Information Systems 47 (2015) 129–177

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

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.

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).

http://dx.doi.org/10.1016/j.is.2014.06.003
0306-4379/

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