SELFUNION operator in order to turn it into a simple set (line 09). Next, the set of all XOR objects that have exactly one ingoing edge is calculated (lines 07 and 08). The resulting set is inner-intersected with the set of all XOR objects (lines 05 and 06) in order to cut away the edge within each inner set. What is left is a set of sets with each inner set containing exactly one XOR object. Again, this set is fed to the SELFUNION operator in order to turn it into a simple set (line 05). Hence, we get two simple sets. The first one contains all XOR objects having one ingoing edge; the second one contains all XOR objects with no ingoing edges. The two sets are unified (line 04) and subtracted (line 02) from the set of all XOR objects (line 03). Consequently, the set of all XOR objects is returned that have

at least two ingoing edges. It is fed to the *DirectedLoops* function (line 01) to determine paths that start and end in the same object.

```

01: DirectedLoops(
02:   COMPLEMENT(
03:     ElementsOfType(O,XOR) ,
04:     UNION(
05:       SELFUNION(INNERINTERSECTION(
06:         ElementsOfType(O,XOR) ,
07:         ElementsWithNumberOfPredRelations(
08:           ElementsOfType(O,XOR),1) ) ) ,
09:       SELFUNION(
10:         ElementsWithNumberOfPredRelations(
11:           ElementsOfType(O,XOR),0) ) ) ) ) )

```

4.4. Syntactical correctness checking

In terms of syntactical correctness checking of conceptual models, consider [27] who identifies a set of syntax errors in EPC models. One of such errors represents the XOR-control structure depicted in Fig. A5 (structure highlighted in bold). In informal English a corresponding pattern query can be described as follows: “Find element paths of arbitrary length that start in an XOR split and end in an AND join. The AND join furthermore has to be succeeding an EPC start event.” Such a structure can cause problems at model runtime. If the start event fires, process execution will halt at the AND join. The AND join will only conclude, if all ingoing branches are executed. Considering the EPC model depicted in Fig. A5 this might not happen, if the process instance ran into the upper XOR branch. A workflow management system that implements this process thus has to contain measures to resolve such a situation. One way to do so might be to terminate process execution after a timeout occurred.

The pattern query given below returns XOR structures as depicted in Fig. A5. In terms of the GMQL constructs, this pattern represents a path from an XOR split (lines 02–12) to an AND join that directly succeeds an event that has no ingoing edges (lines 13–29). The XOR split is calculated by subtracting the set of XORs that have either one or zero outgoing edges from the set of all XOR objects. What is left is the set of all XOR objects that have at least two outgoing edges, ergo all XOR splits. This is achieved by calling *ElementsWithNumberOfSuccRelations* with the set of all XOR objects and the integers 1 (lines 07 and 08) and 0 (lines 11 and 12). The corresponding output sets are then inner-intersected with the set of all XOR objects (lines 05 and 09) in order to remove the edges from these sets. In doing so, the paths that need to be calculated later on will only start in XOR objects. The inner-intersection leads to an intermediary result set consisting of inner sets each containing one XOR object. The SELFUNION operator (lines 05 and 09) turns this set of sets into a simple set. The UNION operator (line 04) then unifies these sets and feeds them to the COMPLEMENT call.

The second parameter of the *DirectedPaths* call (lines 13–29) is calculated by first determining the AND joins in a similar way the XOR splits were calculated (lines 19–29). The result set is then fed to the *AdjacentSuccessors* function (line 15) to determine all AND joins that directly succeed an event with no ingoing edges (lines 16–18). The resulting set is inner-intersected (line 13) with the set of all AND objects (line 14) to remove the events and the connecting edges from that set. As the *DirectedPaths* call expects simple sets as input the SELFUNION operator is called to turn the resulting set of sets into a simple set (line 13). The result is then fed to the *DirectedPaths* function (line 01) to determine all paths from a splitting XOR to a joining AND that is succeeding a start event.

```

01: DirectedPaths(
02:   COMPLEMENT(
03:     ElementsOfType(O,XOR) ,
04:     UNION(
05:       SELFUNION(INNERINTERSECTION(
06:         ElementsOfType(O,XOR) ,
07:         ElementsWithNumberOfSuccRelations(
08:           ElementsOfType(O,XOR),1) ) ) ,
09:       SELFUNION(INNERINTERSECTION(
10:         ElementsOfType(O,XOR) ,
11:         ElementsWithNumberOfSuccRelations(
12:           ElementsOfType(O,XOR),0) ) ) ) ) ,
13:   SELFUNION(INNERINTERSECTION(
14:     ElementsOfType(O,AND) ,
15:     AdjacentSuccessors(
16:       INNERINTERSECTION(O,
17:         ElementsWithNumberOfPredRelations(
18:           ElementsOfType(O,Event),0) ) ,
19:       COMPLEMENT(
20:         ElementsOfType(O,AND) ,
21:         UNION(
22:           SELFUNION(INNERINTERSECTION(

```

```

23:         ElementsOfType(O,AND) ,
24:         ElementsWithNumberOfPredRelations(
25:             ElementsOfType(O,AND),1))) ,
26:     SELFUNION (INNERINTERSECTION(
27:         ElementsOfType(O,AND) ,
28:         ElementsWithNumberOfPredRelations(
29:             ElementsOfType(O,AND),0)))))))))

```

4.5. Model comparison

The GMQL query given below returns the ERM substructure depicted in Fig. 5(b). In informal English this query can be described as follows: “Find all occurrences of a relationship type that is labelled ‘Invoice Position’ and that is adjacent to exactly three entity types that are labelled either ‘Product’, ‘Time’, or ‘Customer’”. The query depicted below therefore makes use of the *ElementsDirectlyRelated* function (line 03). To determine the corresponding set of all relationship types the *ElementsWithNumberOfRelations* function is called with the set of all relationship types that are labelled “Invoice Position” (lines 04–06) and the integer value 3 (line 07). In doing so, all appropriately labelled relationship type objects are calculated that have exactly three relationships. This set is inner-intersected with the set O of all objects in order to cut away the relationships (line 02). The result is fed to the SELFUNION operator (line 02) in order to turn the set of sets into a single set of elements. The second parameter represents the unified set of all entity types that are labelled either “Product”, “Time”, or “Customer” (lines 08–14). Note that the UNION operator requires exactly two input parameters. Consequently, two calls to UNION need to be nested in order to unify three sets of elements (line 08). Finally, the returned sets are fed to the *ElementsDirectlyRelated* function (line 01) in order to determine all relationship types that have exactly three adjacent entity types.

```

01: ElementsDirectlyRelated(
02:     SELFUNION (INNERINTERSECTION(O,
03:         ElementsWithNumberOfRelations(
04:             ElementsWithAttributeOfValue(
05:                 ElementsOfType(O,RelationshipType) ,
06:                 Label, 'Invoice Position')
07:             3))) ,
08:     UNION(UNION(
09:         ElementsWithAttributeOfValue(
10:             ElementsOfType(O,EntityType) , Label, 'Product') ,
11:         ElementsWithAttributeOfValue(
12:             ElementsOfType(O,EntityType) , Label, 'Time') ,
13:         ElementsWithAttributeOfValue(
14:             ElementsOfType(O,EntityType) , Label, 'Customer'))))

```

5. Implementation

We implemented GMQL as a plugin for a meta-modelling tool that was available from a previous research project [36]. Being a meta-modelling tool, it provides two main features: first, it allows for specifying modelling languages. Second, it allows for developing models using the previously defined languages. The tool is also based on the concept outlined in Fig. 6. A language’s syntax is thus represented by a set of interrelated element types. These element types represent the basic constructs of a modelling language. They subcategorize into object types representing the node types of a model (e.g., node type “function” in an EPC) and relationship types representing the edge types. A model is created by instantiating the element types to concrete elements (i.e., objects and relationships).

The tool is based on a layer architecture consisting of a storage layer, a business layer, a presentation layer, and a plugin layer. The *storage layer* provides basic database management functionality to store, delete as well as update model and language data. The tool contains a relational database that provides tables for each model and language construct. This means that all constructs contained in Fig. 6 (i.e., element types, elements, etc.) are represented as tables in the database. The *business layer* provides functionality to develop and manage models as well as languages. The *presentation layer* contains a language editor and a graphical modelling editor as its main components. The former allows the user to develop languages in a two-step procedure. First, the user specifies the abstract syntax of the language by defining its element types and how they can be related. Second, the user specifies the concrete syntax by assigning visual shapes to object types as well as lines and arrows of different styles to relationship types. In case of relationship types that assign attributes to elements, the user can specify that the corresponding relationship type is not displayed as a line but is used to open an attribute menu. Shapes, lines and arrows can be defined in an additional editor. The modelling editor provides functionality to develop a model. Here, the user first selects the language the model is supposed to be developed in. The modelling editor then provides a list of shapes representing the object types of that language. To create a model these shapes can be dragged and dropped on a two-dimensional drawing canvas. If the user tries to draw a relationship between two elements that are not allowed to be connected, the user is informed that a corresponding relationship type does not exist. It is thus not possible to develop models that do not comply with the corresponding language specification. The *plugin layer* provides features to access the functionality of the previous three layers. Plugins provide additional functionality that can be included in or excluded from the tool’s setup as needed.

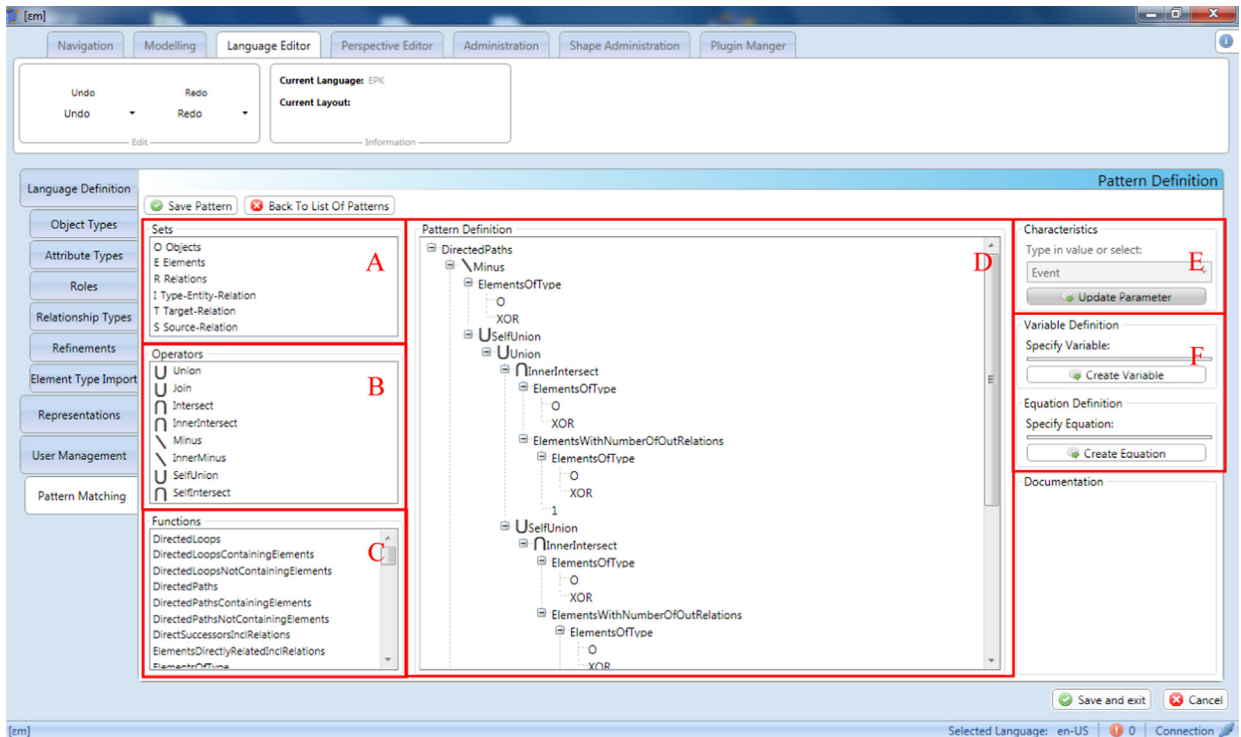


Fig. 9. Defining a model query.

GMQL is implemented as such a plugin. The plugin provides features to define and subsequently run a query on a collection of models. The functionality to define a query is integrated into the language editor of the meta-modelling tool. Fig. 9 contains a screenshot of this editor. The tabs contained on the left-hand side of the editor allow the user to specify the abstract and concrete syntax of the language, amongst others. The lowermost tab provides access to the GMQL functionality. To define a query the user first needs to select the language he wants to create a query for. This is reasonable, as for example, it does not make sense to search for a pattern containing activities in an ER model. To that end, the user is presented with a list of all modelling languages currently contained in the meta-modelling tool. Once a language is selected, the user is presented with a list of all queries that have already been created for this language. The user can choose between creating a new and editing an existing query. Finally, the query editor depicted in Fig. 9 is loaded.

The basic input sets for the matching algorithm (cf. highlighted subsection A of Fig. 9) and all GMQL operators (cf. subsection B) and functions (cf. subsection C) are presented on the left-hand side of the editor. To build up the pattern query the user can drag and drop these GMQL constructs onto the query definition field in the middle of the editor (cf. subsection D). The right-hand side provides features to update parameters (cf. subsection E) and to define variables and variable equations (cf. subsection F). To update a parameter the user can either enter a new or select a predefined value from a list. Variables represent numbers or string values. Numbers are required, for instance, as input for the `ElementsWithNumberOf{SuccPred}` Relations functions. String values can, for instance, be used to specify a particular label. Predefined values represent the element types of the particular modelling language the query is created for. The list of available element types is generated when the query editor is loaded. It contains the names of all element types that belong to the given language. To calculate this list the GMQL plugin accesses the database of the meta-modelling software in order to determine the set of element types the selected language contains. The names of these entity types are then mapped to corresponding string values that can be used in the query definition. This implementation of GMQL is generic in the sense that it allows for defining model queries for each and every modelling language that is contained in the meta-modelling tool. A particular query, however, is always related to a particular modelling language (see above) and thus contains element type information of one modelling language.

Once the user has defined a query, it can be run on one model or a set of models. Corresponding functionality is integrated into the model editor of the meta-modelling software. Fig. 10 depicts a screenshot of this editor. The list on the left-hand side contains the shapes of the language's element types that can be dragged and dropped onto the modelling canvas on the right. The user can access the GMQL search functionality using the interface in the top right corner of the editor (cf. subsection A). The user can trigger a new search run by clicking on the corresponding button. A window appears that contains a list of all GMQL queries that have so far been created for the given modelling language. After query selection the matching algorithm is triggered which returns all pattern occurrences contained in the given model. Each occurrence is highlighted. The user can browse through all returned results. The pattern occurrence highlighted in Fig. 10 is a result of running the query described in Section 4.4 on the depicted model. The navigation tab in Fig. 10 contains a list of all models

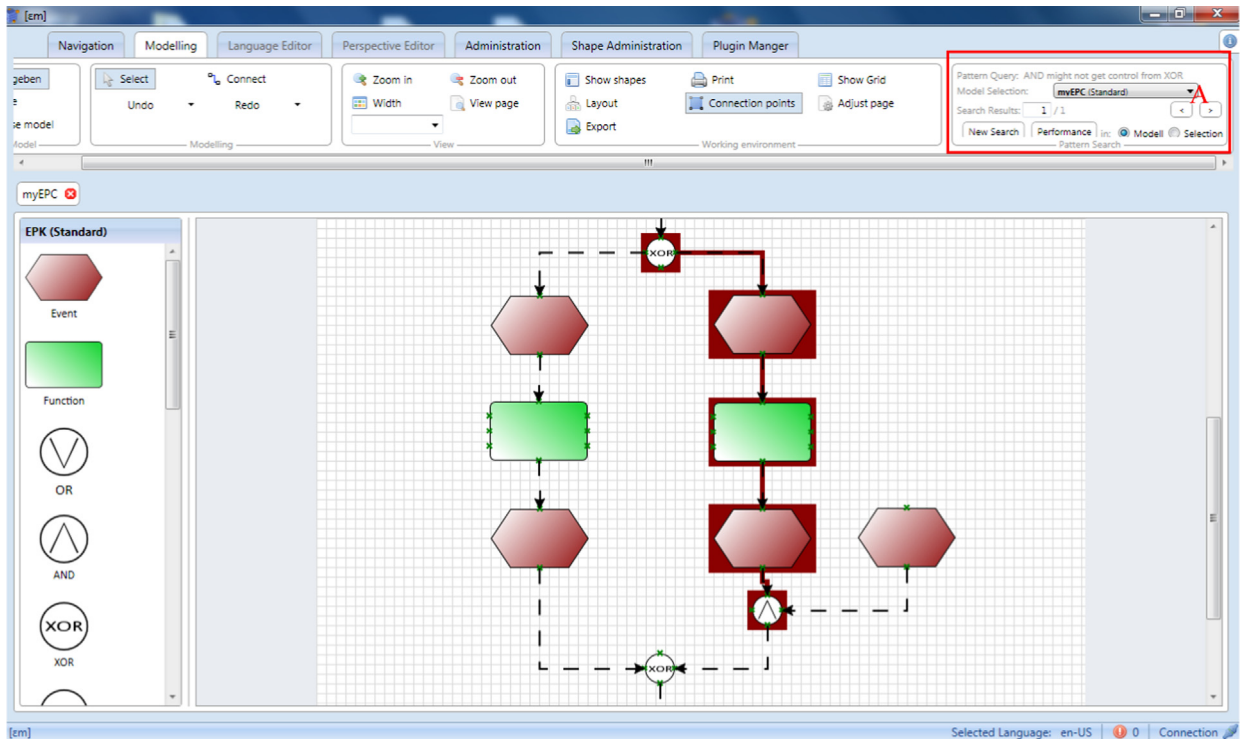


Fig. 10. Running a model query

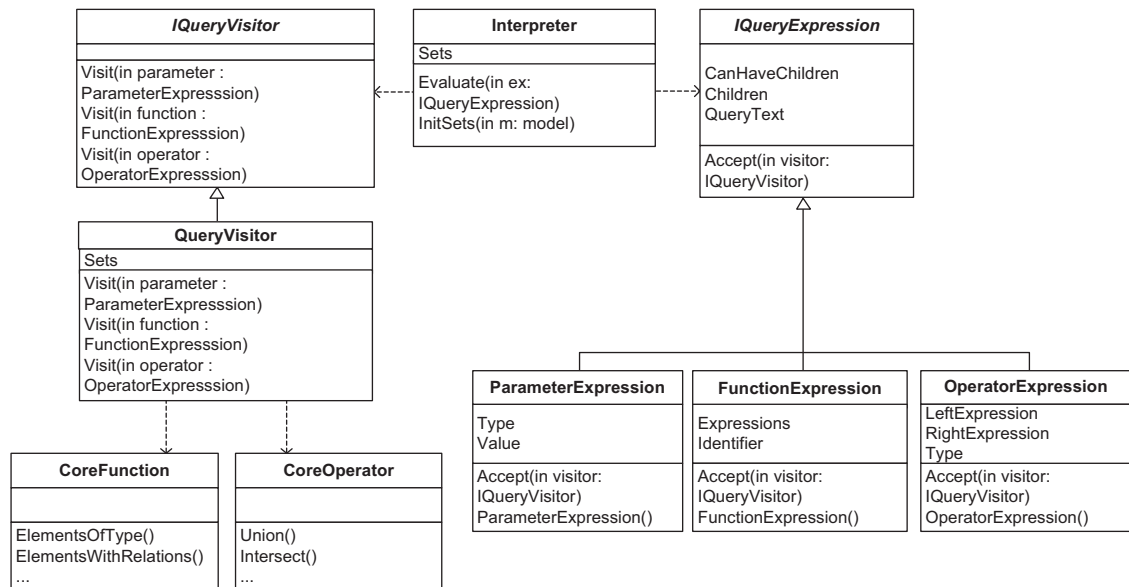


Fig. 11. UML class diagram of the matching algorithm.

that have so far been created. The user can also execute a pattern search on this list. To that end, the user is presented with a list of all queries that have been created for all languages for which models appear in the model list. A selected query is then run on all models of the corresponding language.

The GMQL matching algorithm (R3) is implemented using the visitor design pattern [37] (cf. Fig. 11). As outlined above, a pattern query is represented as a search tree that complies with the GMQL syntax. The matching algorithm traverses this tree in a bottom-up fashion calculating the leaf nodes of the tree first. The result set of a given tree level is then fed to the next higher level. A tree consists of query expression objects that can either be function, operator, or parameter expression objects. Once a search run is triggered, an instance of the interpreter class is created with the model and the query as input.

In a next step, the *InitSets()* method is called on the model to determine the basic input sets E , O and R . These sets are stored in the corresponding property of the interpreter object. Next, the *Evaluate()* method is called with the query expression as input. This method first creates a query visitor object. This object is instantiated with the previously calculated basic sets as input. *Evaluate()* then calls the *Accept()* method on the query expression with the visitor object as input. Depending on the kind of expression the *Accept()* method invokes the corresponding *Visit()* method of the visitor object. These methods determine the child elements of the given expression thereby traversing the query tree in a top down fashion. Each query expression stores its child elements and information about the number of children in respective attributes. The *CanHaveChildren* attribute is only set to false in case the query expression is a parameter expression. In this case, the visitor has reached the lowest level in the expression tree and query execution starts. The visitor thus traverses the query tree again, this time in a bottom-up fashion. The result of one given level is fed to the next higher level as input. In case a given query expression is indeed an operator or function expression, the corresponding *Visit()* methods access the static *CoreFunction* and *CoreOperator* classes providing implementations for each GMQL construct.

If the query contains variables and variable equations, possible values for these variables are determined from the particular model the query is executed on. For each combination of existing values a concrete pattern query is determined that replaces the variables with one combination of values for which the equations hold. The set of valid pattern queries is then executed (cf. [38] for more details).

The GMQL plugin furthermore contains a database to store pattern queries. This database consists of a single table storing information on the ID and name of a query as well as a string value representing the query expression. Once the user hits the save button in the query editor, the object tree that represents the query expression at runtime is parsed to a string value using the ANTLR parser.¹ This parsing procedure is based on the GMQL syntax specification described in Section 3.3. It allows for efficiently loading a query from the database (see more details on loading times in Section 6.2.4). The plugin can thus simply load a single string value that is then reparsed to the query expression at runtime.

6. GMQL runtime performance

6.1. Theoretical runtime complexity

In the following, we present the worst case theoretical runtime complexity of each function (cf. Table 1) and operator (cf. Table 2). The runtime complexity only includes the number of comparison operations. It does not include memory requirements. It furthermore assumes array list objects to be the underlying data structure representing sets. We will refer to n as the cardinal number of a simple input set of elements ($n = |e| \subseteq E$) and m as the cardinal number of a simple input set of relationships ($m = |r| \subseteq R$). The overall runtime complexity of a given pattern query consists of two components. The first component describes the cost of building up the basic input set E , O , and R . The second component defines the sum of all the complexities of the functions and operators the given pattern query contains. As far as the first component is concerned, the set E can be determined in $O(1)$ steps. When a model is loaded from the database, the meta-modelling tool returns a list of all elements that belong to the given model. This list essentially represents the set E . Note that we abstract

Table 1
Theoretical worst case runtime complexity of all functions.

Function	Complexity
<i>ElementsOfType</i> (X, t)	$O(n)$
<i>ElementsWithAttributeOfValue</i> (X, t_a, v)	$O(n * 2 * (sr + dr))$
<i>ElementsWithAttributeOfDataType</i> (X, w)	$O(n * (sr + dr))$
<i>ElementsWithPredSuccRelations</i> (X, Z)	$O(n * m)$
<i>ElementsWithPredSuccRelationsOfType</i> ($X, Z_{(d)}, t_R$)	$O(n * m + m)$
<i>ElementsWithNumberOfRelations</i> (X, n_x)	$O(n * (sro + dro))$
<i>ElementsWithNumberOfPredRelations</i> (X, n_x)	$O(n * dro)$
<i>ElementsWithNumberOfSuccRelations</i> (X, n_x)	$O(n * sro)$
<i>ElementsWithNumberOfRelationsOfType</i> (X, t_R, n_x)	$O(n * (sro + dro) + m)$
<i>ElementsWithNumberOfPredRelationsOfType</i> (X, t_R, n_x)	$O(n * dro + m)$
<i>ElementsWithNumberOfSuccRelationsOfType</i> (X, t_R, n_x)	$O(n * sro + m)$
<i>ElementsDirectlyRelations</i> (X_1, X_2)	$O(n_1 * n_2 * (sro_1 + dro_1))$
<i>AdjacentSuccessors</i> (X_1, X_2)	$O(n_1 * n_2 * sro_1)$
<i>{Directed}Paths</i> (X_1, X_n)	$O(E !)$
<i>{Directed}Paths{Not}ContainingElements</i> (X_1, X_n, X_c)	$O(E !)$
<i>{Directed}Loops</i> (X_1, X_n)	$O(E !)$
<i>{Directed}Loops{Not}ContainingElements</i> (X_1, X_n, X_c)	$O(E !)$
<i>{Shortest!Longest}{Directed}Paths</i> (X_1, X_n)	$O(E !)$

¹ <http://www.antlr.org/>.

Table 2

Theoretical worst case runtime complexity of all operators.

Operator	Complexity
JOIN (X_1, X_n)	Simple sets: $O(n_1 * n_n + n_1 + n_n)$ Sets of sets: $O(N_1 * N_n * (n_{1max} * n_{nmax} + n_{1max} + n_{nmax}))$
UNION (X_1, X_n)	Simple sets: $O(n_1 + n_n)$ Sets of sets: $O(N_1 * N_n * n_{1max} + N_n * n_{nmax})$
INTERSECTION (X_1, X_n)	Simple sets: $O(n_1 * n_n)$ Sets of sets: $O(N_1 * N_n * n_{1max} * n_{nmax})$
COMPLEMENT (X_1, X_n)	Simple sets: $O(n_1 * n_n)$ Sets of sets: $O(N_1 * N_n * n_{1max} * n_{nmax})$
INNERINTERSECTION (X_1, X_n)	Simple set and set of sets: $O(n_1 * n_{nmax} * N_n)$ Sets of sets: $O(N_1 * N_n * n_{1max} * n_{nmax})$
INNERCOMPLEMENT (X_1, X_n)	Simple set and set of sets: $O(N_1 * n_{1max} * n_n)$ Sets of sets: $O(N_1 * N_n * n_{1max} * n_{nmax})$
SELFUNION (X)	Set of sets: $O(N * n_{max})$
SELFINTERSECTION (X)	Set of sets: $O((N-1) * n_{max})^2$

from the particular cost of loading a model which is naturally highly specific to the given modelling tool used to implement GMQL and therefore outside the scope of the paper at hand. From the point of view of the worst-case complexity of *executing a query on a model*, the set E is essentially available after loading of the model is completed. The set E furthermore has to be iterated in order to determine which element is actually an object and which element is a relationship. This can be achieved in linear time.

The *ElementsOfType* function executes in linear time with n being the cardinal number of the input set, because it runs through all elements of the input set and determines their corresponding types. *ElementsWithRelations* takes a set of elements and a set of relationships as input. For each element of the first input set its relationships contained in the second input set are determined. This can be achieved in $O(n * m)$ steps. *ElementsWithPredRelations* and *ElementsWithSuccRelations* are of the same worst case runtime complexity.

ElementsWithRelationsOfType determines all elements of the first input set having relations of a predefined type. The function executes an *ElementsOfType* call with its second and third parameters. *ElementsWithRelations* is then called with the resulting set in combination with the first parameter. Consequently the complexity of *ElementsWithRelationsOfType* is the sum of the complexities of *ElementsOfType* and *ElementsWithRelations*. The same argument can be extended to *ElementsWithPredRelationsOfType* and *ElementsWithSuccRelationsOfType*.

ElementsWithNumberOfRelations is called with a set of elements and an integer. As indicated by Fig. 6 each element can be source and destination element of a set of relationships. These relationship sets have to be checked to determine all relationships of a given element. We refer to sro as the cardinal number of a set of relationships a given element is source of and dro as the cardinal number of a set of relationships a given element is destination of. Consequently, the overall complexity of *ElementsWithNumberOfRelations* in case the first parameter is a simple set is $O(n * (sro + dro))$. For *ElementsWithNumberOfPredRelations* only those relationships have to be considered that a given element is destination of. Consequently, the complexity of that function is $O(n * dro)$. *ElementsWithNumberOfSuccRelations* then is of complexity $O(n * sro)$. Analogously to *ElementsWithRelationsOfType*, the complexity of *ElementsWithNumberOfRelationsOfType* is the sum of the complexities of *ElementsWithNumberOfRelations* and *ElementsOfType*.

ElementsDirectlyRelated takes two sets of elements as input. If one element x_1 of the first set is directly related to an element x_2 of the second set, there has to be one relationship that has x_1 as a source and x_2 as a destination. This has to be checked for x_1 being the destination and x_2 being the source as well, because this function neglects the direction of the relationships. As this procedure has to be executed for every combination of elements of X_1 and X_2 , the overall complexity of this function is $O(n_1 * n_2 * (sro_1 + dro_1))$. This complexity improves to $O(n_1 * n_2 * sro_1)$ for *AdjacentSuccessors*, because this function does consider the direction of the relationships. As attributes are considered objects that are connected to their element via an undirected relationship, the complexities of *ElementsWithAttributeOfValue* and *ElementsWithAttributeOfDatatype* are similar to the complexities of the functions of the second class. As there can be at most $sr + dr$ attributes, the matching algorithm determines for each element of the input set if it is adjacent to one of the $sr + dr$ attributes. For the *ElementsWithAttributeOfValue* function, two additional comparison operations have to be executed to determine the attributes type and value. For the *ElementsWithAttributeOfDatatype* function in contrast, only one comparison operation is necessary to determine the attribute's data type.

The *paths* functions determine all possible paths from each element of the first input set to each element of the second input set. They perform a depth-first-search for each combination of start and target element. As each graph vertex can theoretically be connected to each other vertex, there are a potentially exponential number of paths to calculate. The worst case theoretical runtime performance of calculating all paths in a given graph is facultative in the number of graph elements (i.e., the cardinal number of the set E of all elements) [39].

As the operators can also take sets of sets as input, we denote N as the cardinal number of a set of sets. n_{max} will denote the maximum cardinal number of one inner set that is used to estimate the worst case runtime performance. The JOIN

operator performs a union operation on two sets if these sets have at least one element in common. A join consequently is executed in two consecutive steps. First, it is determined whether the two input sets have at least one element in common. This can be achieved in $O(n_1 * n_n)$ steps. Next, if indeed one element is contained in both input sets, the resulting output set will be created. This can be achieved in $O(n_1 + n_n)$ steps. The overall complexity of the JOIN operator in case of two simple input sets is then the sum of the two consecutive steps, hence $O(n_1 * n_n + n_1 + n_n)$. This complexity increases for sets of sets to $O(N_1 * N_n * (n_{1max} * n_{nmax} + n_{1max} + n_{nmax}))$. Analogously to the second step of the JOIN operator, the complexity of the UNION operator is $O(n_1 + n_n)$ for simple sets and $O(N_1 * n_{1max} + N_n * n_{nmax})$ for sets of sets.

The INTERSECTION operator determines for each element of the first set if it is also contained in the second set. This can be achieved in $O(n_1 * n_n)$ steps for the case of simple sets and $O(N_1 * N_n * n_{1max} * n_{nmax})$ steps for the case of sets of sets. The COMPLEMENT operator determines for each element of the first input set if it is also contained in the second set. If that is the case, the element is subtracted from the result set. As this last step can be performed in constant time, the worst case complexity of the COMPLEMENT operator is identical to that of the INTERSECTION operator.

The complexity of INNERINTERSECTION is similar to the complexity of INTERSECTION. However, as the operator can be called with a simple set and a set of sets, the complexity of INTERSECTION has to be multiplied by the cardinal number N of the set of sets. The same argument can be extended to the INNERCOMPLEMENT operator. In case both input parameters are sets of sets, the complexities of these two operators are identical to those of the respective operators that work only on outer sets.

The SELFUNION operator can only take a set of sets as input. This set of sets will be transformed into a simple set containing all elements of each inner set. The overall complexity of this operator is $O(N * n_{max})$, because every element of every input set has to be copied to the resulting simple set. The SELFINTERSECTION operator determines for every element of the first inner set if this element is contained in any other inner set. The outer set consequently has to be iterated (except the first inner set) which can be achieved in $O(N - 1)$ steps. The elements of the $N - 1$ inner sets then have to be compared to the elements of the first inner set. This can be achieved in $O(n_{max}^2)$ steps. Thus, the overall complexity of the SELFINTERSECTION operator is $O((N - 1) * n_{max}^2)$.

On the basis of the complexities of each function and operator, the overall worst case runtime complexity of a given pattern query can be calculated as the sum of the complexities of all functions and operators contained in the query. The underlying assumption of this worst case scenario is to consider the entire model (i.e., the set E) as input on each tree level. The analysis presented above demonstrates that the theoretical worst case complexity of an entire pattern query can be rather large, particularly if path functions are contained in the query. In this case the overall theoretical worst case complexity is facultative in the number of model elements.

6.2. Runtime performance

The overall runtime performance of a model query language basically consists of three components: the time to load the pattern query, the time to load the model(s), and the time to run the query on the model(s). In the following, we will analyse all three aspects for our GMQL implementation. To that end, we first define a set of pattern queries and a model base that we can use for our experiments. We then describe the technical setup of the experiments, that is, the computer we used and the method of collecting the runtime measurements. At last, we present and discuss the corresponding data.

6.2.1. Pattern queries

To measure the runtime performance of GMQL, we defined 10 pattern queries for EPC models and 10 pattern queries for ERMs. Tables A1 and A2 and in Appendix A contain more information on the exact queries we used. The EPC queries are all based on the workflow patterns defined by van der Aalst et al. [40] and the EPC syntax errors reported in [27]. The ERM queries are based on structures that were commonly observed in the model base we used. To analyse the runtime behaviour of GMQL, we wanted our performance analysis to include queries that we were sure would return large numbers of pattern occurrences. For this reason, we included rather generic pattern queries like the “ER paths” query for ERMs in our analysis. This query returns all paths starting in an entity type and ending in a relationship type. We also included queries that represent patterns observed in the literature in order to see how GMQL performs in real-life application scenarios. In addition, we want to cover a large range of the functions and operators provided by GMQL. Although not every function and operator was used in the test queries, their runtime performance can be expected to fall within the runtime range presented in the analysis at hand. Consider for example the function *ElementsWithPredRelationsOfType* that can be expected to not differ significantly from *ElementsWithSuccRelationsOfType* in terms of runtime performance. As pattern queries in GMQL are assigned to a specific modelling language, the language specifications given below were used. In line with the argument presented above, the set T_O is the set of all object types, whereas the set T_R is the set of all relationship types. Note that a relationship type is named with the initials of the object types it connects. The first initial represents the relationship type's source element type, whereas the second initial represents the relationship type's destination element type. The relationship type ‘FE’ of the EPC specification therefore leads from a function to an event. The language specifications

that were used to define the pattern queries are as follows:

$$\begin{aligned}
 T_{O(EPC)} &= \{\text{Function, Event, XOR, AND, OR}\} \\
 T_{R(EPC)} &= \{\text{AA, AE, AF, AO, AX, EA, EF, EO, EX, FA, FE, FO, FX, OA, OE, OF, OO, OX, XA, XE, XF, XO, XX}\} \\
 T_{O(ERM)} &= \{\text{EntityType, RelationshipType, RelationalEntityType, Attribute, Generalization}\} \\
 T_{R(ERM)} &= \{\text{EA, EG, ER, ERE, RA, RRE, REA, REG}\}
 \end{aligned}$$

6.2.2. Model base

As model base, we chose a repository of ERMs and EPCs that was available from a modelling project in the retail industry [41]. In total, the repository contains 53 EPC models and 46 ERMs. The largest EPC contains 264 elements and represents a material requirements planning process. The smallest EPC contains 15 elements and represents a deviation control process. The largest ERM contains 97 elements and represents an article data model. The smallest ERM contains 16 elements and represents an excerpt of a sales data model. The EPCs and ERMs of this repository are completely independent. They do not contain any cross-references (e.g., an EPC function that requires a particular data input that is modelled in a separate ERM).

We chose this model repository for two reasons. First, these process and data models were developed in a real life modelling project. Consequently, they allow us to test GMQL in a realistic context. Second, the repository explicitly includes very small models having a minimum of 15 elements as well as large models of up to 264 elements. This range of different model sizes allows us to evaluate the scaling characteristics of the pattern matching approach, as the size of the model will most likely influence runtime performance. Additionally, we argue that most conceptual models exhibit model sizes that fall into the range we cover here. We therefore conclude that the results we obtained for this repository are also comparable to other model collections.

6.2.3. Technical setup

To obtain the performance data for query execution in a statistically rigorous manner, we applied the steady-state performance methodology presented in [42]. The methodology was developed to control 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. As described above, GMQL was implemented as a plugin for a meta-modelling tool written in C#. This programming language suffers from the same measurements errors. Therefore, we adapted the approach of Georges et al. [42] to C#. To compensate for the non-deterministic behaviour of the virtual machine, we chose the so-called steady-state performance measurement determining programme performance after JIT compilation and optimization.

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 a complete garbage collection was forced. 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 methodology of Georges et al. [42] adapted to our setting.

During the initial warm-up phase, the JIT compiler optimized the pattern search algorithm. This implies running the algorithm a few times in order to minimize the variation coefficient [42]. This serves to reduce the probability of getting highly dispersed runtime measurements. The variation coefficient is calculated by dividing the standard deviation of all runs by its mean execution time. This led to searching each pattern in each model between five to thirty times. Once this warm-up phase was completed and a steady state was reached, the measurement phase started. During this phase, each query was executed on each model ten times. After ten search runs, the meta-modelling tool was restarted triggering another ten search runs. This process was repeated 35 times to compensate for the non-deterministic behaviour of the virtual machine [42]. For each process, the mean execution times were determined for each combination of query and model. This led to 35 mean execution times for each combination of query and model. The overall execution time of one query in one model was then computed as the mean of these 35 measurements.

To obtain performance data on model and query loading, we loaded each model and query from their respective databases 10 times. We recorded the time it took to perform these operations and determined the mean value of these measurements. As far as query loading is concerned time measurements do not only include the actual loading of the query string, but also reparsing it into the final object structure needed at runtime. The core meta-modelling tool we used as a basis for implementing GMQL can be used with a PostgreSQL or a SQLite database. We therefore conducted this measurement procedure for both database management systems.

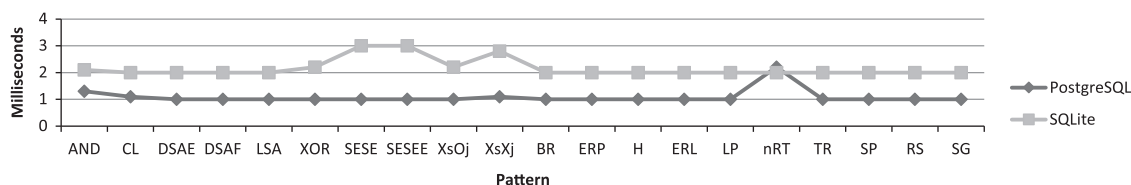


Fig. 12. Runtime measurements for query loading.

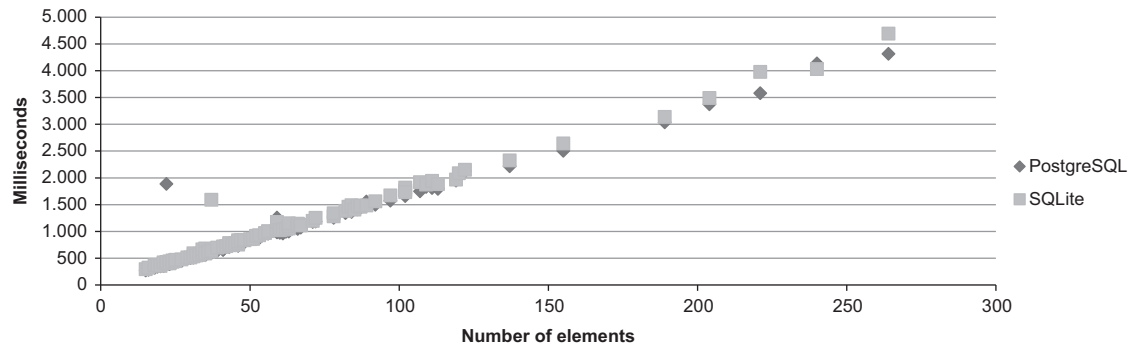


Fig. 13. Runtime measurements for model loading.

6.2.4. Runtime measurements for model and query loading

Fig. 12 contains measurements for loading pattern queries using both a PostgreSQL and a SQLite database. Each pattern is represented on the abscissa (see Tables A1 and A2 in Appendix A for the shortened query names). The ordinate represents time measurements in milliseconds. The figure demonstrates that the queries used in our experiments can be loaded in between one and 2.2 ms using PostgreSQL or in one and 2.8 ms using SQLite. PostgreSQL thus returns pattern queries a little faster than SQLite. In general, the parsing procedure that turns the query expression tree into a string value allows for small loading times, because the string value stored in the database can be retrieved with just one SQL statement and roundtrip to the database. If a complex object structure were mapped to corresponding tables using an object-relational mapper, multiple SQL statements and roundtrips might have been necessary. The chosen approach is thus similar to a document database that serializes data into the JSON format [43].

Fig. 13 contains measurements for loading models. Each model is represented on the abscissa by its number of elements. The ordinate represents time measurements in milliseconds. The figure indicates that load time increases almost linearly with the size of the model. Again, PostgreSQL is able to retrieve model data slightly faster than SQLite. Absolute load times range between 274.1 and 4316.9 ms for PostgreSQL as well as between 298.3 and 4696.2 ms for SQLite. As opposed to the GMQL plugin, the meta-modelling tool does not deserialize the object structure from a String. Instead, the database of the tool contains separate tables for each model and language construct depicted in Fig. 6 (i.e., element types, object types, relationship type, elements, objects, relationships, etc.). The object structure representing the model at runtime is thus mapped to corresponding table entries. In total, a set of eight tables needs to be queried in order to retrieve a model. To that end, multiple SQL statements and database roundtrips have to be performed. The more elements a model contains the more joins have to be performed in subsequent roundtrips to load each of them. This is why load times increase with the size of the model. The database and storage layer of the meta-modelling tool were built in this way to reduce redundant data management and to provide easy debug features. This design choice was taken during the previous research project [36]. Table A3–A5 in Appendix A contain the exact measurements for pattern queries and models. It should be noted that the measurements presented here are only representative of the particular database management systems and the implementation of our meta-modelling tool. If one were to implement GMQL for a different modelling tool that supports different database management systems, these load times might vary significantly (cf. Section 7.2 for a detailed discussion on the limitations of this analysis).

6.2.5. Runtime measurements for query execution

Fig. 14 contains runtime measurements for the ERM queries. Each model is represented on the abscissa by its number of elements. The ordinate represents time measurements in milliseconds on a logarithmic scale. Each measurement depicts one overall execution time of a given query as described above. The figure demonstrates that GMQL is able to answer queries with acceptable runtime performance. In all but eleven cases, results were returned within less than 100 ms. In 326 out of the 460 (46 models \times 10 patterns) cases, the respective query was even answered in less than one millisecond. The longest runtime that was recorded amounts to 565.44 ms. Large models tend to lead to longer runtimes. Other than model size, the figure demonstrates that runtimes increase in case the pattern query contains path functions. In particular, the longest and shortest paths queries as well as the ERP query calculating paths from all entity types to all relationship types take the longest in all 46 models. This was to be expected, because path functions determine all paths from all start to all target elements. As demonstrated in Section 6.1, the maximum number of possible results is therefore facultative in the size of the model. This fact is mirrored in the runtime measurements. Other than model size, the complexity of the pattern query as defined by whether it contains path functions or not also influences runtime performance. However, given the measurements we obtained in our experiments, the runtime performance of GMQL can be considered acceptable.

This claim is also supported by Fig. 15 containing runtime measurements for EPCs. The figure is arranged analogously to Fig. 14. In only 51 out of 530 (53 models \times 10 patterns) cases, runtimes longer than one millisecond were recorded. The longest overall runtime that was measured in our experiments amounts to 71.78 ms. Even if complex pattern queries containing path functions are run on large models with hundreds of elements, runtime performance still remains acceptable

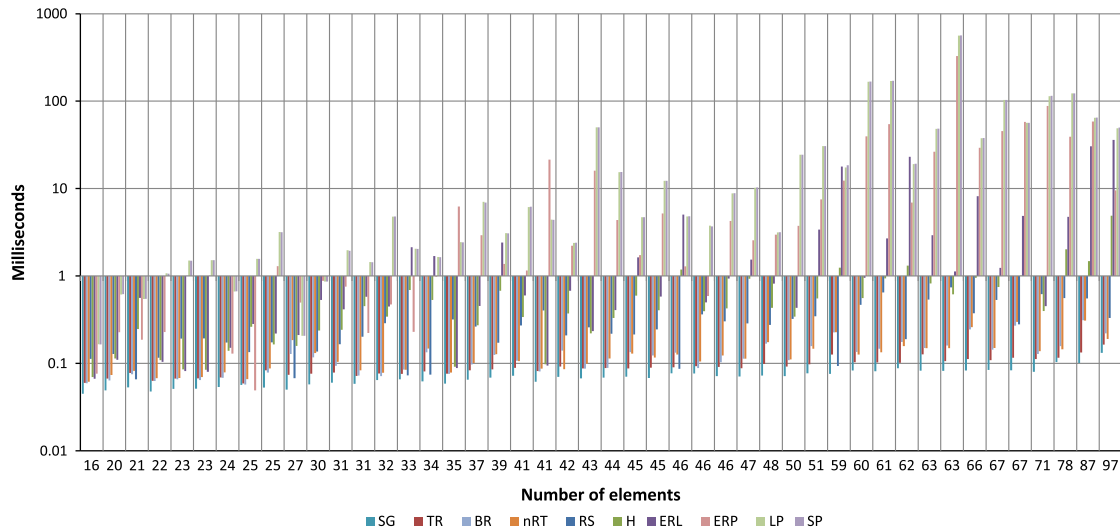


Fig. 14. Overall runtime measurements for ERMs.

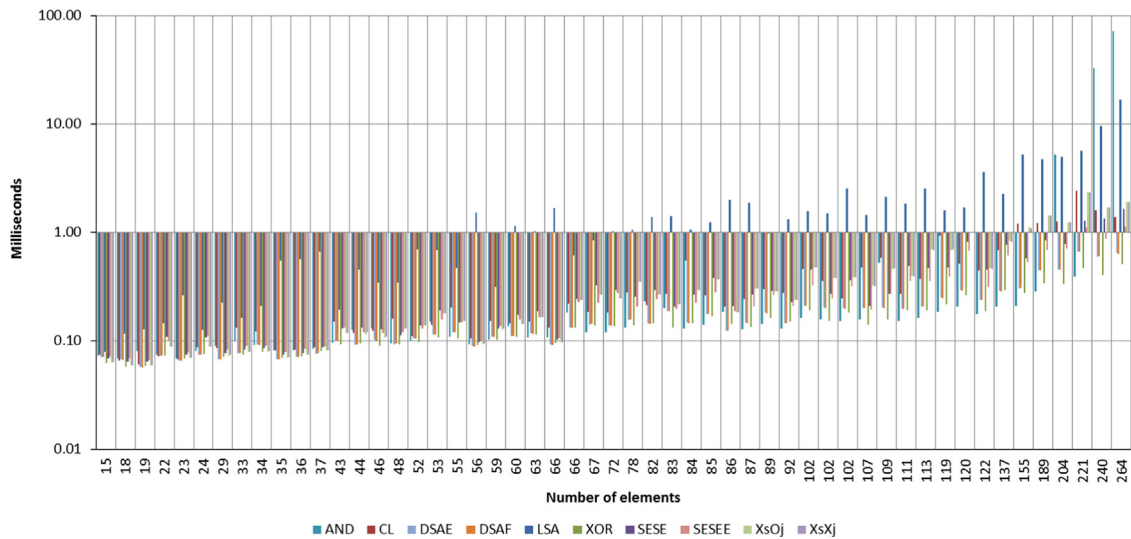


Fig. 15. Overall runtime measurements for EPCs.

for EPC models. The influence of the model size on runtime performance that can be found in ER models is not as significant in EPC models. Although runtimes do increase with the size of the model, the effect is however not as substantial, because pattern occurrences can generally be computed faster in EPCs as in ER models. This can be explained by the fact that EPC diagrams are directed graphs. While in ER models a given element's source and destination elements both need to be considered to determine its neighbours, in EPC diagrams only the destination elements need to be included in the matching process. This speeds up execution times. Note that the measurements presented in Figs. 14 and 15 contain both the time required to build up the basic input sets E , O and R as well as the actual matching procedure.

The confidence interval of the 990 measurements of Figs. 14 and 15 indicates the reliability of these measurements given a confidence level of 99 per cent. In 969 cases the confidence interval is smaller than three per cent of their average runtime. This means that if the performance analysis was repeated, we could say with 99 per cent certainty that these measurements would scatter in a three per cent interval about the data presented here. Our measurements consequently exhibit only very little non-deterministic behaviour and can be considered reliable. Two ERM measurements for the longest path query, however, exhibit a confidence interval of 10.03 and 10.24 per cent of their average runtime. These high values can occur due to garbage collection, just-in-time (JIT) compilation, adaptive optimization, or process switching and are more likely to occur

on long running tasks. As the longest path pattern has a rather high average runtime compared to other ERM patterns, it appears that an adaptive optimization was applied to a few measurements resulting in high variations for these measurements.

While Figs. 14 and 15 provide an overview of GMQL's runtime performance, Tables A6 and A7 in Appendix A contain the exact measurements (RT columns) and information about the number of patterns found in each model (#P columns). Table A6 contains respective information for ER models; Table A7 does so for EPCs. Each model is again represented by its number of elements. The tables suggest that, other than the size of the model and the specific GMQL functions and operators used in a given query, the number of pattern occurrences also influences runtime performance. Consider runtime measurements for the ERP query calculating all paths from all entity types to all relationship types in Table A6. Naturally, every ER model will contain many such paths. Runtime measurements for this query very strongly relate to the number of pattern occurrences. In case this number is low the matching process terminates within fractions of a millisecond (cf. the last row of Table A6). If, however, the model contains a large number of occurrences, runtimes increase accordingly. The largest runtime that was recorded for this pattern query amounts to 328.49 ms. This measurement corresponds to 16,875 paths starting in an entity type and ending in a relationship type. This measurement was obtained in a model consisting of 63 elements. This model exhibits unusual peaks of runtime for other pattern queries as well. The shortest path query took 565.44 ms to terminate, while the longest path query took 562.58 ms. Again, these runtime peaks can be attributed to the fact that these queries contain path functions. In particular, the shortest and longest path functions determine all paths but only return the shortest or longest ones. Therefore, a great number of intermediate pattern occurrences are determined that are then discarded. These long runtimes are consequently caused by the fact that this model contains thousands of paths that start and end in an entity type. The argument that the number of occurrences influences runtime performance is also supported by the TR query returning relationship types that are directly related to exactly three entity types. As hardly any such structures exist in the model collection we used, the matching process terminates in fractions of a millisecond.

Table A7 also confirms the influence of the number of returned pattern occurrences on runtime performance. Consider the AND pattern reported in [27]. The largest runtime for this pattern amounts to 71.78 ms. This measurement corresponds to 447 pattern matches found in a model with 264 elements. In all other models, hardly any such structures occur. The matching process thus terminates within fractions of a millisecond. This finding can be explained by the tree structure of the pattern queries. If at one point in the matching process an empty set is returned to the next higher tree level, the overall matching process concludes quickly, as all subsequent function and operator calls also return empty sets if one input parameter contains no elements.

In summary, we conclude that runtime performance is influenced by a combination of three factors, namely the size of the model, the number of found pattern occurrences, and the usage of path functions within the pattern query. Large models tend to result in long runtimes. However, the longest runtimes were not necessarily observed in the largest model. This finding is supported by the ERM consisting of 63 elements, for which pattern queries took between 328.49 and 565.44 ms. Other than the relatively large model size, these runtimes can be attributed to the fact, that (a) the corresponding queries contained path functions and (b) large numbers of (intermediate) pattern occurrences were determined. Slower runtimes observed in larger ERMs can in turn be explained with less pattern occurrences. Consequently, runtimes are not influenced by any one factor, but by the combination of all three.

Table 3

Coefficients of determination for multiple regression analyses (MS=model size; #P=number of patterns).

Model type	Pattern query	Adj. R^2 (MS)	Adj. R^2 (#P)	Adj. R^2 (MS & #P)	Adj. R^2 (MS ² & #P)
EPC	AND	0.333	0.83	0.827	0.828
	CL	0.813	0.557	0.908	0.956
	DSAE	0.936	0.191	0.937	0.967
	DSAF	0.936	0.747	0.935	0.956
	LSA	0.753	−0.019	0.751	0.877
	XOR	0.949	−0.019	0.948	0.925
	SESE	0.887	0.787	0.923	0.97
	SESEE	0.896	0.592	0.919	0.949
	XsOj	0.874	0.857	0.917	0.96
	XsXj	0.874	0.856	0.916	0.96
	BR	0.66	0.776	0.967	0.97
	ERP	0.147	0.972	0.982	0.983
	ERL	0.426	0.743	0.8	0.849
	H	0.54	0.839	0.88	0.91
ERM	LP	0.175	−0.012	0.156	0.128
	nRT	0.522	−0.007	0.511	0.487
	RS	0.483	0.138	0.607	0.572
	SP	0.175	0.045	0.16	0.133
	SG	0.919	0.599	0.967	0.958
	TR	0.898	0.153	0.973	0.932

To gain a deeper understanding of how much each individual factor and the concert of all factors influence runtime performance, we conducted a multiple regression analysis with various combinations of these inputs. We first separately considered the model size and the number of pattern occurrences. We then considered the combination of these two factors assuming both a linear as well as a non-linear correlation between model size and number of pattern occurrences on the one hand and runtime performance on the other hand. We performed such an analysis for each pattern query, thereby considering all three factors influencing runtime performance. Table 3 contains the coefficients of determination as adjusted R^2 values for each pattern query (cf. rows of the table) and regression model (cf. columns of the table). These values explain the percentage of runtime variability that can be explained by the respective input factor(s). The coefficients are thus a measure of the quality of the underlying regression model.

Considering the cases of only the model size or the number of pattern occurrences as factors explaining runtime performance (cf. columns 3 and 4 of Table 3), we can see that for EPCs the model size can explain significantly more runtime variability than the number of pattern occurrences. This holds true for nine of the ten EPC queries used in our experiments. The respective R^2 values range between 75.3 and 94.9 per cent. For ER models, however, the model size explains more runtime variability than the number of pattern occurrences in six out of ten cases. The range of R^2 values is significantly larger containing values between 17.5 and 91.9 per cent. This leads to conclude that while both factors do explain a (significant) percentage of runtime variability, the model size has a greater influence than the number of pattern occurrences. This is also supported by the R^2 values for those pattern queries containing calls to either the *LongestPaths* or *ShortestPaths* functions (i.e., the LSA query for EPCs as well as the LP and SP queries for ERMs). These functions are built in such a way as to calculate all paths from all start to all target elements, but only return the shortest or longest ones. In case these functions are contained within the pattern queries, the corresponding R^2 values point to no significant correlation between runtime and number of pattern occurrences. This is to be expected, as the matching mechanism will determine a large number of paths (which is computationally expensive), but will only return a comparatively small number of results (i.e., the shortest or longest paths).

The finding that both the aforementioned factors influence runtime performance in various degrees led to conducting two additional multiple regression analyses. First, we assumed that there is a linear correlation between the combination of model size and the number of pattern occurrences on the one hand and runtime performance on the other. Second, having determined that the model size has a greater influence than the number of pattern occurrences in most cases, we assumed that the model size will go into this correlation quadratically (cf. columns 5 and 6 of Table 3). For EPCs, the latter analysis can explain the most runtime variability in eight out of ten cases. Respective R^2 values range between 82.8 and 97 per cent. This range is consistently higher than in all other regression analyses done before. For ERMs, the latter analysis can explain the most runtime variability in four out of ten cases. The former regression model can do so in three out of ten cases. As explained above, if longest or shortest paths need to be determined, the number of final pattern occurrences plays no significant role in explaining any runtime variability. Still, even in those cases where the former model cannot explain the most runtime variability, the coefficients of determination are consistently high. The regression model represented in the last column of Table 3 can therefore best describe the overall runtime performance in terms of model size and number of pattern occurrences.

In summary, this performance evaluation demonstrates that long runtimes occur only in large models and if large numbers of pattern occurrences are found. However, the overall runtime performance of GMQL can be considered acceptable, because most pattern queries could be answered in fractions of a millisecond. The largest runtime that was measured amounts to 565.44 ms for the shortest path query in ERMs. As this query calculates tens of thousands of paths, this measurement can be considered acceptable. The multiple regression analysis presented above indicates that the combination of squared model size and number of pattern occurrences best explains runtime performance, as it is able to explain up to 98.3 per cent of runtime variability given the pattern queries and model collection we used for our experiments.

7. Limitations, related work, summary and outlook

7.1. Limitations of GMQL

In this paper, we presented the generic model query language GMQL. It is designed to allow for finding fragments with specific structural and semantic characteristics in conceptual models created in arbitrary modelling languages. In terms of the requirements for a generic model query language defined in Section 2.6, GMQL treats any conceptual model as an attributed graph. This means that each node and each edge of the model graph has a set of attributes that define its semantics (e.g., type, label, etc.). GMQL is therefore able to express pattern queries for all graph-based modelling languages (R1). A graph essentially consists of two sets, namely the set of its nodes and the set of its edges. GMQL offers set-altering functions and operators that take these basic sets as input and perform certain operations on them. The functions fall into four different classes designed to support requirements R1.1–R1.5. The functions contained in the first class return single model elements that have particular characteristics like a predefined type or label (R1.1). The functions belonging to the second class return single elements that have (a particular number of (ingoing or outgoing)) edges (R1.2). All functions of the first two classes can take variables as input that allow for comparing the values of particular attributes to one another (R1.3). The functions of the third class return adjacent elements and the

edges between them (R1.4). The functions of the fourth class are able to determine element paths of arbitrary length (that do or do not contain particular elements) (R1.5). The GMQL operators are furthermore designed to combine substructures to overall pattern occurrences (R2). Other than constructs to specify a pattern query GMQL includes a pattern matching algorithm (R3).

GMQL differs from general database query languages like SQL. In SQL, the result of a query is a new table that is constructed from the rows and columns of existing tables in the database. The columns of this new table are created by selecting a subset of existing columns, whereas the rows of the new table are created by joining existing rows. So far, GMQL is able to identify distinct pattern structures in a set of distinct model graphs. It is, however, not (yet) able to return a new model by joining previously identified pattern structures. Whether this is required in the context of conceptual model analysis and how this could be implemented in GMQL remains a subject of future research.

The implementation of GMQL we described in [Section 5](#) is only an exemplary implementation for an existing meta-modelling prototype. The fact that the tool we used is a meta-modelling tool that allows for specifying arbitrary graph-based modelling languages eases the implementation effort, because language specifications can be directly accessed by the GMQL plugin. In doing so, element type information contained in these language specifications can be included in the query definition. If one were to implement GMQL for a different modelling tool that does not provide meta-modelling features, one would have to find a way to access the specifications of the modelling languages provided by that particular tool. Non-meta-modelling tools also need to store these language specifications. As opposed to meta-modelling tools that allow for dynamically creating new languages, these specifications are hard-coded in the software. The particular GMQL implementation requires access to these language data. Once the matching mechanism is triggered, the model data has to be converted to the basic input sets E , O , and R . The GMQL implementation therefore also needs access to the model data stored within the database of the given modelling tool.

As explained above, GMQL is able to find distinct pattern occurrences in distinct models. Conceptually, GMQL can already be used to search for pattern occurrences that stretch across multiple models developed in the same modelling language. An example of such a pattern occurrence is a path that starts in one model and ends in another. Conceptually, this can be easily realized by including all elements of the models that are supposed to contain a pattern occurrence in the input set E . The current implementation, however, does not yet support this. Extending our implementation of GMQL to allow for finding pattern occurrences that stretch across multiple models is currently in progress.

Furthermore, as explained in the implementation section, a GMQL pattern query contains element type information of a modelling language. As such, a GMQL query for EPC models would return an empty set if it is run on an ER model. In some model analysis scenarios it may, however, be necessary to define a pattern query that is run on a collection of mixed models. An example of such a query is the separation of duties compliance rule introduced in [Section 2.2](#). In GMQL, such mixed model queries are possible as demonstrated by the corresponding query in [Section 4.2](#). However, our current GMQL implementation does not yet support this. It can be easily extended to support this feature, because the set E simply has to be extended to include all elements of all input models. In addition, the set T of all element types has to be extended to include all types of all involved modelling languages.

GMQL is designed for pattern matching in conceptual model graphs. As demonstrated in the sections on motivating examples and corresponding pattern queries ([Sections 2 and 4](#)), GMQL supports a wide variety of model analysis scenarios that involve pattern matching. Naturally, there are a great number of model analysis scenarios that do not involve pattern matching. Examples of such scenarios are analysing process execution semantics, standardizing element labels, or structurally analysing process models without an a priori known pattern query. These model analysis scenarios are outside the scope of the paper at hand, because they are not concerned with pattern matching. In the following, we will, however, briefly describe some of them in order to better position GMQL in the plethora of model analysis approaches put forth in the literature and explain how they are related to GMQL.

In terms of process execution semantics many process modelling languages allow for creating executable process models (e.g. BPEL). The execution order and semantics of these models can be analysed, for instance, by using labelled transition systems (LTS) [44] describing a set of states a given process instance can be in. Other mechanisms for analysing execution semantics include finite transition systems (FTS) [45] or causal footprints designed to measure the similarity of two given process models [32]. We refer to respective literature on analysing process behaviour for a more detailed discussion of the proposed approaches. GMQL as presented in the paper at hand is designed for pattern matching in model graphs, while the approaches cited above are not. However, some of the issues addressed by these approaches can also be solved using query languages (see related work section below). The GMQL functions and operators can, for instance, be used to solve some of the issues addressed by LTS or FTS. Consider, for example, [24] who augment a process query language to include linear temporal logic in order to detect predecessor/successor patterns in the domain of design time compliance checking. These patterns represent a compliance rule stating that if a particular activity A is performed a certain other activity B has to be performed after. The exemplary BPMN models in the left part of [Fig. 16](#) represent respective patterns. In case (a) the path from A to C represents a violation of the rule, while in case (b) this path does not represent a violation (because of the different execution semantics of the AND and XOR gateways). In [24] these models are fed to a model checker that analyses their execution semantics. In GMQL, these structures can immediately be found (cf. right part of the figure). The respective query identifies directed paths from A to C that do not contain B. In a second step, the result set is cleared by all paths that contain splitting AND nodes. This way, the path in (a) is detected as a compliance violation, while the path in (b) is not. Hence, the specialised path functions GMQL offers are sufficient in this

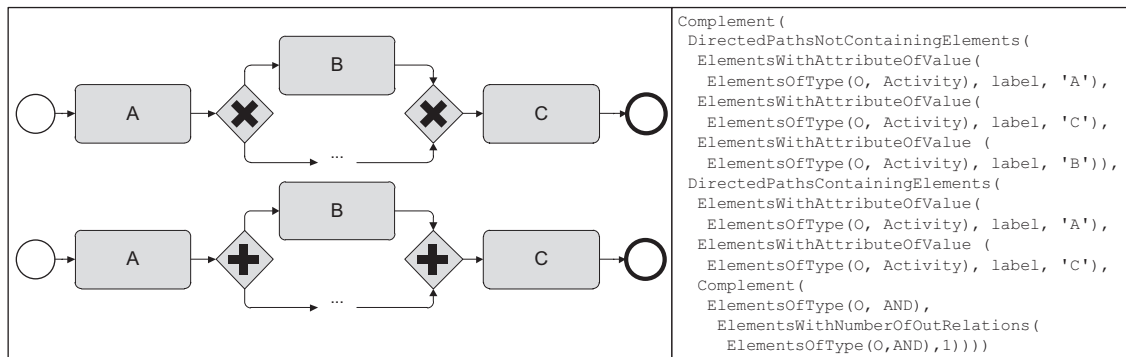


Fig. 16. Predecessor/successor compliance rule (left) and according GMQL query (right).

case to find these structures. They allow for analysing the semantic relationships between sets of model elements. Such relationships can also be expressed using temporal logic statements. Such statements are then analysed by state machines which may suffer from the well-known state space explosion problem [24]. Other approaches use the notion of traces to address these issues [46,47]. A trace represents one possible execution sequence of activities, that is, a path in the model graph. The GMQL functions and operators provide a means of expressing and subsequently analysing such semantic relationships without a state machine.

In terms of standardizing element labels, GMQL is able to include such labels in the matching process. The function *ElementsWithAttributeOfValue* takes a set of elements, an attribute, meaning, for instance, the label of a model element, and a concrete attribute value as input. This allows for specifying an EPC pattern query as follows:

```
ElementsWithAttributeOfValue(ElementsOfType(0, Function), Label, 'Check Invoice')
```

Such a query would return all EPC functions that have the label “Check Invoice”. GMQL is therefore to some extent also suited for a semantic analysis of conceptual models. However, such an analysis relies significantly on a terminological standardization of element labels. A label “Check Invoice” may or may not be semantically identical to a label “Bill Verification”. Simply searching for model elements with a given label can therefore lead to ambiguous, incorrect, or false results. Consequently, prior to executing pattern queries, all models first have to be terminologically standardized. As respective approaches are outside the focus of the paper at hand, we refer to the literature for a discussion of appropriate techniques. For instance, Thomas and Fellmann [48] propose an ontology-based standardization procedure. Delfmann et al. [49] do so using domain thesauri and predefined phrase structure rules providing a template for labels of a given element type. GMQL can be easily augmented to integrate these approaches by including their metrics for semantic or linguistic similarity in its matching process instead of a concrete string value. A semantic analysis of models, however, can only be partially automated. In addition to standardized terms and phrase structures, an analyst must also have an understanding of the semantic domain of the conceptual models he wishes to analyse.

Some model analysis approaches perform a structural analysis without an a prior known pattern query. Popular approaches that fall into this category are the clone detection mechanisms proposed by Uba et al. [50] and Ekanayake et al. [51]. These algorithms determine frequently occurring subgraphs within a collection of process models without taking a predefined pattern query as input. They are thus closely related to the graph-theoretical problem of frequent pattern detection [52]. In contrast, GMQL is closely related to the graph-theoretical problems of subgraph isomorphism and homeomorphism [20] and its corresponding application scenarios.

7.2. Limitations of the performance analysis

The overall performance of a model query language consists of three components, namely the time to load the model(s), the time to load the pattern query, and the time to execute the query. As we have seen, the first two components depend to a large extent on the particular database management system that is used and the implementation of the storage functionality that reads from and writes to this database. Therefore, the measurements that we present here are not representative of other modelling tools. If one were to implement GMQL for a different modelling tool, load times for patterns and models will differ. Other than the particular database management system used (PostgreSQL, SQLite, etc.) the database type (relational database, document database, graph database, etc.) and the particular database schema that stores model data will greatly influence load times as well. Naturally, the particular computer and its hardware configuration (CPU, HDD, SSD, RAM memory, etc.) also play a role. Lastly, if an object-relational mapping framework is used, its performance influences the overall load time performance as well. The particular GMQL implementation we present in this paper parses the actual object structure that represents the query expression at runtime into a simple string value that can be written and read relatively fast. In contrast, the meta-modelling tool we used as a basis to implement GMQL maps all model objects to corresponding table entries. This leads to an increase of load times.

The multiple regression analysis presented above cannot explain all runtime variability. To do this, more model- and query-specific factors need to be taken into account. The matching algorithm is built in such a way as to calculate the input parameters of a given function or operator in exactly the same way they are fed into it. Depending on the model, this can influence runtime performance. Consider the “AND might not get control from XOR”-query for EPC models described in [Section 4.4](#). If a model does not contain any AND connectors, the calculation of the second input parameter results in an empty set, which subsequently leads to the overall matching result being an empty set. However, the first input parameter is still being calculated. In contrast, if a model does not contain any XOR connectors, the matching process aborts much earlier resulting in a smaller runtime.

The overall model structure also influences runtime performance. For instance, if a process model of a given size contains a large number of split connectors, the number of possible paths that may need to be calculated increases accordingly. A process model of the same size that contains a linear sequence of activities can in contrast be searched in comparatively little time. This argument is supported by the execution times reported in ER models that are significantly longer than in EPCs. As these models are undirected graphs, the number of paths that need to be considered in a model of a given size is much higher than the number of paths in an EPC of the same size. This results in longer search times.

The performance analysis is furthermore limited to models that contain at most 264 elements. We argue that most models exhibit sizes that fall into the range we cover here. However, performance problems may occur if a GMQL query is run on a model containing thousand or even tens of thousands of elements. As part of our future research agenda we will therefore conduct runtime experiments on extremely large models. We will furthermore augment GMQL to include information on graph-theoretical model characteristics (e.g., tree-width, planarity, etc.) in its matching process to speed up execution times.

7.3. Comparison of GMQL to existing model query languages

In order to evaluate existing model query languages and compare them to GMQL, we draw on the requirements for a generic model query language defined in [Section 2.6](#). To ascertain a language's features on a more fine-grained level, we refine these requirements to a set of thirteen evaluation criteria. We determine from the literature if a given query language is able to query models of any type (C1) (i.e., process models, data models, organizational charts, etc.) or modelling language (i.e., EPC, BPMN, etc.) (C2). A query language may, for instance, be designed to query arbitrary process models. It is thus designed to query a particular type of model (i.e., process models). Within one model type, however, all languages may be supported. As far as C1 is concerned we use the name of the query language as an indicator of its applicability to different model types (i.e., a process query language is by definition only applicable to process models). General graph query languages are considered independent of a particular model type. As far as C2 is concerned we consider a query approach to be independent of any modelling language if it can conceptually express queries for any language of a given type, if its implementation is language-independent, and if the paper provides exemplary model queries for multiple languages. Its implementation is language-independent, if the query language has been implemented in a meta-modelling tool similar to the one proposed in the paper at hand or if the implementation has access to all language specifications contained in the utilized modelling tool. C2 is only partly fulfilled if the query approach is only conceptually able to do so, its implementation and/or examples, however, are language-specific. C1 and C2 are derived from requirement R1.

We determine if a query language is able to consider type (C3) and label (C4) information in its matching procedure. C3 and C4 are derived from requirement R1.1. Furthermore, we assess if a query language is able to express and execute queries consisting of isomorphic model fragments (C5–C7 as well as C9). To that end, the query language has to be able to express and execute queries consisting of elements having relations (C5), elements having ingoing or outgoing relations (C6), and elements having a particular number of relations (C7). C5–C7 are derived from R1.2. The query language has to be able to express and execute queries that compare attribute values to one another (C8 relating to R1.3). Lastly, the query language has to be able to express and execute queries consisting of adjacent model elements and the relations between them (C9 relating to R1.4). Criteria C10 and C11 refer to partly homeomorphic model fragments. A query language thus has to be able to express and execute queries consisting of arbitrarily long element paths and loops (C10). Furthermore, specific paths that do or do not contain particular elements as well as shortest or longest paths need to be found (C11). C10 and C11 are derived from R1.5. Lastly, a query language has to be able to express and execute queries that combine pattern substructures (C12 relating to R2). In addition to a feature-based comparison, we also determine if a runtime analysis has been conducted for the given query language (C13). Such an analysis has to include runtime measurements in units of time in order to be comparable to the data presented in the paper at hand. In some articles, the performance of the given query language was ascertained as a percentage increase in comparison to some older version of that language (e.g., [53]). We do not consider such analyses, because there is no direct way of comparing corresponding runtime measurements. In addition, a runtime analysis needs to be performed on conceptual models to be considered relevant. If such a runtime analysis has indeed been conducted, we argumentatively compare the measurements to the data presented in this paper. [Table 4](#) summarizes our findings. Each row represents one query language; each column contains one criterion as described above. Each cell contains “+” if the given language supports the corresponding feature and “–” otherwise. “0” means the query language partly fulfils a given feature (see above and explanation in the corresponding paragraph below).

The first language we consider is *aFSA* [53] that allows for querying process models using annotated finite state automata. The language is designed to find structures in process models (i.e., C1=N) that are specified as corresponding automata. Conceptually, the language is able to query process models of any process modelling language if these models are transformed to annotated finite state automata (i.e., C2=P). The tool's implementation and the exemplary queries defined in

the paper are language-specific. Furthermore, it is able to find pattern matches that are isomorphic to a predefined query model. Path search is thus not supported, neither is a runtime evaluation in units of time.

APQL is a process query language designed to analyse the semantic relationships between tasks [54]. An *APQL* query is similar to a temporal logic statement. It consists of a set of so-called assignments that place particular restrictions on the control-flow of a process. These restrictions translate to particular fragments in the model graph. In *GMQL* such restrictions can be specified using the {Directed}Paths{Not}ContainingElements functions. Shortest and longest paths cannot be defined using *APQL*. Neither can attribute values be compared to one another. As the name indicates, the language is designed for analysing process models. Conceptually, it is possible to define queries for any process modelling language. The implementation and examples, however, are restricted to Petri Nets. The runtime analysis suggests that *APQL* is also able to execute queries within a matter of milliseconds. The presented runtime data, however, cannot be directly compared to the ones presented here, because different queries and models were used. Furthermore, runtimes are measured for an entire collection and not for each model. Note that *APQL* returns only the model that contains a pattern match. It is not possible to see how many matches occur and where.

The authors of [55] propose a query language whose matching algorithm is based on the well-known Ullman algorithm for subgraph isomorphism [56]. Subgraph isomorphism is concerned with the identification of exact pattern matches (cf. Section 1). All nodes and edges of a given pattern graph are matched to a subset of nodes and edges in the model graph. As subgraph isomorphism is known to be NP-complete [57], Jin et al. [55] proposes a filtering mechanism that reduces the number of models that need to be checked in order to speed up the matching process. The authors propose a filter that provides information as to what element is contained in what models. If a pattern query, for instance, contains the element “Check Invoice”, then only those models need to be searched that also contain that element. The candidate models are then fed to an adoption of the widely known Ullmann algorithm [56] for subgraph isomorphism. The performance evaluation indicates that the filter significantly speeds up computation time. Pattern occurrences can also be found in a matter of milliseconds. The approach is designed to query process models. As the Ullmann algorithm is a general graph pattern matching algorithm, the query language could be extended to query models of other languages as well. The implementation, however, is specific to Petri Nets. In addition, the Ullmann algorithm is concerned with finding exact matches. Path search is thus not supported, neither is comparison of attribute values or combining substructures. The query language is implemented in the BeehiveZ framework.

BPMN-Q [14] is a query language for BPMN. The language allows for modelling a pattern graph much like a process model is developed. The language allows for finding isomorphic structures as well as paths. *BPMN-Q* does not allow for finding paths with particular restrictions or comparing attribute values to one another. As the name indicates, *BPMN-Q* can only analyse BPMN graphs. Although *BPMN-Q* is also based on the concept that a process model is essentially a graph, its implementation is restricted to BPMN. To query a collection that includes models of arbitrary modelling languages, its implementation would have to be augmented. This is arguably simple for other process modelling languages. If the same holds true for data modelling languages or any other language type also remains to be seen. The performance evaluation of *BPMN-Q* presented in [14] suggests that it may require *BPMN-Q* up to half a second to execute a query on a model consisting of about 30 elements. Although not directly comparable (different pattern queries and different models were used), the runtime data presented in the paper at hand suggests that *GMQL* is able to execute queries faster than *BPMN-Q*. The potentially large runtimes of *BPMN-Q* are caused by the fact that a *BPMN-Q* query is translated to an SQL statement that is run on a relational database containing the model data. As explained in [58,59], SQL may run into performance problems when calculating element paths.

BPMN VQL is another visual query language for BPMN [60]. It is able to find arbitrary isomorphic structures within BPMN models as well as paths. More complex path functions designed to analyse semantic relationships between activities (paths (not) containing elements, etc.) are not supported. Sub-structures can be combined using logical operators. Specific attribute values cannot be compared to another. A performance analysis has not been conducted.

BPQL is a model query language for process models [13]. The authors define a process modelling language and construct a query language that is able to extract information from models developed in this language. *BPQL* is thus specific to a particular model type and modelling language. Similarly to *GMQL*, a *BPQL* query consists of nested calls to a predefined set of functions that perform particular operations on the model. Basically, these functions allow for determining adjacent model elements and start activities of a process. By nesting these functions paths of a predefined length can be determined. The length of these paths is determined by the number of nested function calls. It is not possible to search for paths of arbitrary length and paths having particular restrictions. *BPQL* offers a set of operators that allow for restricting the attribute values of particular model elements. No runtime analysis has so far been conducted for *BPQL*.

BP-QL is another query language for process models [12]. Similarly to *BPQL*, the authors define a meta-model for a process modelling language and construct a corresponding query language that can detect particular structures in the models created with this language. *BP-QL* is able to detect model fragments that are isomorphic to a predefined pattern query. The language can also express path search operations. Particular path search operations allowing for finding paths that do or do not contain particular elements are not included in the query language specification. No runtime evaluation has been conducted for *BP-QL*.

Cypher is a graph query language for the graph database Neo4j [61]. The declarative query language uses concepts of SQL and SPARQL (see below) to define pattern queries. A graph database consists of nodes and connecting relationships that each store data in properties. To the best of our knowledge, Neo4j and *Cypher* have not been used as a repository and

querying approach for conceptual models. However, any conceptual model regardless of its type or language can be represented as a graph and therefore Cypher could be used for different model types. This, however, has not been tried out yet ($C2=P$). Cypher is able to detect graph fragments that are isomorphic to a predefined pattern query. Furthermore, it is possible to search for paths and loops (not) containing a particular element as well as for longest or shortest paths or loops. No runtime analysis has so far been conducted to determine the performance of Cypher on conceptual models. It therefore remains to be seen if Cypher indeed possesses a runtime performance that can match the data presented in the paper at hand.

EMF-IncQuery is a query language for conceptual software models created using the Eclipse Modelling Framework [62]. The query language offers a set of constructs that allow for creating textual pattern queries. These constructs are inherently bound to the meta-model of the EMF modelling language that can be used to develop models that are similar to UML class diagrams. EMF-IncQuery is thus a query language that is specific to a particular model type (class diagrams) and modelling language (EMF modelling language). It allows for finding path structures in models. Additionally, it is possible to express queries consisting of paths that must or may not contain particular elements. Shortest and longest paths cannot be found. EMF-IncQuery is able to consider attribute information in its matching process. Attribute values can be compared to one another. EMF-IncQuery is, however, not able to express queries that consist of elements having a particular number of (incoming or outgoing) relations. In [63], the authors conduct an analysis of the query language's runtime performance concluding that it returns results within fractions of a second even for extremely large models.

IPM-PQL is a process query language designed to detect patterns in process models that are stored in an XML database [64]. An IPM-PQL query is represented as XML code. Conceptually, process models created in arbitrary modelling languages can be queried with this language provided they are transformed to the specific XML format required by IPM-PQL. The implementation exemplarily demonstrates this for a simple process modelling language defined in the paper. The query language is able to consider type and label information in its matching process. It is able to express queries consisting of single elements and their relationships, adjacent elements and paths. More complex path restrictions or a comparison of attribute values cannot be expressed. In addition, the language cannot express queries consisting of elements having a particular number of (incoming or outgoing) relationships or a combination of sub-queries. No performance analysis has been conducted.

The Object Constraint Language (*OCL*) allows specifying expressions on UML models [65]. Among other purposes OCL can be used as a query language for UML models and other model types that can be defined using the Meta-Object Facility (MOF). Conceptually, any modelling language defined in MOF can be queried with OCL. The query language is able to consider type and label information as well as relations. As relations in MOF are not directed, incoming or outgoing relations cannot be expressed. Moreover, it is able to express queries consisting of adjacent elements and comparisons of their attribute values. Paths or loops and restrictions on them cannot be expressed. However, it provides operators to combine sub-queries. No performance analysis has been conducted.

The Process Pattern Specification Language (*PPSL*) is a query language for UML activity models [66]. Models are transformed into labelled transition systems and patterns into temporal logic, allowing a linear temporal logic (LTL) model checker to execute the query. Note that the LTL model checker returns only the model that contains a pattern match. It is not possible to see how many matches occur, but one counter example violating the pattern is provided. As temporal logic is used, it cannot be applied to other model types than process models. Conceptually, the approach can be applied to other process modelling languages. The examples and the described implementation are restricted to UML activity models. PPSL is able to consider type and label information in queries. Furthermore, it is able to express queries to find isomorphic model fragments, but not fragments with a specific number of in- or outgoing relations. Also, paths between UML activities can be found that must or may not contain particular elements. The specified patterns cannot be combined to sub-queries and no performance analysis has been conducted so far.

SPARQL is a query language for graph data that is stored in RDF format [67]. Similarly to Cypher, it provides features to express queries consisting of isomorphic fragments as well as paths and loops that must or may not contain particular elements. Type and label information as well as a comparison of attribute values can be considered in the matching process. The feature set of SPARQL is thus comparable to that of GML. To use SPARQL as a model query language, conceptual models need to be stored in a RDF database. The work of Beheshti et al. [68] provides a SPARQL-based query language that is specific to process models developed in a modelling language that is introduced in that paper. The performance analysis suggests that this SPARQL implementation can query repositories of models within minutes. Note that this analysis is performed on entire model collections instead of single models. The actual model-specific runtime of the query language can therefore not be ascertained from the paper.

The approach labelled *VisTrails* in [11] is a workflow model query approach that is designed to estimate the similarity of a given pattern to the existing workflow models [69]. Conceptually, the approach is based on labelled graphs and could therefore be transferred to other modelling languages. The patterns allow expressing queries to find isomorphic model fragments supporting the type and label information. The number of incoming or outgoing relations and comparisons of attributes are not supported. Furthermore, neither paths or loops nor the combination of sub-queries are supported. A runtime analysis is not given.

VMQL is a visual model query language that was originally designed to query UML models [70]. Extensions for other modelling languages exist as well (e.g., for BPMN [71]). The language allows for visually modelling a query in the same way a model is developed. Therefore, everything that can be modelled can be searched for. The language furthermore provides a

Table 4

Comparison of GMQL to other (model) query languages.

QL / Criterion	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13
aFSA	–	0	+	+	+	+	+	–	+	–	–	–	–
APQL	–	0	+	+	+	+	+	–	+	+	0	+	+
BeehiveZ	–	0	+	+	+	+	+	–	+	–	–	–	+
BPMN VQL	–	–	+	+	+	+	+	–	+	+	–	+	–
BPMN-Q	–	0	+	+	+	+	+	–	+	+	–	+	+
BPQL	–	–	+	+	–	–	–	+	+	–	–	+	–
BP-QL	–	–	+	+	+	+	+	–	+	+	–	–	–
Cypher (Neo4j)	+	0	+	+	+	+	+	+	+	+	+	+	–
EMF-IncQuery	–	0	+	+	+	–	–	+	+	+	0	+	+
IPM-PQL	–	0	+	+	+	–	–	–	+	+	–	–	–
OCL	+	0	+	+	+	–	+	+	+	–	–	+	–
PPSL	–	0	+	+	+	+	–	–	+	+	+	–	–
SPARQL	+	0	+	+	+	+	+	+	+	+	0	+	+
VisTrails	–	0	+	+	+	+	–	–	+	–	–	–	–
VMQL	+	+	+	+	+	+	+	–	+	+	+	–	+
Wise	–	0	–	+	–	–	–	–	–	–	–	–	–
GMQL	+	+	+	+	+	+	+	+	+	+	+	+	+

set of so-called constraints that allow for expressing particular restrictions on the returned pattern matches. Using these constraints, paths of arbitrary length that must or may not contain particular elements can be expressed. These constraints do not allow for comparing attribute values to one another or combining sub-structures. The preliminary performance analysis presented in [70] suggests that the matching process may take up to several minutes for “medium scale models” (p. 26). The runtime performance of this language cannot directly be compared to the one conducted here (different models and queries were used). However, in our experiments, we did not encounter runtimes this long. VMQL can be used to query all types of conceptual models and models developed in arbitrary languages.

Wise is a workflow search engine that allows for finding workflow model fragments based on a keyword search [72]. It therefore only incorporates label information in its matching process. The workflows are stored in a hierarchy allowing one workflow model to be linked to one superior workflow. The query result contains an aggregated graph of all identified workflows that include the keyword. Wise is designed to query hierarchical workflow models alone. The concept, however, can be transferred to other process modelling languages as well.

Table 4 demonstrates that there is indeed no model query language that completely fulfils all requirements for a generic model query language as deduced from the literature (cf. Section 2.6). The presented languages are therefore not able to express some of the queries that need to be run on conceptual model collections for various analysis purposes. In particular, features to express queries consisting of paths that must or may not contain certain elements are not present in many current model query languages. This, however, is an important feature in order to analyse more complex semantic relationships between model elements (see predecessor/successor compliance rule in Fig. 16). Comparing attribute values to one another is a further feature that many model query languages do not provide. Again, this is important in the context of design time compliance checking (cf. separation of duties pattern in Section 4.2).

The fact that no query language proposed in the literature covers all requirements of a generic model query language is not surprising, because these languages were simply not designed to be generic. Instead, many of these languages address a very specific problem, namely pattern matching in models of a specific modelling language. Naturally, a process query language like APQL is not suited to query data models. It provides constructs that are explicitly designed to support pattern matching in process models. Conversely, a language like EMF-IncQuery cannot be used to query process models. While these languages perfectly solve the particular problem they were meant to address, van der Aalst [15] argues that if model analysis approaches developed in the scientific community are ever to disseminate into corporate reality they have to be flexible enough to support a number of different model analysis scenarios and be applicable to a broad range of conceptual models. After all, an organization that uses EPCs to describe its processes cannot directly use a query language like BPMN-Q that supports querying BPMN models. Transformations have to be performed to transfer the source model into the particular notation required by the given query language. Such a transformation may be too complex and thus too costly to be conducted in corporate reality. Furthermore, such a transformation may not be wanted, because members of an organization may not want to change their preferred modelling notations. In addition, an organization would have to use a specialized query approach for each other model type used. In other words, it would need one query language for process models, one query language for data models, one query language for organizational charts, etc. Instead, van der Aalst argues that model analysis approaches are required that by default can be applied to models developed in any graph-based modelling language and to a broad range of analysis purposes [15]. GMQL is a step in this direction. The language is able to express pattern queries that support a broad range of model analysis scenarios (cf. Sections 2 and 4; additional sources containing pattern definitions for these scenarios leading to the same requirements can be found in [73–81]). It is furthermore able to express pattern queries for various model types (e.g. process models, data models, organizational charts) and modelling languages (e.g. EPC, BPMN, SBPML, ERM).

GMQL adopts concepts from meta-modelling [82]. GMQL incorporates type information contained in the meta-model of a given language. GMQL recognizes any model as an attributed graph with nodes and edges each having a set of attributes that further specify the semantics of a model element. As a modelling language essentially provides a set of node and edge types, GMQL uses the same constructs that can be used to specify modelling languages. In doing so, the abstract syntax of a given modelling language is not a constant of the query language any more. This means that the syntax is not “hard-coded” into the query language like in the works presented in [12] or [13]. Instead, the syntax of a given modelling language can be seen as a variable of the query language.

7.4. Summary and outlook

This paper introduced the generic model query language GMQL. In contrast to other model query languages discussed in the literature it is not limited to a particular modelling language or application scenario. It is based on the idea that any conceptual model can be represented in terms of its sets of objects and relationships. GMQL provides a number of functions and operators that takes these sets as input and perform particular operations on them. These functions and operators can be combined thus allowing for specifying arbitrary pattern queries. The paper presented a rigorous runtime analysis of GMQL based on the performance evaluation methodology outlined by Georges et al. [42]. The collected performance data suggest that GMQL queries can be executed within satisfactory time. If runtimes significantly longer than one millisecond were observed, large amounts of pattern occurrences were found. We conducted a multiple regression analysis proposing a statistical model that demonstrates the influence of both model size and number of pattern occurrences on the runtime behaviour. The model is able to explain up to 98.3 per cent of runtime variability. As GMQL is not restricted to a specific modelling language, it is broadly applicable to many model analysis scenarios that include pattern matching.

As part of future research, we will combine GMQL with terminological standardization procedures as outlined in [48,49] to turn it into full-fledged model analysis approach that is able to analyse model structure and semantics. In terms of runtime performance, we will conduct additional runtime measurements on extremely large models. We also aim at comparing GMQL with graph-theoretical algorithms for subgraph isomorphism as well as subgraph homeomorphism. Runtime analyses of algorithms for the former problem suggest that they perform well on conceptual models [48]. As subgraph isomorphism and subgraph homeomorphism are known to be computationally complex, many authors propose algorithms that exploit particular graph characteristics like planarity [83,84] or bounded tree-width [85] to achieve a runtime complexity that is polynomial in the size of the graph. Short-term future research will determine if respective algorithms can be adapted to comply with the language requirements in Section 2.6. If so, long-term future research will determine if these algorithms can be included in GMQL, especially to support the path functions. We also plan on extending our GMQL *implementation* in such a way as to allow for finding pattern occurrences that stretch across multiple models (e.g., a path that starts in model A and ends in model B, or a pattern that stretches across models of different type (cf., e.g., the pattern shown in Fig. 2). This can easily be achieved by extending the input set E accordingly. Another challenge will be the formula-based queries by visual (i.e., graphically “drawable”) ones.

Appendix

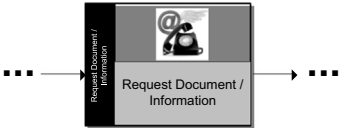
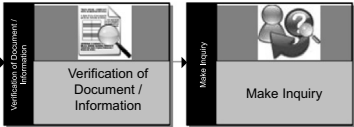
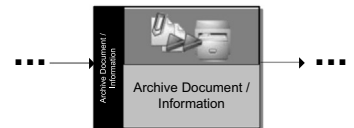
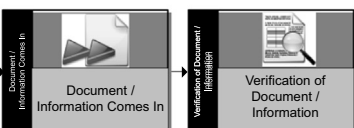
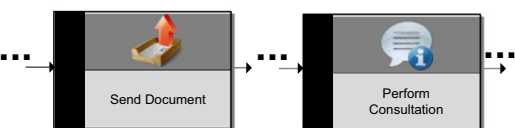
Pattern name	Pattern description	Exemplary pattern occurrences	Requirements for QL
Information deficit	An information deficit occurs whenever the process execution is interrupted, because insufficient information is available. The missing information first has to be retrieved manually (e.g., by making an inquiry to the customer) before the process can continue.	<p>a) </p> <p>b) </p>	<ul style="list-style-type: none"> • Identification of model elements having particular attributes with specific values (e.g., attribute = „label“; attribute value = „Make Inquiry“) (see R.1.1) • Identification of adjacent model elements (see R.1.4)
Manual activities with automation potential	Manual activities with automation potential describe (a sequence of) frequently occurring tasks that may already be partly supported by IT and that have the potential to be entirely executed automatically. Identifying these patterns serves to reduce human labor in routine activities.	<p>a) </p> <p>b) </p>	<ul style="list-style-type: none"> • Identification of model elements having particular attributes with specific values (e.g., attribute = „label“; attribute value = „Archive Document / Information“) (see R.1.1) • Identification of adjacent model elements (see R.1.4)
Document request causing consulting need	A document request causing a consulting need represents a situation in which a document has been sent out to a customer who later has to visit the administration office in order to fill out the document. Searching this pattern serves to identify document requests that need to be restructured so that the customer understands it better.		<ul style="list-style-type: none"> • Identification of model elements having particular attributes with specific values (e.g., attribute = „label“; attribute value = „Perform Consultation“) (see R.1.1) • Identification of paths between model elements (see R.1.5)

Fig. A1. Additional weakness patterns discussed in the literature (cf. [3] and [22]).

Pattern name	Pattern description	Exemplary pattern occurrences	Requirements for QL
Predecessor rule	The predecessor rule states that a particular activity B can only be executed if activity A has been executed before. For instance, before a customer can open a bank account information about her financial status has to be obtained. The model on the right-hand side contains a violation of that rule. It is represented by the path from the first activity to the „open account“ activity that does not contain „obtain customer info“.		<ul style="list-style-type: none"> • Identification of element paths that do not contain particular elements • Identification of model elements having particular attributes with specific values (e.g. attribute = „label“; attribute value = „Open account“)
Start action / End action	The start action and the end action rules require particular activities to be the start or target elements of their respective process models. The exemplary BPMN model excerpt on the right contain violations of that rule given that B and C are required to be the start and end actions. Such patterns can be found by analysing (the number of) outgoing and ingoing arcs of a given element.		<ul style="list-style-type: none"> • Identification of elements having outgoing or ingoing arcs • Identification of elements having a certain number of outgoing or ingoing arcs
Mandatory / forbidden intermediary rule	These rules require that a mandatory / forbidden activity needs to be / must not be executed between two succeeding activities. This rule translates to a path of elements that must (not) include certain other elements. In the BPMN model excerpt on the right the forbidden intermediary rule is violated given that B represents the forbidden activity on a path from A to C. In case B represented a mandatory activity, the path from A to C not containing B would represent a compliance violation.		<ul style="list-style-type: none"> • Identification of element paths that contain particular elements • Identification of element paths that do not contain particular elements

Fig. A2. Additional compliance patterns discussed in the literature (cf. [24] and [23]).

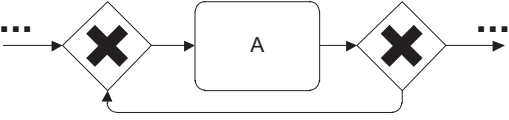
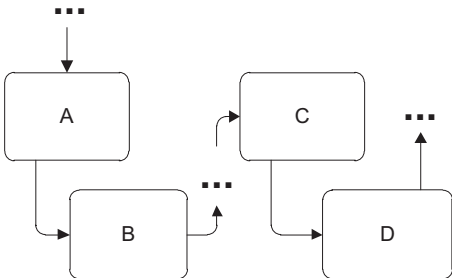
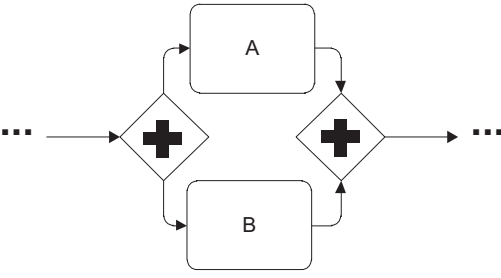
Pattern name	Pattern description	Exemplary pattern occurrences	Requirements for QL
REPEAT pattern	Similarly to the WHILE pattern, this structure represents a situation in which an activity is executed as long as a particular condition applies. In terms of graph structure, this pattern again translates into an element loop.		<ul style="list-style-type: none"> • Identification of element paths that start and end in the same element (i.e. loops) (see R1.5)
SEQUENCE pattern	The SEQUENCE pattern represents a linear sequence of activities. In terms of the graph structure, this pattern simply translates to an element path of arbitrary length.		<ul style="list-style-type: none"> • Identification of element paths of arbitrary length (see R1.5)
FLOW pattern	The FLOW pattern represents a single entry single exit (SESE) fragment whose entry and exit nodes are gateways. In terms of the graph structure, this pattern relates to two paths that start and end in gateways and that contain the same start and target node.		<ul style="list-style-type: none"> • Identification of element paths (R1.5) • Identification of elements having a particular number of ingoing or outgoing relationships (see R1.2) • Possibility to join substructures that have particular elements in common (R2)

Fig. A3. Additional compliance patterns discussed in the literature (cf. [23,24]).



Fig. A4. Many-to-many relationship type pattern.

Model translation also plays an important role in database management. In this context, conceptual data models like ER models are translated to relational schemas that are then used to implement a database, for instance. Pattern matching can be used in this context to detect particular fragments within an ER model that can be automatically translated to a relational schema. Batra [86] proposes a set of such database patterns. Translating them is facilitated by a set of rules that define which elements of the ER model are to be translated to elements of the schema and how [87]. Such a 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, ER model fragments need to be found that represent many-to-many relationship types. An example is depicted as an excerpt of an ER model in Fig. A4. This fragment consists of a relationship type that is directly related to two entity types. The edges connecting the relationship and entity types carry specific labels representing the cardinality of the relationship. A generic model query language designed to find such a pattern consequently needs to be able to express pattern queries consisting of adjacent elements (see R1.4). It furthermore needs to be able to express queries consisting of elements that have particular attributes (like a label) with specific values (like “(0,n)” (see R1.1)).

Pattern name	Pattern description	Exemplary pattern occurrences	Requirements for QL
Deadlock pattern	A deadlock represents a situation in which different branches of a process are waiting for one another thus inhibiting process execution. Such a structure can occur if split gateways are not joined appropriately. In terms of graph structure, we are again interested in determining paths that begin in elements having a particular type and a certain number of outgoing or ingoing arcs (in this case the number of arcs needs to be greater than one).		<ul style="list-style-type: none"> • Identification of element paths of arbitrary length (see R1.5) • Identification of elements having a particular number of outgoing or ingoing arcs (see R1.2) • Identification of elements having a particular type (e.g., XOR gateway, AND gateway) (see R1.1)
XOR-Control pattern	This structure contains an XOR split that is followed by an AND join over an element path of arbitrary length. The AND join is succeeding a start event. At runtime, such a structure can cause a problem, because the AND join will not conclude until both its ingoing branches are completed. This, however, will never occur if the process ran into the upper branch of the XOR.		<ul style="list-style-type: none"> • Identification of element paths of arbitrary length (see R1.5) • Identification of adjacent elements (see R1.4) • Identification of elements having a particular number of outgoing or ingoing arcs (see R1.2) • Possibility to combine particular substructures (see R2)
XOR-Split – OR JOIN	A common syntactical error in EPC diagrams represents an XOR split that is not correctly joined by a corresponding XOR join. In the example on the right, this error translates to a path from an XOR split to an OR join. This paths must not contain any XOR joins. To determine splitting and joining nodes the number of outgoing and ingoing arcs furthermore needs to be considered.		<ul style="list-style-type: none"> • Identification of element paths (see R1.5) • Identification of element paths that do not contain particular elements (See R1.5) • Elements having (a particular number of) ingoing or outgoing arcs (see R1.2)

Fig. A5. Additional syntactical error patterns discussed in the literature (cf. [27,28]).

Table A1 (continued)

Query Name and Description	Pattern Query
	<pre> UNION (ElementsOfType (O, OR) , ElementsOfType (O, XOR)) , 0))))) </pre>
<i>Longest Sequence of Activities (LSA)</i> : The query returns the longest paths that either start or end in an event or function object and do not contain any gateways	<pre> LongestDirectedPaths (O, O) INTERSECTIONDirectedPathsNotContainingElements (UNION (ElementsOfType (O, Event) , ElementsOfType (O, Function)) , UNION (ElementsOfType (O, Event) , ElementsOfType (O, Function)) , UNION (UNION (ElementsOfType (O, AND) , ElementsOfType (O, OR)) , ElementsOfType (O, XOR))) </pre>
<i>XOR split directly related to XOR join (XOR)</i> : The query returns structures consisting of an XOR split, an XOR join, and the relationship connecting them.	<pre> AdjacentSuccessors (COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 0))))) , COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 0)))))) </pre>
<i>SESE XOR fragment (SESE)</i> : The query returns a SESE fragment with XOR gateways as entry and exit objects. The paths between these objects must not contain gateways.	<pre> JOIN (DirectedPathsNotContainingElements (COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 0))))) , COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 0)))))) , UNION (UNION (ElementsOfType (O, XOR) , ElementsOfType (O, OR)) , ElementsOfType (O, AND)))) DirectedPathsNotContainingElements (COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 0))))) , COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 0)))))) , </pre>

Table A1 (continued)

Query Name and Description	Pattern Query
<p><i>SESE XOR fragment containing only events (SESEE):</i> The query returns a SESE fragment with XOR gateways as entry and exit objects. The paths between these objects must only contain event objects.</p>	<pre> UNION (UNION (ElementsOfType (O, XOR) , ElementsOfType (O, OR)) , ElementsOfType (O, AND)))) JOIN (DirectedPathsNotContainingElements (COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 0)))) , COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 0))))) , UNION (UNION (UNION (ElementsOfType (O, XOR) , ElementsOfType (O, OR)) , ElementsOfType (O, AND)) , ElementsOfType (O, Function))) DirectedPathsNotContainingElements (COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 0)))) , COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfPredRelations (ElementsOfType (O, XOR) , 0))))) , UNION (UNION (UNION (ElementsOfType (O, XOR) , ElementsOfType (O, OR)) , ElementsOfType (O, AND)) , ElementsOfType (O, Function))))) </pre>
<p><i>XOR split OR join (XsOj):</i> This query returns paths starting in an XOR split and ending in an OR join. The query is based on an EPC syntax error reported in [27].</p>	<pre> DirectedPathsNotContainingElements (COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 0)))) , COMPLEMENT (ElementsOfType (O, OR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, OR) , UNION (ElementsWithNumberOfPredRelations (ElementsOfType (O, OR) , 1) , ElementsWithNumberOfPredRelations (ElementsOfType (O, OR) , 0))))) , COMPLEMENT (ElementsOfType (O, XOR) , SELFUNION (INNERINTERSECTION (ElementsOfType (O, XOR) , UNION (ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 1) , ElementsWithNumberOfSuccRelations (ElementsOfType (O, XOR) , 0)))))) </pre>

Table A1 (continued)

Query Name and Description	Pattern Query
<i>XOR split XOR join (XsXj)</i> : This query returns paths starting in an XOR split and ending in an XOR join. It represents the syntactically correct version of the query given above.	<pre>DirectedPathsNotContainingElements(COMPLEMENT(ElementsOfType(O,XOR), SELFUNION(INNERINTERSECTION(ElementsOfType(O,XOR), UNION(ElementsWithNumberOfSuccRelations(ElementsOfType(O,XOR),1), ElementsWithNumberOfSuccRelations(ElementsOfType(O,XOR),0))))), COMPLEMENT(ElementsOfType(O,XOR), SELFUNION(INNERINTERSECTION(ElementsOfType(O,XOR), UNION(ElementsWithNumberOfPredRelations(ElementsOfType(O,XOR),1), ElementsWithNumberOfPredRelations(ElementsOfType(O,XOR),0))))), COMPLEMENT(ElementsOfType(O,XOR), SELFUNION(INNERINTERSECTION(ElementsOfType(O,XOR), UNION(ElementsWithNumberOfSuccRelations(ElementsOfType(O,XOR),1), ElementsWithNumberOfSuccRelations(ElementsOfType(O,XOR),0))))))</pre>

Table A2
ERM pattern queries.

Query name and description	Pattern query
<i>n</i> -ary Relationship Type (nRT): The query returns relationship types that are directly related to at least three entity types. One inner set will contain one relationship type, its adjacent entity types, and the edges connecting them.	<pre> ElementsDirectlyRelated(SELFUNION (INNERINTERSECTION(O, COMPLEMENT (ElementsWithRelationsOfType(ElementsOfType(O, RelationshipType), R, ER), UNION (ElementsWithNumberOfRelationsOfType(ElementsOfType(O, RelationshipType), ER, 2), ElementsWithNumberOfRelationsOfType(ElementsOfType(O, RelationshipType), ER, 1))))), ElementsOfType(O, EntityType)) </pre>
<i>Ternary Relationship Type (TR)</i> : The query returns relationship types that are directly related to exactly three entity types. One inner set will contain one relationship type, its adjacent entity types, and the edges connecting them.	<pre> ElementsDirectlyRelated(SELFUNION (INNERINTERSECTION(O, ElementsWithNumberOfRelationsOfType(ElementsOfType(O, RelationshipType), ER, 3))), ElementsOfType(O, EntityType)) </pre>
<i>Binary Relationship Type (BR)</i> : The query returns relationship types that are directly related to exactly two entity types. One inner set will contain one relationship type, its adjacent entity types, and the edges connecting them.	<pre> ElementsDirectlyRelated(SELFUNION (INNERINTERSECTION(O, ElementsWithNumberOfRelationsOfType(ElementsOfType(O, RelationshipType), ER, 2))), ElementsOfType(O, EntityType)) </pre>
<i>ER paths (ERP)</i> : The query will return all paths that start in an entity type and end in a relationship type.	<pre> Paths(ElementsOfType(O, EntityType), ElementsOfType(O, RelationshipType)) </pre>
<i>ER Loops (ERL)</i> : The query will return loops that start and end in the same entity type and do not contain relational entity types.	<pre> LoopsNotContainingElements(ElementsOfType(O, EntityType), ElementsOfType(O, RelationalEntityType)) </pre>
<i>Hierarchy (H)</i> : The query will return loops that start and end in the same entity type and do not contain either entity types or relational entity types.	<pre> LoopsNotContainingElements(ElementsOfType(O, EntityType), UNION(ElementsOfType(O, RelationalEntityType), ElementsOfType(O, EntityType))) </pre>
<i>Longest Paths (LP)</i> : The query will determine the longest paths that start and end in entity types.	<pre> LongestPaths(ElementsOfType(O, EntityType), ElementsOfType(O, EntityType)) </pre>
<i>Shortest Paths (SP)</i> : The query will determine the shortest paths that start and end in entity types.	<pre> ShortestPaths(ElementsOfType(O, EntityType), ElementsOfType(O, EntityType)) </pre>
<i>Receipt Structure (RS)</i> : The query will return a receipt structure as depicted in Fig. 6a).	<pre> JOIN(PathsNotContainingElements(ElementsOfType(O, EntityType), ElementsOfType(O, RelationalEntityType), UNION(ElementsOfType(O, EntityType), ElementsOfType(O, RelationalEntityType))), ElementsDirectlyRelated(INNERINTERSECTION(ElementsWithNumberOfRelationsOfType(O, ERT, 2), ElementsOfType(O, RelationalEntityType)), ElementsOfType(O, EntityType))) </pre>
<i>Specialization / Generalization (SG)</i> : The query returns structures consisting of a generalization object, entity types, and the edges connecting them.	<pre> ElementsDirectlyRelated(ElementsOfType(O, Generalization), ElementsOfType(O, EntityType)). </pre>

Table A3

Load time in milliseconds for EPC pattern queries using PostgreSQL and SQLite.

Pattern database	AND	CL	DSAE	DSAF	LSA	SESE	SESEE	XOR	XsOj	XsXj
PostgreSQL	1,3	1,1	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,1
SQLite	2,1	2,0	2,0	2,0	2,0	3,0	3,0	2,2	2,2	2,8

Table A4

Load time in milliseconds for ERM pattern queries using PostgreSQL and SQLite.

Pattern database	BR	ERL	ERP	H	LP	nRT	RS	SG	SP	TR
PostgreSQL	1,0	1,0	1,0	1,0	1,0	2,2	1,0	1,0	1,0	1,0
SQLite	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0

Table A5

Load time in milliseconds for models using PostgreSQL (PgSQL) and SQLite.

Number of elements	PgSQL [ms]	SQLite [ms]	Number of elements	PgSQL [ms]	SQLite [ms]	Number of elements	PgSQL [ms]	SQLite [ms]
15	274.1	298.3	42	718.2	726.6	67	1071	1122
16	284	325.7	43	706.5	761.8	67	1123.8	1112.7
18	321.2	370.3	43	733.6	786.3	71	1177	1197.4
19	342.2	375.4	44	764.5	763.4	72	1187.8	1256.8
20	350.4	357.9	44	732.5	742.9	78	1256.4	1281.8
21	354.2	424.3	45	766.8	780.7	78	1289.8	1341.2
22	389.6	408.4	45	747.5	768.7	82	1346.6	1383.9
22	1889.4	432.8	46	812.1	838.4	83	1415.2	1464.4
23	389.5	441.8	46	753.1	756.7	84	1363.4	1489.1
23	395.7	405.7	46	736.9	784.9	85	1423.1	1411
23	412.2	411.8	46	735.5	815.1	86	1411.9	1479.2
24	423.7	459.6	47	794.7	840.6	87	1441.5	1463.7
24	432.6	436.9	48	797.4	836.9	87	1454.4	1480.9
25	428.5	448.5	48	797.1	841	89	1559.2	1486.8
25	432.2	473.2	50	837.2	857.3	92	1497.9	1565.5
27	448.7	479.5	51	857.9	862.1	97	1576.9	1674.8
29	497.8	516.9	52	881.5	917	102	1659.6	1766.3
30	504.9	517.4	53	871.8	928.9	102	1701.6	1818.3
31	528.9	594	55	948.2	971.9	102	1704.3	1735.9
31	512.1	535.6	56	959.2	1009.2	107	1750	1921.9
32	540.4	553.7	59	984.3	1037.1	109	1831.6	1874.6
33	558.5	584.4	59	1259.8	1177.4	111	1810.8	1944.6
33	558.3	560.5	60	1007.8	1023.9	113	1794.6	1884.1
34	567.5	599	60	976.4	1028.8	119	1948.8	1970.3
34	629.6	665.1	61	961.7	1035.6	120	2049.9	2087.7
35	630	683.1	62	995.4	1043.9	122	2124.7	2153.3
35	557.8	589.9	63	1054.2	1055.9	137	2217.7	2326.4
36	626.2	623.8	63	1021.3	1097	155	2506.3	2642.6
37	597.9	638.8	63	999.4	1154.2	189	3039.1	3138.7
37	648.1	1592.2	66	1101.8	1139.2	204	3373.2	3494.3
39	638.2	699.7	66	1049	1146.2	221	3583.8	3979
41	660	714.5	66	1106.2	1137.7	240	4138.6	4032.3
41	659.8	719.1	67	1114.2	1127	264	4316.9	4696.2

Table A6

Runtime measurements and number of pattern matches for ERMs.

Query and # of model elements	BR		ERP		ERL		H		LP		nRT		RS		SP		SG		TR	
	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P
97	0.22	4	9.54	288	36.08	586	4.90	116	48.96	80	0.19	1	0.33	4	49.90	118	0.13	8	0.16	1
87	0.31	10	58.47	1472	30.48	328	1.48	44	64.67	12	0.31	0	0.55	10	64.85	50	0.10	4	0.13	0
78	0.16	2	39.13	1030	4.74	98	2.02	56	122.49	32	0.15	0	0.56	11	122.79	66	0.10	5	0.12	0
71	0.13	1	88.31	3214	0.45	2	0.40	2	114.00	24	0.14	0	0.62	36	115.05	16	0.08	0	0.11	0
67	0.14	2	45.54	1536	1.24	20	0.75	14	103.34	12	0.15	0	0.53	19	103.46	32	0.08	2	0.11	0
67	0.27	10	57.95	2649	4.86	120	1.01	30	56.45	2	0.30	0	0.28	3	56.55	34	0.08	3	0.12	0
66	0.24	9	29.30	1450	8.17	234	0.95	28	37.59	2	0.26	0	0.37	11	37.93	34	0.08	3	0.11	0
63	0.15	3	26.34	862	2.93	90	0.82	26	48.18	12	0.15	1	0.54	5	48.50	30	0.08	2	0.13	1
63	0.16	4	328.49	16875	1.13	28	0.62	20	562.58	30	0.15	0	0.74	35	565.44	42	0.08	1	0.11	0
62	0.18	5	6.90	340	23.05	884	1.31	48	19.07	8	0.16	0	0.19	0	19.27	48	0.09	7	0.10	0
61	0.15	3	54.66	1306	2.70	72	0.94	36	169.74	44	0.13	0	0.65	11	170.51	52	0.08	1	0.10	0
60	0.14	2	39.64	2170	0.96	18	0.56	12	166.53	4	0.13	0	0.47	11	167.61	28	0.08	4	0.10	0
59	0.23	9	12.36	581	17.90	590	1.25	52	17.44	28	0.23	1	0.09	0	18.49	52	0.08	2	0.13	1
51	0.16	5	7.52	533	3.40	158	0.55	22	30.60	6	0.15	0	0.35	0	30.66	22	0.08	4	0.10	0
50	0.11	1	3.75	160	0.43	8	0.34	6	24.43	2	0.11	0	0.32	6	24.43	10	0.07	2	0.09	0
48	0.17	6	2.96	156	0.82	26	0.43	14	3.16	6	0.17	0	0.28	3	3.17	16	0.07	1	0.10	0
47	0.11	2	2.55	212	1.54	50	0.93	40	10.20	12	0.11	0	0.29	14	10.30	44	0.07	2	0.09	0
46	0.13	4	1.29	75	5.03	212	1.19	58	4.81	60	0.13	0	0.09	0	4.81	58	0.08	4	0.09	0
46	0.09	0	0.59	44	0.50	8	0.39	8	3.76	32	0.11	0	0.36	4	3.69	12	0.08	3	0.09	0
46	0.10	1	4.26	315	0.93	30	0.43	14	8.80	4	0.12	0	0.30	9	8.83	18	0.07	2	0.09	0
45	0.13	4	1.74	129	1.63	50	0.60	22	4.69	4	0.13	0	0.21	4	4.71	24	0.07	3	0.09	0
45	0.12	3	5.19	347	0.58	16	0.41	12	12.28	4	0.12	0	0.25	9	12.25	46	0.07	1	0.09	0
44	0.09	0	4.37	337	0.41	8	0.33	8	15.37	16	0.11	0	0.22	0	15.48	14	0.07	2	0.09	0
43	0.09	0	16.02	1342	0.23	2	0.22	2	50.13	12	0.10	0	0.26	5	50.29	30	0.07	1	0.09	0
42	0.14	5	2.22	153	0.68	22	0.37	18	2.39	4	0.09	0	0.21	5	2.40	22	0.07	2	0.09	0
41	0.11	2	1.16	98	0.60	24	0.34	12	6.15	6	0.11	0	0.27	0	6.19	12	0.07	4	0.09	0
41	0.08	0	21.47	1288	0.09	0	0.10	0	4.42	88	0.09	0	0.40	98	4.40	14	0.06	0	0.08	0
39	0.13	4	1.37	96	2.41	92	0.68	40	3.08	16	0.13	0	0.17	3	3.08	42	0.07	2	0.09	0
37	0.10	1	2.93	196	0.45	16	0.27	8	7.04	8	0.10	0	0.26	6	6.90	18	0.07	1	0.08	0
35	0.08	0	6.23	680	0.09	0	0.09	0	2.44	72	0.08	0	0.32	72	2.43	12	0.06	0	0.08	0
34	0.13	6	0.98	70	1.69	92	0.53	32	1.65	60	0.15	0	0.07	0	1.64	32	0.06	1	0.08	0
33	0.09	1	0.23	12	2.13	132	0.69	40	2.05	36	0.08	0	0.07	0	2.04	40	0.07	4	0.08	0
32	0.07	0	0.47	60	0.45	20	0.34	20	4.77	8	0.08	0	0.29	14	4.79	32	0.06	1	0.08	0
31	0.09	2	0.76	80	0.42	12	0.24	6	1.97	8	0.10	0	0.17	3	1.94	14	0.06	1	0.08	0
31	0.07	0	0.22	20	0.58	26	0.45	26	1.44	20	0.08	0	0.20	7	1.44	28	0.06	2	0.07	0
30	0.12	5	0.88	88	0.53	24	0.24	12	0.86	16	0.13	0	0.14	3	0.86	14	0.06	1	0.08	0
27	0.13	8	0.50	42	0.21	18	0.16	16	0.21	2	0.18	0	0.07	0	0.21	16	0.05	0	0.07	0
25	0.06	0	0.05	0	0.28	8	0.26	8	1.57	12	0.07	0	0.13	0	1.57	16	0.06	2	0.06	0
25	0.08	1	1.30	201	0.22	8	0.16	8	3.18	12	0.09	1	0.17	4	3.17	22	0.05	0	0.08	1
24	0.07	0	0.13	12	0.15	2	0.14	2	0.67	2	0.08	0	0.17	6	0.67	6	0.05	1	0.07	0
23	0.07	0	1.00	152	0.08	0	0.09	0	1.50	8	0.07	0	0.19	24	1.50	8	0.05	0	0.07	0
23	0.06	0	1.01	152	0.08	0	0.08	0	1.52	8	0.07	0	0.19	24	1.51	8	0.05	0	0.07	0
22	0.06	0	0.23	26	0.10	0	0.11	0	1.07	26	0.07	0	0.12	0	1.07	18	0.05	0	0.06	0
21	0.08	1	0.19	33	0.56	44	0.25	22	0.54	4	0.08	1	0.07	0	0.55	22	0.05	2	0.08	1
20	0.06	0	0.23	24	0.11	0	0.11	0	0.61	4	0.07	0	0.13	4	0.62	22	0.05	0	0.07	0
16	0.06	0	0.08	6	0.07	0	0.07	0	0.17	2	0.06	0	0.11	3	0.16	2	0.05	0	0.06	0

Table A7

Runtime measurements and number of pattern matches for EPCs.

Query and # of model elements	AND		CL		DSAE		DSAF		LSA		XOR		SESE		SESEE		XsOj		XsXj	
	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P	RT	#P
264	71.78	447	1.39	2	0.64	1	0.63	15	16.60	1	0.50	0	1.66	50	1.13	21	1.91	22	1.91	22
240	32.65	477	1.60	6	0.60	0	0.60	9	9.58	2	0.40	0	1.33	36	0.87	12	1.71	25	1.71	25
221	0.39	0	2.42	22	0.67	0	0.66	9	5.65	1	0.47	0	1.27	29	1.10	20	2.33	29	2.33	29
204	5.20	135	1.25	5	0.45	0	0.45	9	4.99	1	0.34	0	0.78	27	0.71	24	1.24	17	1.24	17
189	0.28	0	1.21	12	0.45	0	0.44	11	4.76	1	0.34	0	0.84	16	0.68	6	1.43	24	1.43	24
155	0.21	0	1.20	22	0.30	0	0.31	3	5.22	1	0.27	0	0.57	18	0.53	14	1.10	16	1.09	16
137	0.20	0	0.68	4	0.29	0	0.29	10	2.28	3	0.29	1	0.77	30	0.61	16	0.82	15	0.82	15
122	0.17	0	0.44	5	0.24	0	0.24	5	3.58	1	0.19	0	0.45	16	0.31	8	0.46	8	0.46	8
120	0.21	0	0.51	3	0.29	0	0.29	5	1.71	1	0.26	0	0.81	41	0.68	32	1.00	28	1.00	28
119	0.18	0	0.92	15	0.25	0	0.25	5	1.61	1	0.22	0	0.47	13	0.39	9	0.69	14	0.69	14
113	0.16	0	0.37	2	0.21	0	0.21	5	2.54	1	0.19	0	0.47	26	0.35	12	0.69	12	0.69	12
111	0.15	0	0.27	0	0.20	0	0.20	6	1.83	1	0.19	0	0.49	34	0.35	16	0.40	10	0.39	10
109	0.52	0	0.58	10	0.20	0	0.20	2	2.12	1	0.16	0	0.27	13	0.27	13	0.46	5	0.46	5
107	0.16	0	0.47	4	0.20	0	0.20	3	1.46	1	0.14	0	0.21	5	0.19	2	0.32	9	0.32	9
102	0.16	0	0.46	6	0.21	0	0.21	4	1.58	2	0.19	0	0.45	13	0.32	2	0.47	12	0.47	12
102	0.16	0	0.36	3	0.20	0	0.20	5	1.50	4	0.15	1	0.27	14	0.25	10	0.38	6	0.38	6
102	0.15	0	0.25	0	0.20	0	0.20	5	2.54	3	0.18	0	0.36	12	0.32	9	0.38	8	0.39	8
92	0.13	0	0.27	0	0.14	0	0.14	5	1.31	1	0.15	0	0.23	4	0.21	1	0.24	2	0.24	2
89	0.14	0	0.30	2	0.18	0	0.18	4	0.97	1	0.16	0	0.29	8	0.26	5	0.29	4	0.29	4
87	0.13	0	0.24	2	0.15	0	0.15	3	1.88	2	0.13	0	0.26	12	0.21	3	0.30	6	0.30	6
86	0.18	0	0.21	0	0.12	0	0.12	6	2.00	1	0.14	0	0.21	4	0.19	1	0.18	2	0.18	2
85	0.14	0	0.26	3	0.18	0	0.17	5	1.23	1	0.17	0	0.38	10	0.28	2	0.37	11	0.37	11
84	0.13	0	0.54	16	0.15	0	0.14	2	1.05	1	0.14	0	0.27	9	0.22	4	0.29	8	0.29	8
83	0.20	2	0.27	3	0.19	0	0.19	4	1.40	1	0.13	1	0.21	8	0.20	5	0.22	4	0.22	4
82	0.23	0	0.21	0	0.15	0	0.14	5	1.37	1	0.15	0	0.29	12	0.24	6	0.26	6	0.27	6
78	0.13	0	0.28	3	0.16	0	0.16	3	1.05	1	0.14	2	0.25	12	0.21	3	0.35	6	0.35	6
72	0.12	0	0.18	0	0.14	0	0.14	4	1.01	1	0.14	0	0.29	24	0.28	24	0.25	6	0.25	6
67	0.12	0	0.19	1	0.14	0	0.14	4	0.84	1	0.14	1	0.32	17	0.22	3	0.27	7	0.26	7
66	0.11	0	0.13	1	0.09	0	0.09	0	1.68	1	0.10	0	0.10	0	0.11	0	0.10	0	0.10	0
66	0.18	6	0.22	1	0.13	0	0.13	3	0.61	1	0.13	0	0.24	12	0.23	9	0.24	6	0.24	6
63	0.11	0	0.15	1	0.12	0	0.12	4	1.02	1	0.11	0	0.19	5	0.17	2	0.17	5	0.17	5
60	0.14	0	0.15	1	0.11	0	0.11	2	1.13	1	0.11	0	0.17	7	0.16	4	0.14	5	0.14	5
59	0.10	0	0.15	2	0.11	0	0.11	1	0.31	3	0.10	0	0.13	1	0.14	1	0.13	2	0.13	2
56	0.09	0	0.10	0	0.09	0	0.09	1	1.52	1	0.09	0	0.10	0	0.10	0	0.09	0	0.09	0
55	0.11	0	0.20	3	0.12	0	0.12	3	0.46	1	0.11	1	0.15	1	0.15	1	0.15	3	0.15	3
53	0.15	0	0.14	1	0.11	0	0.11	2	0.69	1	0.11	0	0.19	8	0.16	1	0.18	4	0.18	4
52	0.10	0	0.11	0	0.11	0	0.11	1	0.69	1	0.10	0	0.14	4	0.13	1	0.14	2	0.14	2
48	0.09	0	0.16	2	0.09	0	0.09	1	0.35	5	0.09	0	0.11	0	0.12	0	0.13	0	0.13	0
46	0.13	0	0.12	3	0.10	0	0.10	1	0.34	2	0.09	0	0.13	4	0.12	1	0.11	2	0.11	2
44	0.13	2	0.12	1	0.09	0	0.09	2	0.45	1	0.09	1	0.13	4	0.12	1	0.11	2	0.12	2
43	0.10	0	0.15	2	0.10	0	0.10	1	0.19	1	0.09	0	0.13	4	0.13	4	0.12	4	0.12	4
37	0.08	0	0.09	0	0.08	0	0.08	1	0.66	1	0.08	0	0.09	0	0.09	0	0.08	0	0.08	0
36	0.08	0	0.08	0	0.07	0	0.07	0	0.57	1	0.07	0	0.08	0	0.08	0	0.07	0	0.07	0
35	0.08	0	0.08	0	0.07	0	0.07	0	0.54	1	0.07	0	0.07	0	0.08	0	0.07	0	0.07	0
34	0.09	0	0.12	2	0.09	0	0.09	0	0.21	1	0.08	0	0.09	0	0.09	0	0.08	0	0.08	0
33	0.10	0	0.13	3	0.08	0	0.08	0	0.16	1	0.07	0	0.08	0	0.09	0	0.08	0	0.08	0
29	0.09	0	0.09	1	0.07	0	0.07	1	0.22	2	0.07	0	0.08	0	0.08	0	0.07	0	0.07	0
24	0.08	0	0.09	0	0.07	0	0.07	1	0.12	1	0.08	0	0.11	4	0.11	4	0.09	2	0.09	2
23	0.07	0	0.07	0	0.06	0	0.07	1	0.26	1	0.07	0	0.07	0	0.08	0	0.07	0	0.07	0
22	0.07	0	0.07	0	0.07	0	0.07	1	0.14	1	0.07	0	0.11	4	0.10	1	0.09	2	0.09	2
19	0.08	0	0.06	0	0.06	0	0.06	0	0.13	0	0.06	0	0.06	0	0.07	0	0.06	0	0.06	0
18	0.07	0	0.07	0	0.07	0	0.07	1	0.12	3	0.06	0	0.06	0	0.07	0	0.06	0	0.06	0
15	0.07	0	0.07	3	0.07	0	0.07	0	0.08	0	0.06	0	0.07	0	0.07	0	0.06	0	0.06	0

References

- [1] R.M. Dijkman, M. La Rosa, H.A. Reijers, Managing large collections of business process models – current techniques and challenges, *Comput. Ind.* 63 (2) (2012) 91–97.
- [2] M. Steinhorst, D. Breuker, P. Delfmann, H.-A. Dietrich, Supporting enterprise transformation using a universal model analysis approach, in: *Proceedings of the European Conference on Information Systems (ECIS)*, 2012.
- [3] J. Becker, P. Bergener, D. Breuker, M. Räckers, An empirical assessment of the usefulness of weakness patterns in business process redesign, in: *Proceedings of the European Conference on Information Systems (ECIS)*, 2012.
- [4] S. Sadiq, G. Governatori, K. Namiri, Modeling control objectives for business process compliance, in: G. Alonso, P. Dadam, M. Rosemann (Eds.), *Business Process Management*, Springer, Berlin, Heidelberg, 2007, pp. 149–164.

- [5] L. García-Bañuelos, Pattern identification and classification in the translation from BPMN to BPEL, in: R. Meersman, Z. Tari (Eds.), *On the Move to Meaningful Internet Systems: OTM 2008*, Springer, Berlin, Heidelberg, Monterrey, Mexico, 2008, pp. 436–444.
- [6] Object Management Group, BPMN 2.0 Specification. URL (<http://www.omg.org/spec/BPMN/2.0/PDF/>) (accessed 12.09.13).
- [7] M. Nüttgens, Syntax and Semantik Ereignisgesteuerter Prozessketten (EPK), in: J. Desel, M. Weske (Eds.), *Promise 2002 – Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen*, Proceedings des GI-Workshops und Fachgruppentreffens, Potsdam, Germany, 2002, pp. 64–77.
- [8] J. Desel, G. Juhás, “What is a Petri Net?” Informal Answers for the Informed Reader, in: H. Ehrig, J. Padberg, G. Juhás, G. Rozenberg (Eds.), *Unifying Petri Nets*, Springer, Berlin, Heidelberg, 2001, pp. 1–25.
- [9] P.P.-S. Chen, The entity-relationship model – toward a unified view of data, *ACM Trans. Database Syst.* 1 (1) (1976) 9–36.
- [10] Object Management Group, Unified Modeling Language 2.4.1 Specification, URL: (<http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/>) (accessed 12.09.12).
- [11] J. Wang, T. Jin, R.K. Wong, L. Wen, Querying business process model repositories, *World Wide Web* 17 (3) (2013) 427–454.
- [12] C. Beerl, A. Eyal, S. Kamenkovich, T. Milo, Querying business processes with BP-QL, *Inf. Syst.* 33 (6) (2008) 477–507.
- [13] M. Momotko, K. Subieta, Process query language: a way to make workflow processes more flexible, in: A. Benczúr, J. Demetrovics, G. Gottlob (Eds.), *Advances in Databases and Information Systems*, Springer, Berlin, Heidelberg, Budapest, Hungary, 2004, pp. 306–321.
- [14] A. Awad, 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 (EMISA'07)*, St. Goar, Germany, 2007, pp. 115–128.
- [15] W.M.P. van der Aalst, Business process management: a comprehensive survey, *ISRN Softw. Eng.* (2013) 1–37.
- [16] B. Weiß, A. Winkelmann, A metamodel based perspective on the adaptation of a semantic business process modeling language to the financial sector, in: *Proceedings of the 11th 44th Hawaii International Conference on System Sciences*, IEEE, 2011, pp. 1–10.
- [17] P. Delfmann, S. Herwig, L. Lis, A. Stein, K. Tent, J. Becker, Pattern specification and matching in conceptual models – a generic approach based on set operations, *Enterpr. Model. Inf. Syst. Archit.* 5 (3) (2010) 24–43.
- [18] H.-A. Dietrich, M. Steinhorst, J. Becker, P. Delfmann, Fast pattern matching in conceptual models – evaluating and extending a generic approach, in: M. Nüttgens, O. Thomas, B. Weber (Eds.), *Enterprise Modelling and Information Systems Architectures (EMISA)*, GI, 2011, pp. 79–92.
- [19] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub)graph isomorphism algorithm for matching large graphs, *IEEE Trans. Pattern Anal. Mach. Intell.* 26 (10) (2004) 1367–1372.
- [20] A. Gupta, N. Nishimura, Sequential and parallel algorithms for embedding problems on classes of partial k -trees, in: E. Schmidt, S. Skyum (Eds.), *Algorithm Theory – SWAT'94*, Springer, Berlin, Heidelberg, 1994, pp. 172–182.
- [21] S. Smirnov, H.A. Reijers, M. Weske, From fine-grained to abstract process models: a semantic approach, *Inf. Syst.* 37 (8) (2012) 784–797.
- [22] J. Becker, P. Bergener, M. Räckers, B. Weiß, A. Winkelmann, 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 (ECIS)*, Pretoria, South Africa, 2010.
- [23] J. Becker, P. Bergener, P. Delfmann, B. Weiß, 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 (ICIS)*, 2011.
- [24] A. Awad, G. Decker, M. Weske, Efficient compliance checking using BPMN-Q and temporal logic, in: M. Dumas, M. Reichert, M.-C. Shan (Eds.), *Business Process Management*, Springer, Berlin, Heidelberg, Milan, Italy, 2008, pp. 326–341.
- [25] C. Ouyang, M. Dumas, A.H.M. ter Hofstede, W.M.P. van der Aalst, Pattern-based translation of BPMN process models to BPEL web services, *Int. J. Web Serv. Res.* 5 (1) (2008) 42–62.
- [26] J. Mendling, H.A. Reijers, W.M.P. van der Aalst, Seven process modeling guidelines (7PMG), *Inf. Softw. Technol.* 52 (2) (2010) 127–136.
- [27] J. Mendling, Detection and prediction of errors in EPC business process models (Doctoral dissertation), WU Vienna University of Economics and Business Administration (2007).
- [28] J. Mendling, H.M.W. Verbeek, B.F. van Dongen, W.M.P. van der Aalst, G. Neumann, Detection and prediction of errors in EPCs of the SAP reference model, *Data Knowl. Eng.* 64 (1) (2008) 312–329.
- [29] B. Kiepuszewski, A. Hofstede, C. Bussler, On structured workflow modelling, in: B. Wangler, L. Bergman (Eds.), *Advanced Information Systems Engineering*, Springer, Berlin, Heidelberg, 2000, pp. 431–445.
- [30] J. Vanhatalo, H. Völzer, F. Leymann, Faster and more focused control-flow analysis for business process models through SESE decomposition, in: B. Krämer, K.-J. Lin, P. Narasimhan (Eds.), *Service-Oriented Computing – IC3OC 2007*, Springer, Berlin, Heidelberg, Vienna, Austria, 2007, pp. 43–55.
- [31] A. Polyvyanyy, L. García-Bañuelos, M. Dumas, Structuring acyclic process models, *Inf. Syst.* 37 (6) (2012) 518–538.
- [32] B.F. van Dongen, R.M. Dijkman, J. Mendling, Measuring similarity between business process models, in: Z. Bellahsene, M. Léonard (Eds.), *Advanced Information Systems Engineering*, Springer, Berlin, Heidelberg, Montpellier, France, 2008, pp. 450–464.
- [33] Z. Yan, R.M. Dijkman, P. Grefen, Fast business process similarity search, *Distrib. Parallel Databases* 30 (2) (2012) 105–144.
- [34] S. Hefner, M. Dittmar, *SAP R/3 Finanzwesen*, Addison-Wesley, Munich, 2001.
- [35] ISO/IEC Standard 14977:1996(E), URL (<http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>) (accessed 3.11.13).
- [36] P. Delfmann, S. Herwig, M. Karow, L. Lis, Ein konfiguratives Metamodellierungswerkzeug, in: P. Loos, M. Nüttgens, K. Turowski, D. Werth (Eds.), *Modellierung betrieblicher Informationssysteme*, GI, Saarbrücken, Germany, 2008, pp. 109–127.
- [37] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, Amsterdam, Netherlands, 1994.
- [38] S. Bräuer, P. Delfmann, H.-A. Dietrich, M. Steinhorst, Using a generic model query approach to allow for process model compliance checking—an algorithmic perspective, in: R. Alt, B. Franczyk (Eds.), *Proceedings of the 11th International Conference on Wirtschaftsinformatik (WI) 2013*, Universität Leipzig, Leipzig, Germany, 2013, pp. 1245–1259.
- [39] J.C. Tiernan, An efficient search algorithm to find the elementary circuits of a graph, *Commun. ACM* 13 (12) (1970) 722–726.
- [40] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros, Workflow patterns, *Distrib. Parallel Databases* 14 (1) (2003) 5–51.
- [41] J. Becker, R. Schütte, *Handelsinformationssysteme*, Redline Wirtschaft, Frankfurt, Germany, 2004.
- [42] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, in: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ACM, New York, NY, USA, 2007, pp. 57–76.
- [43] MongoDB, What is a JSON database? URL (<http://www.mongodb.com/json-and-bso>) (accessed 18.01.13).
- [44] J. Becker, D. Breuker, P. Delfmann, H.-A. Dietrich, M. Steinhorst, Identifying business process activity mappings by optimizing behavioral similarity, in: *Proceedings of the 18th Americas Conference on Information Systems (AMCIS 2012)*, Seattle, WA, USA, 2012.
- [45] J. Cortadella, M. Kishinevsky, L. Lavagno, A. Yakovlev, Deriving Petri nets from finite transition systems, *IEEE Trans. Comput.* 47 (8) (1998) 859–882.
- [46] R. Lu, S. Sadiq, G. Governatori, Compliance aware business process design, in: A. ter Hofstede, B. Benatallah, H.-Y. Paik (Eds.), *Business Process Management Workshops*, Springer, Berlin, Heidelberg, Brisbane, Australia, 2008, pp. 120–131.
- [47] L.T. Ly, S. Rinderle-Ma, K. Göser, P. Dadam, On enabling integrated process compliance with semantic constraints in process management systems, *Inf. Syst. Front.* 14 (2) (2012) 195–219.
- [48] O. Thomas, M. Fellmann, Semantic process modeling – design and implementation of an ontology-based representation of business processes, *Bus. Inf. Syst. Eng.* 1 (6) (2009) 438–451.
- [49] P. Delfmann, S. Herwig, L. Lis, Unified enterprise knowledge representation with conceptual models – capturing corporate language in naming conventions, in: N.J. Jay F., W.L. Currie (Eds.), *Proceedings of the International Conference on Information Systems (ICIS)*, Phoenix, AZ, USA, 2009.
- [50] R. Uba, M. Dumas, L. García-Bañuelos, M. La Rosa, Clone detection in repositories of business process models, in: S. Rinderle-Ma, F. Touman, K. Wolf (Eds.), *Business Process Management*, Springer, Berlin, Heidelberg, Clermont-Ferrand, France, 2011, pp. 248–264.

- [51] C.C. Ekanayake, M. Dumas, L. García-Bañuelos, M. La Rosa, A.H.M. ter Hofstede, Approximate clone detection in repositories of business process models, in: A. Barros, A. Gal, E. Kindler (Eds.), *Business Process Management*, Springer, Berlin, Heidelberg, Tallinn, Estonia, 2012, pp. 302–318.
- [52] X. Yan, J. Han, gSpan: graph-based substructure pattern mining, in: *Proceedings of IEEE International Conference on Data Mining*, IEEE Computer Society, Maebashi City, Japan, 2002, pp. 721–724.
- [53] B. Mahleko, A. Wombacher, Indexing business processes based on annotated finite state automata, in: *Proceedings of 2006 IEEE International Conference on Web Services (ICWS'06)*, IEEE, Washington, DC, USA, 2006, pp. 303–311.
- [54] L. Song, W. Jianmin, A.H.M. ter Hofstede, M. La Rosa, C. Ouyang, L. Wen, A Semantics-based Approach to Querying Process Model Repositories, Technical Report, School for Information Systems of the Queensland University of Technology, 2011.
- [55] T. Jin, J. Wang, N. Wu, M. La Rosa, A.H.M. ter Hofstede, Efficient and accurate retrieval of business process models through indexing, in: R. Meersman, T. Dillon, P. Herrero (Eds.), *On the Move to Meaningful Internet Systems: OTM 2010*, Springer, Berlin, Heidelberg, Herssonis, Crete, Greece, 2010, pp. 402–409.
- [56] J.R. Ullmann, An algorithm for subgraph isomorphism, *J. ACM* 23 (1) (1976) 31–42.
- [57] M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York, W. H. Freeman and Company, 1979.
- [58] S. Sakr, Storing and querying graph data using efficient relational processing techniques, in: J. Yang, A. Ginige, H. Mayr, R.-D. Kutsche (Eds.), *Information Systems: Modeling, Development, and Integration*, Springer, Berlin Heidelberg, Sydney, Australia, 2009, pp. 379–392.
- [59] S. Sakr, A. Awad, A framework for querying graph-based business process models, *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*, ACM Press, New York, USA, 2010, 1297–1300.
- [60] C. Di Francescomarino, P. Tonella, Crosscutting concern documentation by visual query of business processes, in: D. Ardagna, M. Mecella, J. Yang (Eds.), *Business Process Management Workshops*, Springer, Berlin, Heidelberg, Milano, Italy, 2009, pp. 18–31.
- [61] Neo4j, Cypher Query Language. URL (<http://docs.neo4j.org/chunked/stable/cypher-query-lang.html>) (accessed 17.10.13).
- [62] G. Bergmann, Z. Ujhelyi, I. Ráth, D. Varró, A graph query language for EMF models, in: J. Cabot, E. Visser (Eds.), *Theory and Practice of Model Transformations*, Springer, Berlin, Heidelberg, Zurich, Switzerland, 2011, pp. 167–182.
- [63] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, et al., Incremental evaluation of model queries over EMF models, in: D. Petriu, N. Rouquette, Ø. Haugen (Eds.), *Model Driven Engineering Languages and Systems*, Springer, Berlin, Heidelberg, Oslo, Norway, 2010, pp. 76–90.
- [64] I. Choi, K. Kim, M. Jang, An XML-based process repository and process query language for integrated process management, *Knowl. Process Manag.* 14 (4) (2007) 303–316.
- [65] Object Management Group, Object Constraint Language, URL (<http://www.omg.org/spec/OCL/2.3.1/>) (accessed 17.10.13).
- [66] A. Foerster, G. Engels, T. Schattkowsky, R. Van Der Straeten, A. Forster, Verification of business process quality constraints based on visual process patterns, *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, IEEE, Washington, DC, USA, 2007, 197–208.
- [67] World Wide Web Consortium, SPARQL Query Language for RDF, URL (<http://www.w3.org/TR/rdf-sparql-query/>) (accessed 17.10.13).
- [68] S.-M.-R. Beheshti, B. Benatallah, H.R. Motahari-Nezhad, S. Sakr, A query language for analyzing business processes execution, in: S. Rinderle-Ma, F. Toumani, K. Wolf (Eds.), *Business Process Management*, Springer, Berlin, Heidelberg, Clermont-Ferrand, France, 2011, pp. 281–297.
- [69] C.E. Scheidegger, H.T. Vo, D. Koop, J. Freire, C.T. Silva, T. Silva, Querying and re-using workflows with VisTrails, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data – SIGMOD'08*, ACM Press, New York, NY, USA, 2008, 1251–1254.
- [70] H. Störrle, VMQL: a visual language for ad-hoc model querying, *J. Vis. Lang. Comput.* 22 (1) (2011) 3–29.
- [71] H. Störrle, V. Acretoie, Querying business process models with VMQL, *Proceedings of the 5th ACM SIGCHI Annual International Workshop on Behaviour Modelling – Foundations and Applications – BMFA'13*, ACM Press, New York, NY, USA, 2013, 1–10.
- [72] Q. Shao, P. Sun, Y. Chen, WISE: a workflow information search engine, in: *Proceedings of the 25th International Conference on Data Engineering (ICDE'09)*, IEEE, Shanghai, China, 2009, 1491–1494.
- [73] H.A. Reijers, J. Mendling, R.M. Dijkman, Human and automatic modularizations of process models to enhance their comprehension, *Inf. Syst.* 36 (5) (2011) 881–897.
- [74] M. Weidlich, A. Polyvyanyy, N. Desai, J. Mendling, Process compliance measurement based on behavioural profiles, in: B. Pernici (Ed.), *Advanced Information Systems Engineering*, Springer, Berlin, Heidelberg, Hammamet, Tunisia, 2010, pp. 499–514.
- [75] D. Knuplesch, L.T. Ly, S. Rinderle-Ma, H. Pfeifer, P. Dadam, On enabling data-aware compliance checking of business process models, *Conceptual Modeling – ER 2010*, Springer, Berlin, Heidelberg, Vancouver, BC, Canada, 2010, 332–346.
- [76] R. Laue, J. Mendling, Structuredness and its significance for correctness of process models, *Inf. Syst. E-Bus. Manag.* 8 (3) (2010) 287–307.
- [77] B. Weber, M. Reichert, J. Mendling, H.A. Reijers, Refactoring large process model repositories, *Comput. Ind.* 62 (5) (2011) 467–486.
- [78] M. Gupta, R. Rao, A. Pande, A.K. Tripathi, Design pattern mining using state space representation of graph matching, in: N. Meghanathan, B. Kaushik, D. Nagamalai (Eds.), *Advances in Computer Science and Information Technology*, Springer, Berlin, Heidelberg, Bangalore, India, 2011, pp. 318–328.
- [79] J. Dong, Y. Zhao, T. Peng, A review of design pattern mining techniques, *Int. J. Softw. Eng. Knowl. Eng.* 19 (06) (2009) 823–855.
- [80] J. de Lara, H. Vangheluwe, ATOM3: a tool for multi-formalism and meta-modelling, in: R.-D. Kutsche, H. Weber (Eds.), *Fundamental Approaches to Software Engineering*, Springer, Berlin, Heidelberg, 2002, pp. 174–188.
- [81] D. Lucrédio, R.M. Fortes, J. Whittle, MOOGLE: a model search engine, in: K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, M. Völter (Eds.), *Model Driven Engineering Languages and Systems*, Springer, Berlin Heidelberg, 2008, pp. 296–310.
- [82] S. Brinkkemper, Method engineering: engineering of information systems development methods and tools, *Inf. Softw. Technol.* 38 (4) (1996) 275–280.
- [83] D. Eppstein, Subgraph isomorphism in planar graphs and related problems, *J. Graph Algorithms Appl.* 3 (3) (1999) 1–27.
- [84] F. Dorn, Planar subgraph isomorphism revisited, in: *27th International Symposium on Theoretical Aspects of Computer Science (STACS 2010)*, Nancy, France, 2010, pp. 263–274.
- [85] M. Hajiaghayi, N. Nishimura, Subgraph isomorphism, log-bounded fragmentation, and graphs of (locally) bounded treewidth, *J. Comput. Syst. Sci.* 73 (5) (2007) 755–768.
- [86] D. Batra, Conceptual data modeling patterns: representation and validation, *J. Database Manag.* 16 (2) (2005) 84–106.
- [87] K. Duddy, A. Gerber, M. Lawley, K. Raymond, J. Steel, Model transformation: a declarative, reusable pattern approach, in: *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference*, 2003, IEEE Computer Society, 2003, pp. 174–185.