Using a Generic Model Query Approach to Allow for Process Model Compliance Checking – An Algorithmic Perspective

Sebastian Bräuer, Patrick Delfmann, Hanns-Alexander Dietrich, and Matthias Steinhorst

University of Münster, ERCIS, Leonardo Campus 3, 48149 Münster, Germany
{braeuer,delfmann,dietrich,steinhorst}@ercis.uni-muenster.de

Abstract. Increased regulation forces financial companies to assure their business processes' compliance with legal and company-internal rules. In this paper, we introduce a model-driven business process compliance checking approach. It allows for defining compliance rules and identifying their occurrences in process models based on a graph theory-based approach. We outline the challenges to be met in the conceptualization of the approach and especially its implementation through suitable algorithms. Furthermore, we present an according modeling tool and evaluate the approach against related work.

Keywords: Business Process Management, Process Model Compliance Checking, Pattern Matching, GMQL

1 Introduction

In the wake of recent financial and economic crises, financial institutions are faced with a steady increase in regulations [1]. Against this backdrop, compliance checking has become of major interest in both Business Process Management research [2] and practice [3]. Compliance checking means determining if all business processes of a company comply with existing internal and external regulations [4]. It impacts process modeling, because regulations need to be represented in the process models of a company. This is commonly referred to as design time compliance checking [3].

In this context, a regulation can be understood as a restriction on the control flow of a process model. This restriction can be represented as a subsection or pattern of the overall model graph. For example, a simple, yet commonly occurring compliance rule states that a particular activity A must be preceded by an activity B. To apply for a loan, for instance, a customer's financial background first needs to be investigated. For a process model to comply with this rule, every instance of A (apply for loan) must consequently have a predecessor B (investigate financial background). To check whether a given set of process models does indeed comply with this rule, all corresponding pattern instances need to be found. A respective pattern can thereby either represent a compliance violation where A is *not* preceded by B or a compliant model subsection where A is *indeed* preceded by B. Design time compliance checking thus corresponds to pattern matching in process models.

As many companies develop and maintain large repositories of process models [6], a manual pattern search is unfeasible. Implementing an automated solution, however, proves challenging for three reasons:

- First, a company may use different modeling languages to document its business processes. A compliance checking approach therefore should to support different graph-based modeling languages.
- Second, as design time compliance checking equals the problem of graph pattern matching, a compliance checking approach must be able to identify arbitrarily complex structures within the overall model graph. In particular, compliance regulations often translate to (cyclic) paths of elements, which need to be found.
- Third, attribute values of model objects need to be compared to one another in the matching process. An attribute of a model object can for instance be its label or additional information about the object that need to be captured in the model. Consider the example of the 4-eye-principle which dictates that two particularly critical business activities need to be executed by two different employees or organizational units. Fig. 1 contains a violation of this rule in a BPMN-like process model, because activities “Check loan application” (A) and “Verify loan application” (B) are executed by the same organizational unit. In terms of a pattern query, this violation refers to a path from activity A to activity B such that the label attributes of the organizational units directly related to these activities carry the same value.

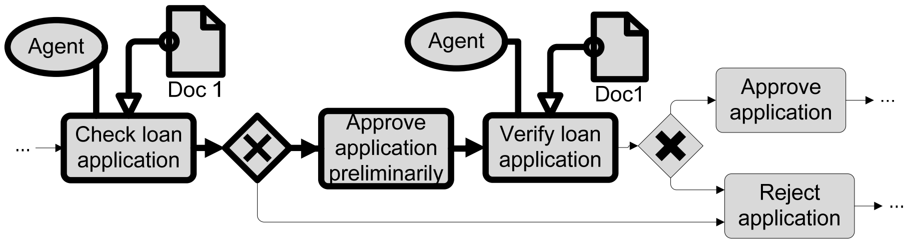


Fig. 1. Violation of the 4-eye-principle

To meet these challenges, a generic model query language (GMQL) has recently been proposed [7] and applied in the context of design-time compliance checking [8]. As our previous work focusses on conceptually specifying GMQL, the purpose of this paper is to provide an algorithmic specification of the matching process. We explain how GMQL is able to interpret variables and variable conditions. The paper also presents a working implementation of the algorithmic specification. As some of the compliance checking approaches have not been implemented, the paper contributes to the proliferation of applicable compliance checking approaches in research and practice.

As the development of a model query language falls into the realm of design science research, the remainder of this paper is structured according to the phases of a design science research process as outlined by PEFFERS ET AL. [9]. To define the objectives of our solution, we briefly introduce the concept of GMQL (Section 2). We design and develop the solution by proposing an algorithmic specification of GMQL in Section 3. This specification is implemented and applied in the context of design time compliance checking in Section 4. This section also provides a discussion of the prerequisites and limitations of GMQL. We evaluate GMQL against the backdrop of existing literature in Section 5. The paper closes with a summary of its main findings and an outlook to future research in Section 6. Figure 2 contains a description of the research process. The figure lists each phase, the research method used to complete the phase, and the section of the paper at hand containing the respective findings.

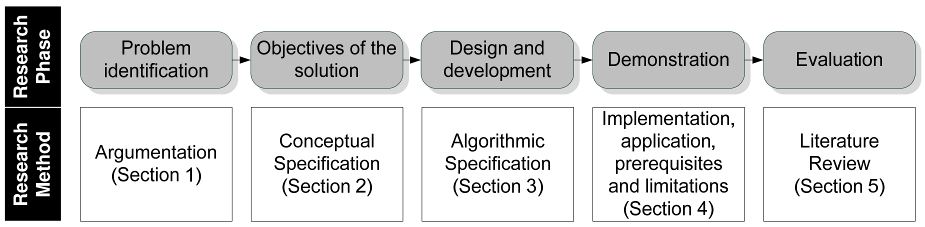

Research Phase
Problem identification
Objectives of the solution
Design and development
Demonstration
Evaluation
Research Method
Argumentation (Section 1)
Conceptual Specification (Section 2)
Algorithmic Specification (Section 3)
Implementation, application, prerequisites and limitations (Section 4)
Literature Review (Section 5)

Fig. 2. Research methodology and structure of paper

2 Objectives of the Solution

GMQL recognizes any conceptual model as two basic sets. These are the set O of its objects and the set R of its relationships. The set E of elements is defined as the union of O and R. GMQL provides set-modifying functions and operators that perform operations on these basic sets. To be able to specify pattern queries, we define four classes of such functions. First, we have to be able to recognize elements of a particular type of the language and elements having attributes (of a particular type or value):

- *ElementsOfType(X,a)* returns all elements of the input set $X \in E$ that belong to a particular type a. The respective elements are put into a single output set.
- *ElementsWithTypeAttributeOfValue(X,a,b)* returns a set containing all elements of the input set of elements $X \in E$ that have an associated type attribute of name a with the attribute value b. A type attribute is assigned to all instances of a given object type. An example of a type attribute is the label of a BPMN activity. As all activities have a label the according attribute is assigned to the type specification.
- *InstanceAttributesOfValue(Y,a,b)* returns a set containing all instance attributes of the input object set $Y \in O$ whose attribute is of name a and whose values are equal to b. An instance attribute is assigned to an object instances. An example of an instance attribute is a clause of a particular law that needs to be adhered to in order to compliantly execute a process activity.
- *ElementsWithTypeAttributeOfDataType(X,a)* returns a set containing all elements of the set of elements $X \in E$ that are assigned type attributes of a particular data type. The data type of a type attribute can be STRING, INT, DOUBLE, BOOL, and ENUM. In case the second parameter is set to INT the functions will therefore return all elements that have type attributes of type INT. In case of a process activ-

ity, respective type attributes can for instance be execution time, costs, or number of employees involved to execute the activity.

- *InstanceAttributesOfDataType(Y,a)* returns a set containing all instance attributes of the object set Y ∊ O whose data types equal the parameter a. Again, the above-mentioned data types are supported.

Furthermore, to examine neighborhood relationships, we need to identify all elements having (a particular number of) ingoing or outgoing relations (of a particular type):

- *ElementsWith{In|Out}Relations(X,Z)* return all elements of X ∊ E and their {ingoing | outgoing} relationships defined in Z ∊ R. These functions each return a set of sets. Each inner set contains an element of X and all its relationships of Z.
- *ElementsWith{In|Out}RelationsOfType(X,Z,c)* return all elements of X ∊ E and their {ingoing | outgoing} relationships of Z ∊ R that are of type c. Again, these functions return a set of sets with each inner set containing one element of X as well as its {ingoing | outgoing} relationships of Z that belong to type c.
- *ElementsWithNumberOf{In|Out}Relations(X,Z,n)* return all elements of X ∊ E that have a predefined number n of relationships of Z ∊ R. These functions return a set of sets with each inner set containing one element and its n relationships.
- *ElementsWithNumberOf{In|Out}RelationsOfType(X,Z,n,c)* are a combination of the two latter groups of functions. They return elements having a predefined number n of relationships of Z ∊ R that are of type c. These functions return a set of sets.

In addition, we want to be able to find particular elements, their immediate neighbors, and the relationship(s) between them:

- *ElementsDirectlyRelated(X_1,X_2)* and *DirectSuccessors (X_1,X_2)* return all elements of X_1 ∊ E, their neighboring elements of X_2 ∊ E, as well as the relationships between the respective elements. ElementsDirectlyRelated (X_1,X_2) only works on undirected graph sections whereas DirectSuccessors (X_1,X_2) only works on directed graph sections.

Lastly, to be able to find structures representing element paths of arbitrary length, we included the following functions in the pattern matching mechanism:

- *{Directed}Paths(X_1,X_n)* return all {directed} paths between all elements of X_1 ∊ E and all elements of X_n ∊ E. One inner set of the resulting set of sets contains one path from one element of X_1 to one element of X_n.
- *{Directed}Paths{Not}ContainingElements(X_1,X_n,X_c)* return all paths from elements of X_1 ∊ E to all elements of X_n ∊ E that contain at least one or no element of X_c.

For all paths-functions GMQL offers versions that determine only the *shortest* or *longest paths* as well as *loops*. As its theoretical basis is set theory, GMQL furthermore incorporates the set operators *Union*, *Intersect*, and *Complement* that perform the standard set operations on two sets of elements. Analogously to the *Intersect*- and *Complement*-Operators, the approach offers versions working on sets of sets (*Inner-Intersect* and *InnerComplement*). The *Join*-operator unifies two inner input sets if they have at least one element in common. The *SelfUnion* operator turns a set of sets into a single set, whereas a *SelfIntersect* operator performs an intersection on all inner sets resulting in one single set that holds all elements contained in every inner set.

These set-modifying functions and operators allow for constructing arbitrary pattern queries recursively, as the result set of one particular function/operator call serves as input for another function/operator (cf. Section 4.1).

In the context of design time compliance checking not only the graph structure needs to be analyzed but also information that is stored in object variables. Consider, again, the 4-eye-principle illustrated in Fig. 1. This rule requires the query language to treat values of particular attributes (in this case the labels of two organizational units) as variables that are to be compared to one another. Variables act as wildcards in the matching process and are of a predefined type. Available variable types are *ElementType*, *RelationshipType*, *Integer*, *AttributeDataType*, *AttributeName*, and *AttributeValue*. This implies that all functions that take instances of these data types as input can also take a variable of the respective type as input. The *ElementsOfType* function, for instance, is called with a set of elements and an element type specification. Instead of this type specification the function alternatively can be fed a variable of type *ElementType*. To define variable conditions only those variables can be used that have the same type. In other words, only variables of the same type can be compared to one another. Possible condition types are equal (=), unequal ($\neq$), smaller-than (<), greater-than (>), smaller-than-or-equal ($\leq$), and greater-than-or-equal ($\geq$).

3 Design and Development

A pattern query is represented in a tree structure. The matching algorithm is implemented using the visitor design pattern known from software engineering [10]. The visitor walks through the query expression in a bottom-up fashion calculating the leaf nodes first. In the following, this tree structure and the visitor calculating the node results will be referred to as the standard pattern compiler.

If a pattern query is run on a model, the matching algorithm distinguishes three cases. First, the query does not contain any variables or conditions. Second, the query contains variables but no conditions. Third, the query contains both variables and conditions. In case the query does not contain either variables or conditions, the standard pattern compiler is executed. If, however, the query contains either a variable and/or a condition a pre-compiler is executed that replaces the variable with the concrete values that can be found in the model the query is executed on. Each variable instantiation is then used to create a standard pattern query (without variables or conditions) that is fed to the standard pattern compiler. As this increases runtimes by the number of created pattern queries, a caching mechanism was implemented that allows for caching previously calculated (fragments of) pattern queries. The caching mechanism is implemented as a hash table allowing access to intermediary matching results in constant time.

For evaluating an arbitrary query q against a model m the function *CompilePatternQuery* (cf. Figure 3) can be used. As an output a set of sets S_r is calculated that contains all pattern matches. As a first step, the query is analyzed and all occurring variables are stored in a set S_v. A variable thereby at least consists of a type and a label. In case that no variables are discovered within the query, no pre-compiling is necessary and the query is included in the set S_q that is directly handed over to the actual compiler for evaluating the query against the model.

```
PROCEDURE CompilePatternQuery(q, m)
    INPUT: Query q with potential variable expression and op-
     tional condition(s); model m to be analysed
    OUTPUT: Set $S_r$ that contains all pattern matches

    Create a set $S_v$ containing all variables in q
    IF $S_v$ is empty THEN
        Add q to the set $S_q$ containing all queries
    ELSE IF q contains at least one condition THEN
        Create set $S_c$ holding all conditions of q
        $S_q$ ← PrecompilePatternQueryWithConditions(q, m, $S_v$,
                $S_c$)
    ELSE q contains no conditions
        $S_q$ ← PrecompilePatternQueryWithVariables(q, m, $S_v$)
    END IF
    Create $S_r$ by executing all patterns in $S_q$ on model m
END PROCEDURE CompilePatternQuery
```

Fig. 3. Pseudo-code of the function CompilePatternQuery

In case that the query includes at least one variable ($S_v \neq \emptyset$) the compiler analyses whether the query also includes any conditions. If this is the case, the set S_c is created, which contains all conditions that are included in *q*. A condition thereby consists of a left and right value, as well as an operator that is used for comparing the included values. With the query *q*, the model *m*, the set of variables S_v and the set of conditions S_c as parameters, the function *PrecompilePatternQueryWithConditions* is called (cf. Figure 4). The goal of this call is the creation of one or several queries that do not contain any variables or conditions but only those concrete value occurrences for which the conditions are fulfilled. In analogy to the conventional approach, these queries are then finally handed over to the standard pattern compiler as the set of sets S_q.

For generating the set of queries the algorithm requires the creation of an auxiliary empty set S_a. After that we use a loop over the set of variables S_v to determine the set of value occurrence S_{vo} per variable s_v and store S_{vo} in S_a. These execution steps are exemplarily depicted in Figure 5. a-c for variables of type Integer. The potential value occurrences for a variable s_v are identified by searching for all possible value occurrences within the whole model. In case of a variable of type Integer the algorithm for instance collects the number of incoming and outgoing relationships per object. For determining possible value occurrences of a variable of type *AttributeValue* the attribute values of instance attributes and type attributes are collected. For the other attributes the identification of value occurrences is performed accordingly.

```
PROCEDURE PrecompilePatternQueryWithConditions (q, m, $S_v$, $S_c$)
    INPUT: Query q with variable expression and condition(s);
      model m to be analysed; set $S_v$ containing all variables;
      set $S_c$ containing all conditions
    OUTPUT: Set of precompiled pattern queries $S_q$
    Create empty auxiliary set $S_a$
    FOREACH $s_v$ in $S_v$
        Get set of value occurrences $S_{vo}$ for $s_v$ in model m
        Add $S_{vo}$ to $S_a$
    END FOREACH
    Create a set of sets $S_{qvc}$ that contains the Cartesian
      products of all $S_{vo}$ in $S_a$ to obtain all possible value
      combinations
    FOREACH $s_{qvc}$ in $S_{qvc}$
        IF all conditions in $S_c$ hold true on $s_{qvc}$ THEN
            Add set of value occurrences $s_{qvc}$ to a set of
              verified value combinations $S_{vvc}$
        END IF
    END FOREACH
    FOREACH $s_{vvc}$ in $S_{vvc}$
        Create a clone q' of the original query
        Replace variables $s_v$ in q' with identified valid
          value occurrences from $s_{vvc}$
        Add q' to set of precompiled pattern queries $S_q$
    END FOREACH
END PROCEDURE PrecompilePatternQueryWithConditions
```

Fig. 4. Pseudo-code of the function PrecompilePatternQueryWithConditions

On the basis of the determined value occurrences that are stored per variable in S_a, the Cartesian product of all sets of value occurrences S_{vo} in the auxiliary set S_a is generated for obtaining sets with all possible value combinations (cf. Figure 5 d). These sets are stored in the set S_{qvc}. For determining those value combinations that fulfill the associated conditions, a loop over the set S_{qvc} is used (cf. Figure 5 e). Thereby, a set of value occurrences s_{qvc} is only written to the set of verified value combinations S_{vvc} if all conditions are fulfilled by the according values (cf. Figure 5 f). Finally, for each of these verified value combinations s_{vvc} a copy *q'* of the original query *q* is generated in which all variables s_v are replaced by their concrete value occurrences. This clone *q'* is then added to the set of precompiled queries S_q that is returned by the function.

a) Derived sets:

$S_v = \{A, B, C\}$

$S_c = \{\{A \neq B\}, \{C = 4\}\}$

b) Value occurrences for each variable:

$S_{vo} = \{1, 2, 3\} \mid s_v = A$
$S_{vo} = \{1\} \mid s_v = B$
$S_{vo} = \{4, 5\} \mid s_v = C$

c) Auxilary set of value occurrences:

$S_a = \{\{1, 2, 3\}, \{1\}, \{4, 5\}\}$

d) Creation of the Cartesian product:

$S_{qvc} = \{\{1, 1, 4\}, \{1, 1, 5\}, \{2, 1, 4\}, \{2, 1, 5\}, \{3, 1, 4\}, \{3, 1, 5\}\}$

e) Checking of conditions:

$A \neq B \wedge C = 4$

$1 \neq 1 \wedge 4 = 4$ ×
$1 \neq 1 \wedge 5 = 4$ ×
$2 \neq 1 \wedge 4 = 4$ ✓
$1 \neq 1 \wedge 5 = 4$ ×
$3 \neq 1 \wedge 4 = 4$ ✓
$3 \neq 1 \wedge 5 = 4$ ×

f) Set of validated value occurrences:

$S_{vvc} = \{\{2, 1, 4\}, \{3, 1, 4\}\}$

Fig. 5. Exemplary execution steps of the function PrecompilePatternQueryWithConditions

In all other cases a query is present that includes variables ($S_v \neq \emptyset$), but that does not have any associated conditions ($S_c \neq \emptyset$). Again, a pre-compiling of this query is required. In this case, we execute the function *PrecompilePatternQueryWithVariables* with the query *q*, the model *m*, and the set of variables S_v as input (cf. Figure 6).

```
PROCEDURE PrecompilePatternQueryWithVariables(q, m, $S_v$)
    INPUT: Query q with variable expression; model m to be
      analysed; Set $S_v$ containing all variables
    OUTPUT: Set of precompiled pattern queries $S_q$
    Create temporary set of possible patterns $S_q$ and add q to
    it
    Create an additional empty temporary set of possible que-
    ries $S_q'$
    FOREACH $s_v$ in $S_v$
        Get set of value occurrences $S_{vo}$ for variable $s_v$ in
        model m
        FOREACH $s_{vo}$ in $S_{vo}$
            FOREACH $s_q$ in $S_q$
                Replace $s_v$ in $s_p$ by $s_{vo}$ and add $s_q$ to $S_q'$
            END FOREACH
        END FOREACH
    $S_q$ = $S_q'$
    Clear $S_q'$
    END FOREACH
END PROCEDURE PrecompilePatternQueryWithVariables
```

Fig. 6. Pseudo-code of the function PrecompilePatternQueryWithVariables

As described above, a set of queries S_q is finally returned that includes all adjusted queries. For generating this set the algorithm first requires the creation of two temporary sets S_q and S_q' that are used for handling the original query *q* as well as the cleaned instances of this query where some or all of the variables have been replaced by concrete values. After that, we use a loop over the set of variables S_v to determine the set of value occurrences S_{vo} analogously to the approach with conditions. Two embedded loops (the outer one over the set of value occurrences S_{vo}, the inner one over the temporary set of possible patterns S_q) are employed for replacing the variables within the queries of the set S_q. Thereby, for each inserted value occurrence an altered query is added to the other temporary set of possible queries S_q'. Inside the outermost loop the set S_q is assigned to the set S_q' that now contains the partly replaced queries (if it is the ith iteration, the variable s_{vi} is replaced). The temporary set S_q' is then cleared and the loop starts over. Finally, the set S_q contains all pre-compiled patterns that do not include any more variables. This set is then fed to the standard pattern compiler.

4 Demonstration

4.1 Implementation and Application

The pattern query given below represents the violation of the 4-eye-principle outlined in the introductory section:

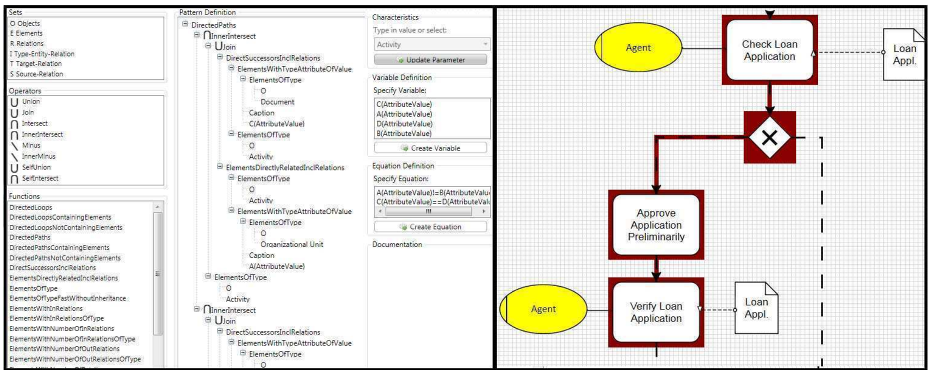
```
DirectedPaths(
  InnerIntersect(Join(
        DirectSuccessorsInclRelations(
           ElementsWithTypeAttributeOfValue(
              ElementsOfType(O, Document), Caption, A)
           ElementsOfType(O, Activity))
        ElementsDirecltyRelatedInclRelations(
           ElementsWithTypeAttributeOfValue(
              ElementsOfType(O, OrgaUnit), Caption, C)
           ElementsOfType(O, Activity)))
     ElementsOfType(O, Activity))
  InnerIntersect(Join(
        DirectSuccessorsInclRelations(
           ElementsWithTypeAttributeOfValue(
              ElementsOfType(O, Document), Caption, B)
           ElementsOfType(O, Activity))
        ElementsDirecltyRelatedInclRelations(
           ElementsWithTypeAttributeOfValue(
              ElementsOfType(O, OrgaUnit), Caption, D)
           ElementsOfType(O, Activity)))
     ElementsOfType(O, Activity)))

A=B;C=D
```

The query complies with the BPMN-like process modeling language depicted in Figure 1. It returns process paths of arbitrary length that start and end in an activity that is directly connected to both an organizational unit as well as a document. In addition to the label values of the organizational units being equal, the query further restricts the result set to include only those paths whose start and end activities are directly related to document types that also have equal labels.

Both parameters of the *DirectedPaths* call are structured analogously. The join-construct returns all activities, their adjacent organizational units and documents, as well as the relationships between these objects (note that the *DirectSuccessors* call is necessary, because the relationship between an activity and a document is directed (cf. Figure 1)). This construct is then inner-intersected with the set of all activities to only feed activities to the *DirectedPaths* call. The set of organizational units and the set of documents that appear in both input parameters of the *DirectedPaths* call are further restricted to include only those objects that have captions of a particular value. These values take variables A to D as input that form two conditions specifying that all corresponding attributes need to have the same value.

This query is only an example of how GMQL can be used in the domain of design time compliance checking. Further examples are introduced in the following section as well as in [8]. In order to specify a query for a different modeling language the type information only needs to be adapted to that language (e.g. in an EPC an activity is called a function). GMQL is thus generic in the sense that it can be used for models created in arbitrary graph-based modeling languages.



```
DirectedPaths
  ∩InnerIntersect
    ∪Join
      DirectSuccessorsInclRelations
        ElementsWithTypeAttributeOfValue
          ElementsOfType
            O
            Document
          Caption
          C(AttributeValue)
        ElementsOfType
          O
          Activity
      ElementsDirectlyRelatedInclRelations
        ElementsOfType
          O
          Activity
        ElementsWithTypeAttributeOfValue
          ElementsOfType
            O
            Organizational Unit
          Caption
          A(AttributeValue)
    ElementsOfType
      O
      Activity
  ∩InnerIntersect
    ∪Join
      DirectSuccessorsInclRelations
        ElementsWithTypeAttributeOfValue
          ElementsOfType
            O
```

```
C(AttributeValue)
A(AttributeValue)
D(AttributeValue)
B(AttributeValue)
```

```
A(AttributeValue)!=B(AttributeValu
C(AttributeValue)==D(AttributeVal
```

Fig. 7. Query specification environment (left) and model editor (right)

GMQL was implemented as a plugin for an existing meta-modeling tool that was available from a prior research project. Figure 7 depicts a screenshot of the meta-modeling tool's language editor that contains a query specification environment (left-hand part of Figure 7). The various functions and operators are available on the left-hand side. Variables and conditions can be defined on the right-hand side. These elements can be dragged and dropped to the pattern definition field where the query can be constructed. The right-hand part of Figure 7 contains an excerpt of the tool's modeling environment, which allows for selecting and running a previously defined pattern query on a model. All identified pattern occurrences are highlighted. Note that in this case a path is found that starts and end in an activity. Start and end objects are both directly related to an organizational unit and a document that have identical labels.

4.2 Prerequisites and Limitations

GMQL is able to identify structural patterns in process models of arbitrary modeling languages. However, design time compliance checking cannot solely be based on a structural analysis of the model graph. In addition, the semantics of element labels need to be considered as well. As indicated above, GMQL considers a label to be an attribute of an element. Respective information can thus be included in the matching process. Particular semantic issues like synonyms or homonyms, however, are not directly addressed by GMQL. However, GMQL allows for integrating approaches supporting terminological unambiguousness easily. Parts of the underlying algorithms comparing attribute labels are built generically. This means that, for example, it can be determined whether two nodes carry labels with the same *meaning* by checking their underlying ontological or linguistic concepts (e.g., "check invoice" ~ "bill audit"). Corresponding approaches maintaining the meaning of element labels can be found in [11] or [12] and can be used together with GMQL.

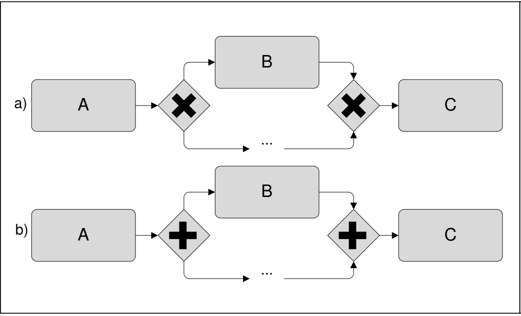
An additional prerequisite of using GMQL is an appropriate level of detail for a given model. In particular, the model needs to be syntactically correct and terminologically unambiguous. Element paths, for instance, can only be found if the model graph is not fragmented. The application of process modeling guidelines as proposed by [13] is thus advisable.

To apply GMQL in an enterprise, process managers have to learn GMQL in order to define the queries representing the appropriate compliance rules. Alternatively, an enterprise-wide role can be set up that is responsible for defining queries. In any way, we expect that this will require a considerable learning effort.

In terms of limitations, GMQL is limited to design time compliance checking in conceptual process models. In this context design time refers to an analysis of the model graph structure. Therefore, compliance violations that do not translate to specific model subsections cannot be found by our approach. A common compliance rule, for instance, states that particular business documents need to be stored for a given period of time. Such a rule cannot be checked in the model graph structure and can thus not be found by GMQL. In addition, compliance violations that occur during process runtime cannot be detected by GMQL. To determine if a given process instance contains a compliance violation respective mining techniques need to be applied that are beyond the scope of this paper. Such approaches are discussed in the area of business process intelligence [14].

Furthermore, GMQL does not consider the execution semantics of a process model. It analyses the graph structure and includes type and label information in its matching process. GMQL does not recognize the semantics of a given element type (e.g. an XOR split carries a different meaning than an AND join). In our experience, this, however, is not necessary in many cases. Consider, for example, the predecessor/successor compliance rule as depicted in Figure 8. Let the rule require that activity B needs to be performed after activity A has been executed. In case a) the path from A to C represents a violation of this rule, while in case b) it does not (because of the AND split/join). The GMQL query given in the right-hand part of Figure 8 identifies corresponding structures. It determines directed paths from A to C that do not contain

B. From the resulting set of sets, all paths that contain AND split nodes are subtracted. An AND split is a node having at least two outgoing edges. Thus, the set of all ANDs that have one outgoing edge is subtracted from the set of all ANDs resulting in the set of all ANDs that have at least two outgoing edges (i.e. AND splits). In case a), the query thus identifies the path from A to C as a compliance violation; while in case b) the empty set is returned (i.e. no violation is found). This argument demonstrates that including execution semantics in the matching process is not always necessary and analyzing the graph structure suffices to detect compliance violations.



```
Complement(
 DirectedPathsNotContainingElements(
  ElementsWithTypeAttributeOfValue(
   ElementsOfType(O, Activity), label, A),
  ElementsWithTypeAttributeOfValue(
   ElementsOfType(O, Activity), label, C),
  ElementsWithTypeAttributeOfValue (
   ElementsOfType(O, Activity), label, B)),
 DirectedPathsContainingElements(
  ElementsWithTypeAttributeOfValue(
   ElementsOfType(O, Activity), label, A),
  ElementsWithTypeAttributeOfValue (
   ElementsOfType(O, Activity), label, C),
  Complement(
   ElementsOfType(O, AND),
    ElementsWithNumberOfOutRelations(
     ElementsOfType(O,AND),1))))
```

Fig. 8. Predecessor/successor compliance rule (left) and according GMQL query (right)

5 Evaluation

Design-time compliance checking requires an approach that is able to (a) identify arbitrarily complex structures within a model graph, (b) define pattern queries containing variables and conditions, and (c) support multiple modeling languages.

Many approaches discussed in the literature transform a process model into some form of finite-state automaton. This aims at checking the temporal logic of the task execution. The approach of GOVERNATORI ET AL. and the related research approaches [15-18] fall into this category. As they transform the models to a state-based event system, it is possible to map control flow restrictions to these states. This transformation, however, does not allow for finding more complex structures like highly-branched structures. This drawback is overcome in [19], as the transformation is a preprocessing step to an SQL-based query approach that is able to find more complex structures. Thus, element paths can be calculated. However, SQL suffers from performance problems when calculating paths, because node and edge tables need to be joined a potentially huge number of times [20]. In contrast, GMQL allows for calculating element paths without complex join operations. Furthermore, as GMQL provides specialized paths functions determining paths that (do not) contain particular elements a transformation to finite-state automata can be avoided which further increases runtime performance (see example above for more details). In addition, the approaches introduced above allow for defining compliance violations as formulae that are fed to the state automaton. These formulae require extensive knowledge about the used symbols and operators. GMQL queries in contrast are, albeit complex at times, comparatively easy to understand, because the names of the functions and operators are self-explanatory.

Other approaches are restricted to particular modeling languages or language types [21-24]. The work of MONAKOVA ET AL. [25], for instance, transforms BPEL models to logical representations to allow for control- and data-flow analysis. The logical representation of a BPEL model stores the execution path and variables that allow for verifying compliance rules. WOLTER ET AL. [26-27] allow conditions to support different access control and authorization strategies in extended BPMN models. It is furthermore not possible to define arbitrary pattern queries. Moreover, OCL can be used to define and check compliance violations [28]. However, OCL is also restricted to UML. In contrast, GMQL can be used on models created in arbitrary graph-based modeling languages, because it is based on the idea that any model can be represented as the set of its objects and relationships.

Therefore, GMQL is similar to many approaches put forth in the area of general pattern matching in (process model) graphs. In graph theory, the problem of pattern matching is known as the problem of subgraph isomorphism (SGI). A plethora of algorithms has been developed in recent years that perform well in practical runtime scenarios despite the theoretical intractability of this problem (e.g., [29]). SGI, however, is concerned with finding one-to-one mappings between a given pattern graph and a subsection of a model graph. This requires knowing the exact structure of the pattern before searching it, which is unrealistic in many compliance checking scenarios, because, for instance, the length of a paths between particular elements is not always a priori known (see 4-eyes-principle above). The problem of finding similar patterns in a graph is known as subgraph homeomorphism (SGH). This problem, however, covers aspects of minor containment leading to an abundance of possible (mostly not suitable) pattern occurrences. These go too far for the purpose of pattern matching in process models [30]. Also, the theoretical runtime complexity of this problem is even worse than that of SGI [30]. Neither SGI nor SGH are able to define restrictions in the form of variables and conditions as it is possible with GMQL.

This literature analysis demonstrates that GMQL advances the current state of the art in design time compliance checking, because it is able to find arbitrarily complex patterns in models of any graph-based modeling languages. At the same time it allows for defining restrictions on attributes of particular model elements while avoiding computationally expensive model transformations to finite-state automata.

6 Summary and Outlook

We presented an algorithmic specification of a generic model query language. Besides the graph structure of a model, the language compiler is able to process attributes, variables, and variable conditions. Future research will continue along multiple consecutive lines. We plan to extend GMQL to allow for searching pattern occurrences that stretch across multiple models. This implies that GMQL must also allow for defining pattern queries that apply to models of more than one language. So far, a pattern query can only be executed on a model of the language the query was created for. Relaxing this restriction will allow for verifying, if a hierarchy structure captured in an organizational chart is properly represented in a process model. Both extensions can be easily implemented by augmenting the set of elements that is initially fed into matching algorithm. We aim at gaining a deeper understanding of the compliance checking domain in order to determine if GMQL is indeed sufficient for this field. This may require including additional information in the matching process which so far has not been necessary (e.g. execution semantics). We will explore the possibility of compliance checking during modeling by enriching a process modeling language with GMQL-based concepts supporting the ex-ante specification of compliance rules.

References

1. Opromolla, G.: Facing the Financial Crisis: Bank of Italy's Implementing Regulation on Hedge Funds. Journal of Investment Compliance 10 (2), 41-44 (2009)
2. Abdullah, S.N., Indulska, M., Sadiq, S.: A Study of Compliance Management in Information Systems Research. In: Proceedings of the 17th European Conference on Information Systems, pp. 1-10 (2009)
3. Sadiq, S., Governatori, G., Namiri, K.: Modeling Control Objectives for Business Process Compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.): BPM 2007. LNCS, Vol. 4714, pp. 149-164. Springer, Heidelberg (2007)
4. El Kharbili, M., De Medeiros, A.K.A, Stein, S., van der Aalst, W.M.P.: Business process compliance checking: Current state and future challenges. In: MoBIS 2008. LNI, Vol. 141, pp. 107-113. GI, Bonn (2008)
5. Cabanillas, C., Resinas, M., Ruiz-Cortés, A.: Hints on how to face business process compliance. Ac-tas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos 4 (4), 26-32 (2010)
6. Uba, R., Dumas, M., García-Bañuelos, L., La Rosa, M.: Clone Detection in Repositories of Business Process Models. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.): Business Process Management. LNCS, Vol. 6896, pp. 248–264. Springer, Heidelberg (2011)
7. Delfmann, P., Herwig, S., Lis, L., Stein, A., Tent, K., Becker, J.: Pattern Specification and Matching in Conceptual Models. A Generic Approach Based on Set Operations. Enterprise Modelling and Information Systems Architectures 5 (3), 24-43 (2010)
8. Becker, J., Bergener, P., Delfmann, P., Weiß, B.: Modeling and Checking Business Process Compliance Rules in the Financial Sector. In: Proceedings of the 32nd International Conference on Information Systems (2011)
9. Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A design science research methodology for information systems research. Journal of Management Information Systems 24 (3), 45-77 (2007)
10. Gamma, E., Helm, R., Johnson, R. E.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Longman, Amsterdam (1995)
11. Thomas, O., Fellmann, M.: Semantic Process Modeling – Design and Implementation of an Ontology-based Representation of Business Processes. Business and Information Systems Engineering 1 (6), 438-451 (2009)
12. Delfmann, P., Herwig, S., Lis, L.: Unified Enterprise Knowledge Representation with Conceptual Models - Capturing Corporate Language in Naming Conventions. In: Proceedings of the 30th International Conference on Information Systems (2009)

13. Mendling, J., Reijers, H., van der Aalst, W.: Seven Process Modeling Guidelines. Information and Software Technology 52 (2), 127-136 (2010)
14. Jiafei, L., Jagadeesh, C., van der Aalst, W.: Mining Context-Dependent and Interactive Business Process Maps using Execution Patterns. In: Proceedings of the 6th International Workshop on Business Process Intelligence (2010)
15. Governatori, G., Milosevic, Z.: A formal analysis of a business contract language. International Journal of Cooperative Information Systems 15 (4), 659-685 (2006)
16. Hoffmann, J., Weber, I., Governatori, G.: On compliance checking for clausal constraints in annotated process models. Information Systems Frontiers 43, 1-23 (2009)
17. Lu, R., Sadiq, S., Governatori, G.: Compliance aware business process design. In: ter Hofstede, A., Benatallah, B., Paik, H.-Y. (eds.): Business Process Management Workshops. LNCS, Vol. 4928, pp. 120-131 , Springer, Heidelberg (2008)
18. Lu, R., Sadiq, S., Governatori, G.: Measurement of Compliance Distance in Business Processes. Information Systems Management 25 (4), 344-355 (2008)
19. Awad, A., Weske M.: Visualization of Compliance Violation in Business Process Models. In: Awad, A., Weske, M. (eds.): BPM 2009. LNBIP 43, pp. 182-193. Springer (2010)
20. Sakr, S.: Storing and Querying Graph Data Using Efficient Relational Processing Techniques. In: Proceedings of the 3rd International United Information Systems Conference (2009)
21. Liu, Y., Muller, S., Xu, K.: A static compliance-checking framework for business process models. IBM Systems Journal 46 (2), 335-361 (2007)
22. Knuplesch, D., Ly, L., Rinderle-Ma, S., Pfeifer, H., Dadam, P.: On Enabling Data-Aware Compliance Checking of Business Process Models. In: Conceptual Modeling – ER 2010. LNCS, Vol. 6412, pp. 332-346. Springer, Heidelberg (2010)
23. Trčka, N., van der Aalst, W., Sidorova, N.: Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. In: Advanced Information Systems Engineering. LNCS, Vol. 5565, pp. 425-439. Springer, Heidelberg (2009)
24. Kumar, A., Liu, R.: A Rule-Based Framework Using Role Patterns for Business Process Compliance. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.): Rule Representation, Interchange and Reasoning on the Web. LNCS, Vol. 5321, pp. 58-72, Springer, Heidelberg (2008)
25. Monakova, G., Kopp, O., Leymann, F., Moser, S., Schäfers, K.: Verifying Business Rules Using an SMT Solver for BPEL Processes. In: Proceedings of the Business Process and Services Computing Conference, pp. 81-94 (2009)
26. Wolter, C., Miseldine, P., Meinel, C.: Verification of business process entailment constraints using SPIN. In: Massacci, F., Redwine, S.T., Zannone, N. (eds.): Engineering Secure Software and Systems, pp. 1-15 (2009)
27. Wolter, C., Meinel, C.: An approach to capture authorisation requirements in business processes. Requirements Engineering 15 (4), 359-373 (2010)
28. Warmer, J., Kleppe, A.: Object Constraint Language 2.0. mitp, Bonn 2004
29. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE transactions on pattern analysis and machine intelligence 26 (10), 1367-1372 (2004)
30. Lingas, A., Wahlen, M.: An exact algorithm for subgraph homeomorphism, Journal of Discrete Algorithms 7 (4), 464-468 (2009)