# Detecting Quality Problems in Research Data: A Model-Driven Approach

Arno Kesper
Philipps-Universität Marburg
arno.kesper@uni-marburg.de

Viola Wenz
Philipps-Universität Marburg
viola.wenz@uni-marburg.de

Gabriele Taentzer
Philipps-Universität Marburg
taentzer@uni-marburg.de

## ABSTRACT

As scientific progress highly depends on the quality of research data, there are strict requirements for data quality coming from the scientific community. A major challenge in data quality assurance is to localise quality problems that are inherent to data. Due to the dynamic digitalisation in specific scientific fields, especially the humanities, different database technologies and data formats may be used in rather short terms to gain experiences. We present a model-driven approach to analyse the quality of research data. It allows abstracting from the underlying database technology. Based on the observation that many quality problems show anti-patterns, a data engineer formulates analysis patterns that are generic concerning the database format and technology. A domain expert chooses a pattern that has been adapted to a specific database technology and concretises it for a domain-specific database format. The resulting concrete patterns are used by data analysts to locate quality problems in their databases. As proof of concept, we implemented tool support that realises this approach for XML databases. We evaluated our approach concerning expressiveness and performance in the domain of cultural heritage based on a qualitative study on quality problems occurring in cultural heritage data.

## CCS CONCEPTS

• **Information systems** → **Expert search**; **Query languages**; • **Software and its engineering** → **Domain specific languages**.

## KEYWORDS

Data quality, Model-driven development, Pattern matching

**ACM Reference Format:**
Arno Kesper, Viola Wenz, and Gabriele Taentzer. 2020. Detecting Quality Problems in Research Data: A Model-Driven Approach. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Montreal, QC, Canada.* ACM, New York, NY, USA, 28 pages. https://doi.org/10.1145/3365438.3410987

## 1 INTRODUCTION

Research data is defined as data that is "generated in the course of scientific work" [21]. The quality of research data is essential for scientific progress. Establishing and maintaining data quality throughout the data life cycle is still a challenge [21].

*MODELS '20, October 18–23, 2020, Montreal, QC, Canada*


We present an approach to analyse the quality of research data. For the requirements elicitation and evaluation we chose the domain of cultural heritage research. However, we do not see any obstacles to applying our approach to other kinds of research data since the approach itself is not dependent on specific characteristics of the chosen domain.

As common in various research fields, there is a high amount of uncertainty in cultural heritage data, such as data on buildings or artworks. It can occur in various forms and often remains implicit. Data fields may remain empty, for example, since the requested information is unknown. Or information that should be unique (e.g. a year of birth) may not as there exist several scientific opinions about that. Hence, it is very natural for a scientific discourse that there are quality problems in curated databases. As there is no standard definition for *data quality* in a scientific context, we start with a literature survey to investigate multiple quality dimensions, such as consistency, completeness and precision [3, 29, 42]. Furthermore, we elicited data *quality problems* as they occur in curated cultural heritage databases. Identified quality problems, such as imprecise, redundant and semantically incorrect data, are mapped to affected quality dimensions.

A major step to data quality assurance is to analyse the inherent quality problems. We can observe that the digitalisation of the humanities has started but this process is not settled yet. Dependent on the kinds of research question tackled, the underlying database technologies used may largely differ. The digitalisation of texts, for example, has resulted in a standard XML format [38] called TEI [41]. Recently researchers have started to use graph database technology in the context of digital humanities to interrelate text representations with further information [27].

Due to this dynamic digitalisation process, we present a model-driven approach to data quality analysis based on *patterns*. In contrast to related approaches [6, 19, 20, 26, 33], it allows for specifying anti-patterns for data quality problems that are generic concerning the underlying database technology and format. Such a generic pattern can be adapted to several database technologies, resulting in several abstract patterns. A domain expert chooses an abstract pattern as a *template* and concretises it to the domain-specific database format and to a concrete quality problem. Data analysts can apply the resulting concrete patterns to analyse their databases.

As a proof of concept, we implemented a realisation of our approach for XML databases with the Eclipse Modelling Framework [16]. To apply a pattern to XML data, we translate it to XQuery [12]. The approach was tested in detail for expressiveness and performance. In total, 85 % of the identified data quality problem variants are covered by our approach. The application to large cultural heritage databases of about 80 % of 43 patterns took less than 20 seconds.

In summary, this paper makes the following contributions:

- A literature survey on quality dimensions for research data and a study of quality problems occurring in cultural heritage data (Sec. 2)
- A model-driven approach to data quality analysis (Sec. 3)
- Tool support for XML data (Sec. 4)
- An evaluation wrt. expressiveness and performance (Sec. 5)

We discuss related work in Section 6 and conclude in Section 7.

## 2 DATA QUALITY PROBLEMS

Data quality problems can be considered as specific problem instances in one or more data *quality dimensions* [29]. We start by investigating several quality dimensions especially relevant in the context of research data. Thereafter, we present data quality problems occurring in cultural heritage data collected through a qualitative study and map them to the affected dimensions. Finally, we introduce a little example to illustrate a few data quality problems.

### 2.1 A Literature Survey

Since we present an approach for pattern-based quality analysis of research data, we focus on *aspects of data quality that are inherent to the data itself* and do not depend on any external impacts, such as technologies employed to provide or secure data. There are no common definitions of data quality dimensions [3, 29, 42]. Thus, based on a brief meta-survey we will define quality dimensions particularly relevant for research data in the following.

Zaveri et al. [42] presented 23 data quality dimensions, based on a review of 21 papers on quality assessment of linked open data. The identified dimensions are grouped as follows: The *accessibility* dimensions are related to access and retrieval of data by authorized humans or machines. The *intrinsic* dimensions focus on the data itself and in contrast to the *contextual* dimensions are independent of the usage context. *Trust* dimensions measure the trustworthiness of data. Dimensions related to the timeliness of data and the frequency of change over time are grouped to *dynamicity*. Dimensions related to the *representation* of data are also grouped. Batini et al. [3] identified the following dimensions to be frequently covered by approaches to classify data quality: *accuracy, completeness, consistency* and *timeliness*. Laranjeiro et al. [29] additionally identified *accessibility* to be often defined in the literature.

Since we focus on quality dimensions directly affected by the data itself, we exclude the accessibility dimensions described by Laranjeiro et al. [29] and Zaveri et al. [42]. However, we adopt the intrinsic dimensions described by Zaveri et al., namely *accuracy*, *consistency*, *conciseness* (i.e. *uniqueness*) and *timeliness*. We split the dimension that is referred to as *accuracy* into *correctness* and *precision* as this enables a finer differentiation, especially concerning the uncertainty often included in research data. Due to this uncertainty, the *trustworthiness* (called *believability* by Zaveri et al.) of data plays an important role as well. Since lack of knowledge is in the nature of research, we consider *completeness* as a dimension of interest. Furthermore, the data itself may show characteristics which negatively impact its *understandability*.

We briefly define the selected quality dimensions for research data as follows:

- **Correctness** is the degree to which the data correctly represents the real-world values (semantic correctness) and is free of syntactical errors (syntactic correctness).
- **Completeness** is the degree to which all required information is present in the data.
- **Consistency** is the absence of logical or representational contradictions within the data.
- **Precision** describes how exactly the data represents real-world values.
- **Uniqueness** is the unambiguous interpretability of data and thus the absence of redundancies.
- **Understandability** is the ease with which humans can read and interpret the data.
- **Timeliness** measures how up-to-date the data is.
- **Trustworthiness** is defined as the degree to which the data is accepted to be correct and credible.

### 2.2 Data Quality Problem Elicitation Considering Cultural Heritage Data

This paper emerged in the scope of the KONDA project[1]. The goal is to develop a continuous quality management process for cultural heritage data. We conducted 6 qualitative interviews and a workshop with 19 domain experts that perform acquisition, modelling, management and usage of various kinds of cultural heritage data (e.g. data on technical objects or artworks) to investigate the question: *What quality problems occur in cultural heritage data?*

We compiled a comprehensive specification of 94 data quality problems [24] through structured capturing of various aspects per problem, such as its impact on data quality and possible causes. 73 of the problems are directly related to the data itself. The others are beyond the scope of this paper as they depend on external impacts, such as data models.

By grouping the data-centric problems according to conceptual similarity and by further abstracting from the captured problems we created a list of rather general quality problems and variants presented in Table 1. For each problem, the table shows the affected quality dimensions. In the following, we will explain the less obvious problems and discuss relations between certain problems. The identified quality problems are not disjoint. Hence, a certain characteristic of data may imply multiple of the quality problems listed. *Illegal values*, for example, may be indicators for further problems such as misplaced information or misspellings. A *functional dependency* occurs if the values of a set of fields determine the values of another set of fields. *Contradictory relationships* are present if constraints concerning multiple references between records (e.g. irreflexivity and asymmetry) are violated. Regarding *imprecise data*, multiple possible alternatives may be listed or imprecise numerical values may be given. Abstract terms, ambiguous values (e.g. homonyms) and unexplained abbreviations cannot unambiguously be interpreted. *Misplaced information* occurs if values are placed into wrong fields or extraneous data is given in a field (e.g. title and name in a name field). *Semantically incorrect data* may overlap with

[1]The project "Kontinuierliches Qualitätsmanagement von dynamischen Forschungsdaten zu Objekten der materiellen Kultur unter Nutzung des LIDO-Standards" (KONDA) is funded by the German Federal Ministry of Education and Research.

**Table 1: Overview of identified data quality problems and affected quality dimensions.**

| Quality Problem | Affected Quality Dimensions |
|---|---|
| *Illegal values:* wrong datatype, domain violation (interval, set, syntax violation) | Syntactic correctness |
| *Missing data:* missing values, missing references, missing records, dummy values | Completeness |
| *Referential integrity violation* | Completeness |
| *Unique value violation* | Uniqueness |
| *Violation of a functional dependency* | Consistency, Semantic correctness |
| *Contradictory relationships* | Consistency, Semantic correctness |
| *Imprecise data:* alternative possible values, imprecise numerical values, abstract terms, ambiguous values, abbreviations | Precision, Uniqueness, Understandability |
| *Misplaced information:* misfielded values, extraneous data | Understandability, Consistency, Semantic correctness |
| *Redundant data:* exact duplicate records, approximate duplicate records, information placed in multiple locations | Uniqueness |
| *Heterogeneous data:* heterogeneous measure units, heterogeneous value representations, heterogeneous structural representations | Consistency, Understandability |
| *Misspellings* | Consistency, Understandability |
| *Semantically incorrect data:* false values, false references, doubtful data | Semantic correctness, Timeliness, Trustworthiness |

other quality problems such as *illegal values*, *functional dependency violations* and *contradictory relationships*.

The data quality problems described in the literature [19, 25, 29, 34–36] and those identified in our study overlap largely. However, the strategies for categorizing quality problems and the granularities of subdividing quality problems vary.

Note that most of the identified quality problems could be avoided in theory through constraints ensured during data creation. However, changes in requirements and constraints and also in the technologies and data formats used must be considered. Due to these dynamics, methods for retrospective data quality analysis are absolutely required and the model-driven (i.e. generic) approach is especially valuable. Furthermore, in some cases, data is valid with respect to the schema constraints but is still considered problematic and thus is relevant for data quality analysis. Examples are incomplete or imprecise data.

### 2.3 Running Example

To demonstrate concrete instances of some of the quality problems presented in Table 1, we introduce a simple example for XML data depicted in Listing 1. It is based on a schema which allows for describing buildings and architects. Each such element has an ID and includes a `name` element. `Building` elements additionally include a `city` and a `country` element. `Architect` elements further include *potential* `birthyears`. The following quality problems can be found in the example data:

(1) The `architect` record includes two specifications of the architect's year of birth. This indicates imprecise data, a form of uncertainty, since conflicting alternatives are listed. Hence, this example shows that data may be valid with respect to the schema but needs additional data quality analysis.

(2) The `building` records show a *violation of a functional dependency* between city and country. Both buildings are stated to be located in New York City, but their indicated countries differ. This suggests semantic incorrectness and inconsistency.

**Listing 1: Running example including two quality problems**

```
<data>
	<building id="1">
		<name>Empire State Building</name>
		<city>New York City</city>
		<country>USA</country>
	</building>
	<building id="2">
		<name>Chrysler Building</name>
		<city>New York City</city>
		<country>unknown</country>
	</building>
	<architect id="3">
		<name>William F. Lamb</name>
		<birthyear>1883</birthyear>
		<birthyear>1884</birthyear>
	</architect>
</data>
```

## 3 A MODEL-DRIVEN APPROACH TO DATA QUALITY ANALYSIS

Given a large variety of data quality problems which may have various concrete forms, a *model-driven approach* is promising to develop long-lasting concepts and tooling for data quality analysis independent of concrete technologies and formats. In the following, we will start with an overview of our approach and present some example patterns. Thereafter, we will introduce the metamodel for representing patterns.

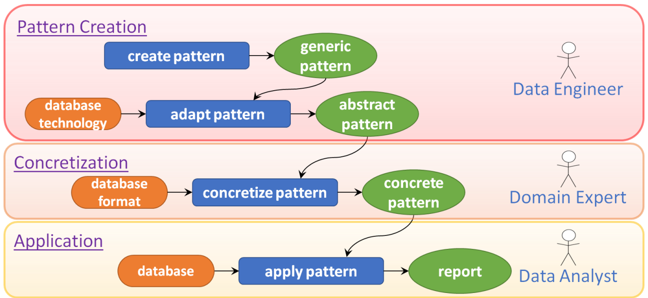


**Figure 1: Workflow of pattern creation and application**

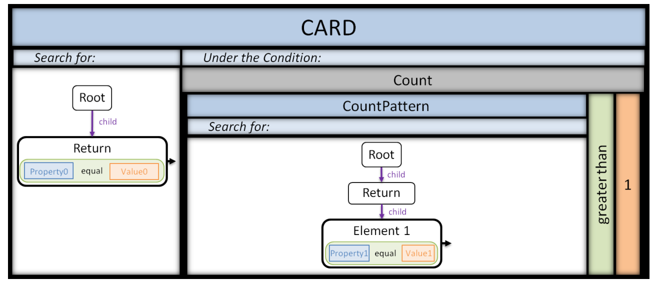


**Figure 2: Abstract pattern CARD for detecting violations of a cardinality restriction; XML-adaption**

## 3.1 The Overall Approach

Our proposed workflow for data quality analysis is visualised in Fig. 1. It starts with a *data engineer* creating a *generic pattern* for identifying a general and often domain-independent quality problem (that is not yet covered by existing patterns). Since the pattern detects phenomena that negatively impact data quality, it is actually an anti-pattern. Generic patterns are completely independent of the database technology and database format. A pattern is defined as a first-order logic expression over graph structures. Hence, data engineers must have a comprehensive understanding of both first-order logic and graphs. Furthermore, the ability to think abstractly, structurally and analytically is required. However, as we plan to provide a graphical modelling workbench, programming skills will not be required to define a pattern.

In the next step, an *abstract pattern* is created by adapting a generic pattern to a specific database technology (e.g. XML). Depending on the database technology this can be done automatically or semi-automatically with input from a data engineer. Abstract patterns are still independent of any database format (e.g. of a specific XML database).

In the next phase of the workflow, a *domain expert* (e.g. culture historian) chooses an abstract pattern fitting for the problem of interest and concretises it for the domain-specific database format. The result is a *concrete pattern*. To choose the right abstract pattern, at least a superficial understanding of first-order logic and graphs is necessary. To concretise a pattern, both an understanding of the database format and domain knowledge are required.

A *data analyst* can apply a concrete pattern (i.e. the generated query) to any database which conforms to the format that the pattern was concretised for. The result consists of all the data items that match the pattern and thus show some quality problem. The data analyst gets an overview of located quality problems and can decide then how to handle these problems, thus, how to initiate the improvement of data quality.

## 3.2 Example Patterns

To get a first idea of how abstract and concrete patterns may look like, we reconsider the example quality problems of Section 2.3. For each example, we specify an abstract pattern first and show it in a graphical form. Thereafter, the abstract pattern is concretised to the XML database format used in Listing 1 by setting several parameters. We omit the presentation of the corresponding generic patterns, which differ from the presented abstract patterns only in that they do not specify the types of relations and do not contain root elements as they are an XML-specific phenomenon. Examples of generic patterns are presented in Section D of the appendix.

Each pattern consists of two parts: a graph and a condition. The graph indicates the elements that are to be selected if they meet the condition. In the following diagrams it is shown on the left-hand side. The condition is visualised hierarchically on the right-hand side. It is a first-order logic expression over graph structures, which entails the following advantages. Graphs allow a comprehensible representation and expressing all kinds of data structures (e.g. trees). Over time, first-order logic has proven to be "adequate to the axiomatization of all ordinary mathematics" [15] and to be a good compromise between expressiveness and efficiency [18]. The number of graphs in the condition depends on its structure. Elements are bound from left to right. Properties are just shown as necessary.

*Problem 1.* The *abstract pattern* CARD which is visualised in Fig. 2 detects field repetitions that, for example, hint at imprecise data. The right part of this pattern specifies a count condition which is fulfilled if there is more than one element contained in the return element whose Property1 satisfies the given comparison relation.

To *concretise* the abstract pattern, a *domain expert* specifies properties and values by *setting several parameters*. They are indicated by the blue and orange boxes in the diagrams. For simplicity, relations and comparison operators are already predefined in the presented abstract patterns. To detect problem (1), the parameters need to be specified as shown in Listing 2. Since the abstract pattern is concretised for XML data, a property is defined by choosing `name`, `attribute` (plus specifying the name) or `content`.

**Listing 2: Concretisation of the CARD pattern (Fig. 2)**

```
Property0 and 1 = name, Value0 = "architect", Value1 = "birthyear"
```

Once all parameters are specified, the *concrete pattern* is automatically translated into a *query*. It can then be applied to a database by a *data analyst* to detect concrete problem occurrences. *The concrete pattern returns elements of type* `architect` *that contain more than one element of type* `birthyear`. Hence, when applied to the example data in Listing 1, it returns the `architect` element with `ID 3`.

*Problem 2.* The abstract pattern FUNC which is depicted in Fig. 3 finds violations of a functional dependency by detecting two container elements with two subelements each that are in specific comparison relations. One of the container elements is returned.

The pattern can be concretised for the running example by specifying the input values presented in Listing 3.

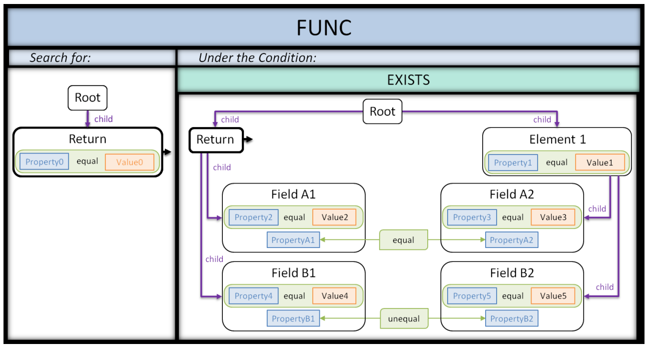


**Figure 3: Abstract pattern FUNC for detecting functional dependency violations; XML-adaption**

**Listing 3: Concretisation of the FUNC pattern (Fig. 3)**

```
Property0 to 5 = name, PropertyA and B = content,
Value0 and 1 = "building",
Value2 and 3 = "city", Value4 and 5 = "country"
```

*The concrete pattern finds elements of type* `building` *for which there exists another* `building` *stated to be located in the same* `city` *but in a different* `country`. This pattern is satisfied for the `building` elements with `ID 1` and `ID 2` in the example data in Listing 1.

## 3.3 A Metamodel for Data Quality Analysis

The core of our model-driven approach to data quality analysis is a metamodel for specifying patterns to localise quality problems in data. The metamodel allows expressing *generic, XML-adapted abstract and concrete patterns* as first-order logic expressions over graph structures. In Section 3.3.1, we will explain it in more detail. In Sections 3.3.2 and 3.3.3, we will discuss which parts of the metamodel are affected by the adaption of a *generic pattern* to a specific database technology and the concretisation of an *abstract pattern*. An example abstract pattern presented as an instance model can be found in Section B.1 of the appendix.

*3.3.1 Metamodel Overview.* Fig. 4 shows the most relevant part of our metamodel; the complete version and well-formedness rules are given in Section A of the appendix. The metamodel has five packages: `patternstructure` is concerned with the logical structure of a pattern, `graphstructure` is used to specify pattern graphs, `operators` contains operators to define conditions on pattern elements, `parameters` holds placeholders that are not specified in generic or abstract patterns but are to be set in concrete patterns. Package `adaptionxml` holds classes which enable the adaption of a generic pattern to XML; it will be discussed in Section 3.3.2. It is not required for representing *generic* patterns. The design of the packages `patternstructure` and `graphstructure` is inspired by the implementation design of nested graph conditions presented by Nassar in [32].

The package **`patternstructure`** includes classes for specifying the logical structure of a pattern. Each *generic, abstract or concrete pattern* is represented as an instance of `CompletePattern`, which extends `Pattern`. A `Pattern` always contains a `Graph` and a `Condition`. The `Graph` determines which elements should be returned if the `Condition` is satisfied. By nesting `Conditions` we can express first-order logic formulas over graph structures. There are four types of conditions. A `QuantifiedCondition` is specified via a `quantifier`. It contains a `Graph` specifying the domain of discourse as well as a further `Condition`. A `Formula` is specified by a logical `operator` and two further conditions which serve as arguments. The logical operator *not* is modelled separately in a `NotCondition` as it has only one argument. A `CountCondition` allows expressing a cardinality constraint. It compares a `CountPattern` with another `CountPattern` or a `NumberElement`, thus a primitive number, via a specified operator. Just like a `CompletePattern`, the `CountPattern` consists of a graph and a nested condition. The `CountPattern`, however, does not represent the matches themselves but instead represents the *number* of matches. These occurrences of the pattern in the data can partly overlap concerning the matched elements.

The package **`graphstructure`** allows specifying `Graphs` in *generic patterns* as compositions of named `Elements` and arbitrary many directed `Relations` in between. Elements of different graphs of a pattern correspond to each other if they have the same name. Each `Element` may contain arbitrary many `Properties` which may be subject to conditions. If a `Graph` is contained in a `QuantifiedCondition`, each included `Element` that does not correspond to an `Element` in a previous `Graph` is bound by the condition's `quantifier`. The elements returned by the pattern are specified by the `returnElements` association between `Graph` and `Element`.

The package **`operators`** allows defining conditions on elements. Each `Operator` is assigned to exactly one `Graph` by being contained in its `operatorList`. The arguments of an operator must be components of this graph. A `Comparison` has two `Comparable` items (of type `Element`, `Operator`, `Property`, `ParameterValue` or `UnknownParameterValue`) as arguments. The concrete operator is specified as a parameter via the association `optionParam`, which will be explained later on. A `Match` operator checks a `Property` for a regular expression that is given as a parameter.

The package **`parameters`** holds `Parameters` for indicating which information is not yet given in a generic or abstract pattern but shall be available in a *concretisation* of that pattern. Each parameter is contained in the `parameterList` of the `CompletePattern`. Parameter values can be predefined in a generic or abstract pattern or a description can be provided. Subclasses of `ParameterValue` (only partly shown in Fig. 4) each represent concrete literal values of a specific type. In a generic or abstract pattern, the argument types of a `Comparison` do not have to be set. Thus, an `UnknownParameterValue` can be used as a placeholder for a concrete `ParameterValue`. The class `OptionParam` allows specifying parameters whose domain is defined via an enumeration of values of type `T` given in the `options` attribute. In a concrete pattern, the chosen value for this parameter is specified in the `value` attribute. This class is used, for example, to specify the concrete operator of a `Comparison`, e.g. `less than`.

*3.3.2 Adaption to a Specific Database Technology.* The packages discussed above allow the definition of *generic patterns* independent of any database technology. To represent *abstract patterns* adapted to a specific database technology, `Element`, `Relation` and `Property` must be subclassed correspondingly. For the adaption of a *generic pattern*, instances of `Element`, `Relation` and `Property` in the pattern

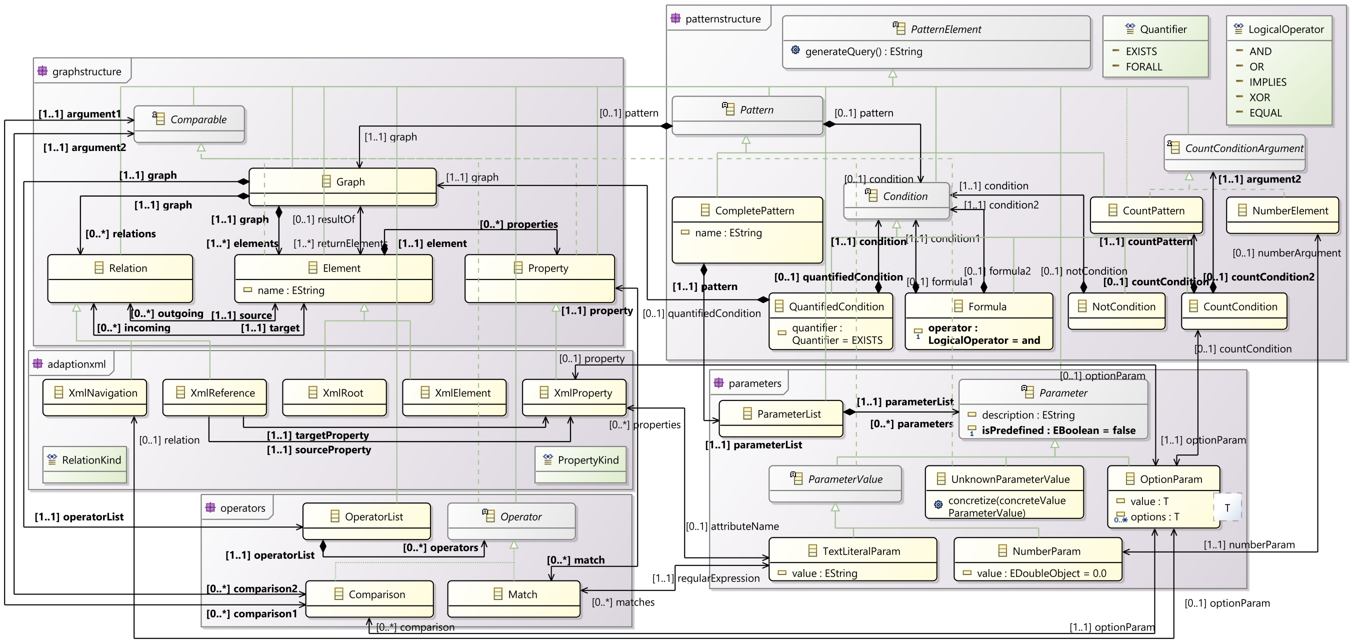


**Figure 4: Condensed metamodel for patterns which allow localising quality problems in data**

must be replaced by instances of corresponding subclasses via a (semi-) automatic algorithm.

*Adaption to XML.* Our implementation provides classes for representing XML-adapted abstract patterns in the package **adaptionxml**. In the course of the adaption, the graphs of the generic pattern are transformed into trees spanned by relations based on XPath [39] axes (represented by `XmlNavigation`). Edges between the branches of these trees represent identifier-based references within the XML data (represented by `XmlReference`).

The class `XmlRoot` represents the root of the XML document and serves as the root of the trees in the pattern. The class `XmlElement` represents XML elements in general. Each `XmlElement` has exactly one incoming `XmlNavigation`. The axis is specified by the association `optionParam<RelationKind>`. The enumeration `RelationKind` comprises XPath axes. Besides the axes `child` and `descendant` for representing containment relations, further axes are also practical. The axes `self` and `descendant-or-self` are useful if multiple elements of a pattern graph may correspond to the same XML element. The axis `following`, for example, is relevant for XML data in which the order of elements has an impact on the meaning of that data. An XML element may have the following three kinds of properties that we encoded in the enumeration `PropertyKind`: an element name, a named attribute and some content between the start and end tag. The kind of an `XmlProperty` is specified by `optionParam<PropertyKind>`. If a named attribute is addressed, its name is specified by the `attributeName` of type `TextLiteralParam`.

For the adaption of a generic pattern to XML, all instances of `Element` and `Property` are automatically replaced by instances of `XmlElement` and `XmlProperty`, respectively. Next, the data engineer decides for each `Relation` whether it represents an XPath axis (`XmlNavigation`) or a reference (`XmlReference`). The instance of `Relation` is replaced correspondingly. In the latter case, `Properties` are automatically inserted into the `source` and `target` elements. In the last step, the `XmlRoot` is automatically inserted in each graph of the pattern. For each `XmlElement` which has no incoming `XmlNavigation`, an incoming `XmlNavigation` from `XmlRoot` is inserted. Thereby the tree structure is completed.

*Adaption to other database technologies.* To support other database technologies, further adaption packages with subclasses of `Element`, `Relation` and `Property` as well as the corresponding parameter kinds need to be implemented.

In a *relational database*, for example, an `Element` corresponds to a row of a specific table while a `Property` represents a column of the table. To specify a relation between elements, the names of the columns holding the foreign and primary key are required. Thus, the new subclass of `Relation` compares properties of both elements analogously to the `XmlReference` class.

In *graph databases* which are specified via subject-predicate-object triples, an `Element` corresponds to a node. A primitive predicate of an `Element` may be represented as a `Property` and a relation between two nodes as a `Relation`. The new subclasses of `Property` and `Relation` must be implemented correspondingly.

To increase the generality of generic patterns, relations expressing an identity (like the `self` axis mentioned above) should be supported for other database technologies as well.

### 3.3.3 *Concretisation.*

As discussed above, all variable parts of an abstract pattern are modelled as `Parameters` contained in the `parameterList` of a `CompletePattern`. To concretise an abstract pattern, these parameters must be modified as follows. For each contained `OptionParam` one of the options must be chosen as the `value`. Each `UnknownParameterValue` must be replaced by a concrete `ParameterValue` via the method `concretize`. Furthermore, each `value` of a `ParameterValue` must be specified.

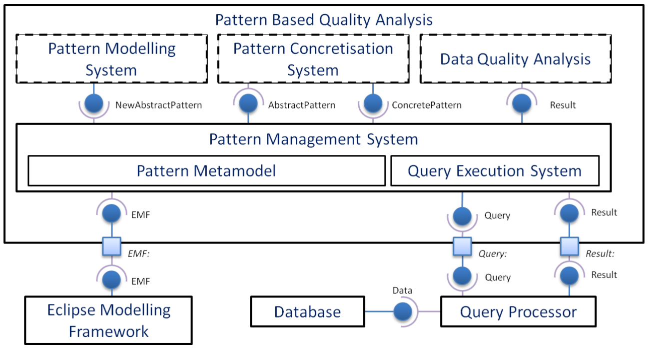


**Figure 5: Component diagram of the implementation**

## 4 TOOL SUPPORT

In the following, we report on a proof-of-concept implementation, in which we have instantiated our conceptual approach to XML.

### 4.1 Tool architecture

Fig. 5 shows the architecture of our tool implementation. The `Pattern Management System` stores predefined and user-defined generic, abstract and concrete patterns. It encompasses an implementation of the metamodel discussed in Section 3.3 based on the Eclipse Modelling Framework (EMF) [16]. The implementation includes an algorithm for adapting a generic pattern to XML (see Section 3.3.2) and an algorithm for translating concrete patterns to XQuery [12], which will be shortly presented in Section 4.2.

The `Query Execution System` applies a chosen set of patterns to a selected database by evaluating the generated queries. In our implementation this is done via BaseX [22], an open-source XQuery processor. The result is passed to the `Data Quality Analysis` component, which visualises the matched data regions together with metadata on the query execution.

The dashed components in Fig. 5 form the front end of our implementation and are currently under development. We will use Eclipse Sirius [17] to develop a graphical modelling workbench for specifying generic and abstract patterns. For the concretisation we will provide a form-based view for entering parameter values.

### 4.2 Mapping Patterns to Queries

To detect quality problems by concrete patterns, they must be translated into queries. The query language and thus the algorithm for translation depends on the database technology chosen.

*4.2.1 Mapping Patterns to XQuery.* We implemented a systematic translation of any concrete pattern represented via the metamodel to XQuery [12] by means of the `generateQuery()` method of each class. Each query consists of a single or multiple `for` clauses, a `where` clause and a `return` clause. Listing 4 shows the generated XQuery expression for the concrete `FUNC` pattern presented in Section 3.2. Further generated queries are shown in Section C of the appendix.

For the translation of a `CompletePattern`, first its `Graph` is traversed by navigating along outgoing `XmlNavigations` and starting at the `XmlRoot`. For each such relation, a `for` clause is appended to the query. Its path expression specifies the navigation from the `source` element to the `target` element via the given XPath axis. Conditions at the `target` element as well as incoming or outgoing `XmlReferences` are translated to XPath predicates inside square brackets following the path expression.

A pattern's nested condition is translated step-by-step to a nested expression being of the kinds `some`, `every`, `and`, `or`, `not` or `count`; this expression constitutes the query's `where` clause. The graphs contained in a `QuantifiedCondition` are traversed as described above. Each `XmlElement` which does not correspond to an element in a previous graph or is subject to a condition is translated to a `some` or `every` expression depending on the given quantifier. The path expressions and predicates are created as discussed above. A `CountPattern` is translated as follows. Its graph and the nested condition are translated as those of a `CompletePattern`. The resulting expression serves as the argument to a `count` expression.

The `return` clause determines that the pattern's `returnElements` are returned by the query.

**Listing 4: XQuery for the concrete FUNC pattern**

```
for $var1 in /child::*[./name()="building"]
where some $var2 in $var1/child::*[./name()="city"]
 satisfies some $var3 in $var1/child::*[./name()="country"]
 satisfies some $var4 in /child::*[./name()="building"]
 satisfies some $var5 in $var4/child::*[./name()="city"]
  [$var2/data()=./data()]
 satisfies some $var6 in $var4/child::*
  [./name()="country"][$var3/data()!=./data()]
satisfies true()
return $var1
```

*4.2.2 Mapping Patterns to other Query Languages.* To enable the application of patterns adapted to other database technologies, analogous algorithms for translating concrete patterns into corresponding query languages must be implemented. For example, patterns adapted for relational databases may be translated into SQL [10]. Patterns adapted for RDF [30] databases may be translated to SPARQL [23]. There already exist approaches [19, 20, 26] for pattern-based quality analysis of linked data based on SPARQL templates. We will compare with those in detail in Section 6.

## 5 EVALUATION

For the evaluation we will investigate two research questions:

**RQ1:** How far does our approach enable the detection of quality problems in research data?
**RQ2:** What is the query response time of the problem detection?

RQ1 will be discussed based on the application of our approach to two cultural heritage databases in Section 5.1 and on a more general level in Section 5.2. RQ2 will be answered in Section 5.3.

### 5.1 Application to Cultural Heritage Databases

To evaluate our approach, we compiled a representative selection of quality problems, created patterns to detect these problems and applied them to two databases. The first database is present in MIDAS [8], an XML-format for the description of art-historical objects. This database holds information on more than 700,000 such objects plus related entities in more than 47 million XML elements. The second database is present in LIDO [9], a CIDOC-CRM [11] application and XML format for harvesting and exchanging metadata of collectibles. It describes more than 300,000 cultural heritage objects in more than 48 million XML elements.

Remember that the list of quality problems shown in Table 1 is the result of our elicitation of quality problems occurring in cultural heritage databases. For the evaluation, our goal was to cover each problem variant that is listed in Table 1 by a pattern. We selected concrete quality problems that were either explicitly mentioned by domain experts or which violate the LIDO schema specification [9] or the MIDAS manual [8]. We created a set of patterns that cover 85 % of the problem variants listed in Table 1. However, *false values and references*, *missing records* and *misspellings* could not be covered. We will elaborate on this in the next section.

We created altogether *21 generic patterns* and adapted them to XML, resulting in *21 abstract patterns*. Some of the patterns are closely related to those presented by Kontokostas et al. [26] and Fürber et al. [19] [20]. Table 2 covers 27 problem variants. As mentioned above, four variants could not be covered. Hence, to detect all other variants in both databases, 46 *concrete patterns* would be needed. Because internal references are not intended in the LIDO format, LIDO data cannot be checked for *referential integrity violations* and *contradictory relationships*. The integrity of references to external resources such as published controlled vocabulary cannot be checked with our current implementation. As the LIDO schema specification explicitly allows multiple different measure units, we do not consider heterogeneous units in LIDO databases to be a problem and thus did not create a corresponding pattern. Hence, we created a set of *43 representative concrete patterns* in total. We applied them to our selected databases and checked, for each pattern, a random sample of the matched elements for correctness. The generic, abstract and concrete patterns are presented in Section D of the appendix.

*We conclude that, for cultural heritage data, our approach is effective since it enables the detection of 85% of the quality problem variants elicited in our study.* This answers RQ1 in the context of the application of our approach to cultural heritage databases. We will discuss RQ1 on a more general level in the following section. The main *threat to validity* is that we applied our approach only to research data of one domain (i.e. cultural heritage) and one database technology (i.e. XML).

## 5.2 Expressiveness

In the following, we will discuss the expressiveness of our approach with regard to the data quality problems that can be detected. Table 2 gives an overview of *potentials* enabling and *limitations* preventing the detection of certain quality problems. Problem variants are listed only if the affected potentials and limitations differ significantly.

*5.2.1 Potentials.* The strength of our approach lies in the detection of problems that reveal themselves through the syntax and especially, the structure of the data. In the following, we will discuss important features for enabling this functionality.

Our approach allows checking values concerning *regular expressions*. This enables the detection of syntactical patterns that hint at certain quality problems, e.g. implicitly encoded *imprecise numerical values*. Further, *misplaced information*, *misspellings* and *semantically incorrect data* can be detected via regular expressions if they are accompanied by violations of syntactical rules.

Moreover, our approach allows casting values in the data via predefined operations and *comparing* these values to other values in the data or constant literals, e.g. to detect *interval violations*.

By *enumerations*, we can check whether a value is (not) contained in a list of constant literals and thereby detect problems such as *set violations* and *dummy values*. This could alternatively be checked using regular expressions or disjunction of multiple graphs.

Our approach allows checking the *existence of complex graph structures*, which may include *comparisons* between elements and properties. For example, this enables the detection of *contradictory relationships* of a fixed size.

A further feature is the ability to express conditions concerning the *number of occurrences* of a pattern in the data. For example, this allows detecting *unique value violations*. Restricted forms of *approximate duplicate records* can be detected by specifying a fixed set of properties whose equalities across records indicate that the records are duplicates.

A particular strength of our approach lies in *first-order logic expressions over graph structures*. For example, this allows checking the non-existence of certain graph structures. Hence, *referential integrity violations* can be detected, which additionally hint at *missing records*. Via first-order logic expressions and comparisons, our approach allows for checking complex domain-specific dependencies between values of one or multiple related records. For example, an artist's date of birth must precede the dates of creation of his or her artworks. Violations of such plausibility rules hint at *semantically incorrect data*.

*5.2.2 Limitations.* In the following, we will discuss the limitations of our approach and affected quality problems.

Besides calculating the number of occurrences of a pattern in the data, our approach does not include any *metrics* currently, e.g. for similarity checks [14, 40]. This prevents the detection of *misspellings* and *approximate duplicates* in general. We plan to investigate the integration of such metrics in the future.

We detected occurrences of *dummy values*, *ambiguous values*, *abstract terms* and *heterogeneous value representations* (e.g. synonyms) in the cultural heritage data by the enumerations feature discussed above. The corresponding patterns scan the data for keywords manually specified by domain experts. Integrating a fuzzy search mechanism [4, 7, 28] based on domain-specific *ontologies* would enable the universal detection of such problems.

As discussed above, our approach supports the comparison of values within the data, e.g. as part of plausibility rules. For the selected cultural heritage data, there hardly exist any syntactical rules (e.g. on how to record a date). Therefore, the syntax is very heterogeneous. Furthermore, uncertainties are often encoded implicitly. Thus, the comparison of such values without applying *complex operations on these values* beforehand is not possible. We will investigate this aspect in future work.

Some data quality problems require *external knowledge about the real world* to be detected. For instance, our approach does not allow detecting any *semantically incorrect values* that, however, satisfy the specified plausibility rules and do not show through illegal values, functional dependency violations or contradictory relationships. The required real world values, however, might be present in other databases. In future work, we will investigate the comparison of

**Table 2: Potentials allowing and limitations preventing the detection of quality problems.**

| Quality Problem | Variants | *Potentials* | | | | | | *Limitations* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **RE** | **C** | **E** | **GS** | **OC** | **FOL** | **M** | **O** | **CVO** | **RWV** | **HOL** |
| *Illegal values* | wrong datatype | ✓ | | | ✓ | | | | | | | |
| | domain violation (interval, set, syntax) | ✓ | ✓ | ✓ | | | | | | | | |
| *Missing data* | missing values | ✓ | ✓ | | ✓ | | ✓ | | | | | |
| | missing references and records | | | | ✓ | | ✓ | | | | × | |
| | dummy values | ✓ | ✓ | ✓ | | | | | × | | | |
| *Referential integrity violation* | | | | | ✓ | | ✓ | | | | | |
| *Unique value violation* | | | ✓ | | ✓ | ✓ | | | | | | |
| *Violation of a functional dependency* | | | ✓ | | ✓ | | | | | | | |
| *Contradictory relationships* | | | | | ✓ | | | | | | | × |
| *Imprecise data* | alternative possible values | ✓ | | | ✓ | ✓ | | | | | | |
| | imprecise numerical values | ✓ | | | ✓ | | | | | | | |
| | abstract terms and ambiguous values | | | ✓ | | | | | × | | | |
| | abbreviations | ✓ | | | | | | | | | | |
| *Misplaced information* | | ✓ | | | | | | | | | × | |
| *Redundant data* | | | ✓ | | ✓ | ✓ | | × | | | | |
| *Heterogeneous data* | heterogeneous measure units | ✓ | | | | | | | | | | |
| | heterogeneous value representations | ✓ | | ✓ | | | | | × | | | |
| | heterogeneous structural representations | | | | ✓ | | ✓ | | | | | |
| *Misspellings* | | ✓ | | | | | | × | | | | |
| *Semantically incorrect data* | | ✓ | ✓ | | ✓ | ✓ | ✓ | | | × | × | |

**RE**: regular expressions, **C**: comparisons, **E**: enumerations, **GS**: graph structures, **OC**: occurrence count (metric), **FOL**: first-order logic, **M**: further metrics, **O**: ontology, **CVO**: complex value operations, **RWV**: real-world values, **HOL**: higher-order logic

records from different databases (possibly with different formats and underlying technologies) to detect semantically incorrect values. If the data does not contain any hints for a relation or record to be missing (e.g. a referential integrity violation), then the *absence of relations and records* can also not be detected.

As we do not support *higher-order logic*, we cannot express *contradictory relationships* of variable size.

*5.2.3 Summary.* Concerning RQ1, we found that, in general, the strength of our approach lies in expressing first-order logic conditions over arbitrarily complex graph structures, thereby detecting problems that reveal themselves through the structure of the data. Problems that require the ability to assess the meaning of data, instead, or the knowledge of real-world values to be detected, are not covered by our approach yet.

## 5.3 Performance

We measured the runtimes of the 43 applied patterns via BaseX, which is a "main memory application" [22], on a Windows 10 PC with a 4.2 GHz CPU and 16 GB RAM. Table 3 gives a rough overview.

To analyse the factors that impact the runtime, we take a closer look at the generated queries. For each `Element` in the pattern a nested loop expression is created. We observed that in practice, these expressions typically select only elements on one specific level in the XML hierarchy (via the `child` axis) or all elements below a certain level (via the `descendant` axis) that is not the top level. Nevertheless, some of the patterns are translated into multiple nested loops that when applied to the large cultural heritage databases iterate through very large sets of XML elements. The patterns that ran longer than 4 minutes are concerned with the following quality problems: *unique value violation*, *violation of a functional dependency*, *exact and approximate duplicate records.* For example, the pattern that detects functional dependency violations in the LIDO data, is translated into a query consisting of 7 nested loops. Its application results in more than 77 billion element comparisons and a runtime of almost six hours.

**Table 3: Runtimes of patterns applied to cultural heritage databases**

| Percentage of patterns | 70 % | 80 % | 90 % | 95 % |
|---|---|---|---|---|
| Runtime | < 10 s | < 20 s | < 4 min | < 6 h |

*Concerning RQ2 we conclude that 80% of the patterns can easily be executed each time the data is modified without causing major limitations in day-to-day work since they run less then 20 seconds.* Others may run multiple hours or days and thus, in practice, should be

**Table 4: Comparison of related work with our approach**

| Approach | Generality | Expressiveness | Pattern Language |
|---|---|---|---|
| Kontokostas et al. [26] | abstract (RDF) | RE, C, GS, OC, FOL | SPARQL |
| Fürber et al. [19, 20] | abstract (RDF) | RE, C, E, GS, OC, FOL | SPIN |
| Bizer et al. [6] | concrete (Named Graphs) | RE, C, GS, PL | WIQA-PL |
| Bicevska et al. [33] | concrete (relational) | C, GS, FOL, CS, GR | graphical DSL, SQL |
| **Our approach** | generic (PoC: XML) | RE, C, E, GS, OC, FOL | metamodel (+ GUI) |

**RE**: regular expressions, **C**: comparisons, **GS**: graph structures, **OC**: occurrence count, **E**: enumerations, **PL**: propositional logic, **CS**: column sum, **GR**: group rows

executed less often. The main *threat to validity* is that we considered only two XML databases of a single domain.

The runtime can be tweaked by designing patterns such that the number of visited XML elements is minimised, e.g., by using a fixed number of `child` axes instead of the `descendant` axis or by applying patterns only to parts of the database. In future work we will investigate how to support users in designing efficient patterns.

## 6 RELATED WORK

In our literature study, we focused on pattern-based approaches to data quality analysis presented in research papers. For each approach, we determine the level of abstraction, measure the expressiveness by means of the features discussed in Section 5.2 and consider the pattern language. Table 4 gives an overview.

Kontokostas et al. [26] and Fürber et al. [19, 20] presented similar approaches to detect quality problems in Linked Data based on parameterized SPARQL or SPIN query templates. Hence, they do not consider patterns on the generic level. The approaches have similar potentials and limitations as ours. In the evaluation part of their paper, Kontokostas et al. identified the necessity of creating further abstract patterns to cover a wider range of quality problems. However, both approaches require abstract patterns (i.e. queries) to be written in SPARQL or SPIN containing more implementation details than in our model-driven approach.

Bizer et al. [6] presented a policy framework for quality-driven information filtering. They consider patterns on the concrete level only. The approach is limited to propositional logic and does not support the counting of occurrences of a pattern in the data. Patterns are expressed with a custom SPARQL-based language.

Bicevska et al. [33] proposed to specify data quality requirements for specific database technologies and formats using a domain-specific language (DSL). Thus, concrete patterns are considered only. The approach differs slightly from the others with regard to the supported features. The pattern language, i.e., the DSL, is not presented in the paper. Informal explanations are used instead to describe examples for quality specifications. The authors proposed to translate them into a query language, such as SQL, but did not present an algorithm. To motivate their approach, Bicevska et al. discussed the use of the object constraint language (OCL) to define data quality. OCL is powerful enough to specify constraints in first-order logic and beyond. However, it is not well-suited for domain experts without good skills in object-oriented programming. Furthermore, OCL is fully typed and therefore, well-suited for structured data, whereas research data is often semi-structured [2].

*In summary, our approach is the only one that supports generic patterns and thus, is flexible concerning the underlying database technology.* The expressiveness of most of the approaches considered seems to be similar to ours. Existing pattern languages often require the specification of details that may require skills that domain experts usually do not have. By being model-driven, our approach abstracts from query languages. This lays the foundation for a potentially user-friendly and universal representation of patterns.

In future work we will compare our approach also to non-pattern-based approaches and industrial tools [13] to detect quality problems, for example via machine learning [1] or metric-based methods [31]. Furthermore, we will compare our work to approaches for generic data model definitions [37].

## 7 CONCLUSION

To support a dynamic digitalisation of specific scientific fields, especially the humanities, we presented a model-driven approach to analyse the quality of research data. It supports the specification of patterns to identify data quality problems, independent of the underlying database technology and format. A proof-of-concept implementation shows how this approach can be used for XML databases. We evaluated it for expressiveness and performance by applying it to two cultural heritage databases. While its expressiveness is comprehensive for pattern-based approaches, it has to be integrated with further techniques for quality analysis such as metrics and ontologies, to cover the wide range of data quality problems.

In future work, we will empirically evaluate the usability of the proposed pattern notation. As we currently see no obstacle to applying our approach to research data from other scientific fields such as biodiversity and also non-research data, we will investigate those applications in the future. To evaluate the overall concept of our approach we will implement it for further database technologies. Further ahead, we plan to combine our approach with further analysis techniques such as machine learning.

Our overall goal is to develop a framework for quality assurance of research data, where the detection of quality problems is the first essential step.

## ACKNOWLEDGMENTS

This work has been developed in the project KONDA. It is partly funded by the German Federal Ministry of Education and Research (Grant No.: 16QK06A). We would like to thank the members of the project for the excellent collaboration and support. Many thanks to Oguzhan Balandi, Michalis Famelis and Regine Stein for the extensive reviews and their constructive criticism. Further we would like to thank the anonymous reviewers for their valuable suggestions.

# REFERENCES


[1] Ziawasch Abedjan, Cuneyt Gurcan Akcora, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2015. Temporal Rules Discovery for Web Data Cleaning. *Proc. VLDB Endow.* 9, 4 (2015), 336–347. https://doi.org/10.14778/2856318.2856328

[2] Serge Abiteboul. 1997. Querying Semi-Structured Data. In *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings (Lecture Notes in Computer Science)*, Foto N. Afrati and Phokion G. Kolaitis (Eds.), Vol. 1186. Springer, 1–18. https://doi.org/10.1007/3-540-62222-5_33

[3] Carlo Batini, Cinzia Cappiello, Chiara Francalanci, and Andrea Maurino. 2009. Methodologies for data quality assessment and improvement. *ACM Comput. Surv.* 41, 3 (2009), 16:1–16:52. https://doi.org/10.1145/1541880.1541883

[4] Simran Bijral and Debajyoti Mukhopadhyay. 2014. Efficient Fuzzy Search Engine with B -Tree Search Mechanism. In *2014 International Conference on Information Technology, ICIT 2014, Bhubaneswar, India, December 22-24, 2014*. IEEE, 118–122. https://doi.org/10.1109/ICIT.2014.19

[5] Paul V. Biron and Ashok Malhotra. 2004. *XML Schema Part 2: Datatypes Second Edition.* W3C Recommendation. W3C. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[6] Christian Bizer and Richard Cyganiak. 2009. Quality-driven information filtering using the WIQA policy framework. *J. Web Semant.* 7, 1 (2009), 1–10. https://doi.org/10.1016/j.websem.2008.02.005

[7] Dario Bonino, Fulvio Corno, Laura Farinetti, and Alessio Bosca. 2004. Ontology driven semantic search. *WSEAS Transaction on Information Science and Application* 1, 6 (2004), 1597–1605.

[8] Jens Bove, Lutz Heusinger, and Angela Kailus. 2001. *Marburger Informations-, Dokumentations- und Administrations-System (MIDAS): Handbuch und CD (Literatur und Archiv; 4). - 4. überarbeitete Auflage.* https://archiv.ub.uni-heidelberg.de/artdok/3770/

[9] Erin Coburn, Richard Light, Gordon McKenna, Regine Stein, and Axel Vitzthum. [Online]. LIDO (Lightweight Information Describing Objects). http://network.icom.museum/cidoc/working-groups/lido/. http://network.icom.museum/cidoc/working-groups/lido/

[10] C. J. Date and Hugh Darwen. 1997. *A Guide to SQL Standard, 4th Edition.* Addison-Wesley.

[11] Martin Doerr, George Bruseker, Chryssoula Bekiari, Christian Emil Ore, Thanasis Velios, and Stephen Stead. 2020. Definition of the CIDOC Conceptual Reference Model. http://www.cidoc-crm.org/sites/default/files/CIDOCCRM_v6.2.930-4-2020.pdf. (Accessed on 05/15/2020).

[12] Michael Dyck, Jonathan Robie, and Josh Spiegel. 2017. *XQuery 3.1: An XML Query Language.* W3C Recommendation. W3C. https://www.w3.org/TR/2017/REC-xquery-31-20170321/.

[13] Lisa Ehrlinger, Elisa Rusz, and Wolfram Wöß. 2019. A Survey of Data Quality Measurement and Monitoring Tools. *CoRR* abs/1907.08138 (2019). arXiv:1907.08138 http://arxiv.org/abs/1907.08138

[14] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE Trans. Knowl. Data Eng.* 19, 1 (2007), 1–16. https://doi.org/10.1109/TKDE.2007.250581

[15] William Ewald. 2019. The Emergence of First-Order Logic. In *The Stanford Encyclopedia of Philosophy* (spring 2019 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

[16] Eclipse Foundation. [Online]. Eclipse Modeling Project. https://www.eclipse.org/modeling/emf/

[17] Eclipse Foundation. [Online]. Sirius. https://www.eclipse.org/sirius/

[18] Enrico Franconi, Alessandro Mosca, Xavier Oriol, Guillem Rull, and Ernest Teniente. 2019. $\text{OCL}_{\text{FO}}$: First-Order Expressive OCL Constraints for Efficient Integrity Checking. *Softw. Syst. Model.* 18, 4 (Aug. 2019), 2655 – 2678. https://doi.org/10.1007/s10270-018-0688-z

[19] Christian Fürber and Martin Hepp. 2010. Using SPARQL and SPIN for Data Quality Management on the Semantic Web. In *Business Information Systems, 13th International Conference, BIS 2010, Berlin, Germany, May 3-5, 2010. Proceedings (Lecture Notes in Business Information Processing)*, Witold Abramowicz and Robert Tolksdorf (Eds.), Vol. 47. Springer, 35–46. https://doi.org/10.1007/978-3-642-12814-1_4

[20] Christian Fürber and Martin Hepp. 2011. Swiqa - a semantic web information quality assessment framework. In *19th European Conference on Information Systems, ECIS 2011, Helsinki, Finland, June 9-11, 2011*, Virpi Kristiina Tuunainen, Matti Rossi, and Joe Nandhakumar (Eds.). 76. http://aisel.aisnet.org/ecis2011/76

[21] German Council for Scientific Information Infrastructures (RfII). 2020. *The Data Quality Challenge. Recommendations for Sustainable Research in the Digital Turn.* Göttingen. http://www.rfii.de/?p=4203

[22] Christian Grün. 2006. Pushing XML Main Memory Databases to their Limits. In *Tagungsband zum 18. GI-Workshop über Grundlagen von Datenbanken (18th GI-Workshop on the Foundations of Databases), Wittenberg, Sachsen-Anhalt, Deutschland, 6.-9. Juni 2006*, Stefan Brass and Alexander Hinneburg (Eds.). Institute of Computer Science, Martin-Luther-University, 60–64. http://dbs.informatik.uni-halle.de/GvD2006/gvd06_gruen.pdf

[23] Steven Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language.* W3C Recommendation. W3C. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[24] Arno Kesper, Markus Matoni, Julia Rössel, Michelle Weidling, and Viola Wenz. 2020. Catalog of Quality Problems for Data, Data Models and Data Transformations. https://doi.org/10.5281/zenodo.3955500

[25] Won Y. Kim, Byoung-Ju Choi, Eui Kyeong Hong, Soo-Kyung Kim, and Doheon Lee. 2003. A Taxonomy of Dirty Data. *Data Min. Knowl. Discov.* 7, 1 (2003), 81–99. https://doi.org/10.1023/A:1021564703268

[26] Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen, and Amrapali Zaveri. 2014. Test-driven evaluation of linked data quality. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel (Eds.). ACM, 747–758. https://doi.org/10.1145/2566486.2568002

[27] Andreas Kuczera. 2016. Digital Editions beyond XML - Graph-based Digital Editions. In *Proceedings of the 3rd HistoInformatics Workshop on Computational History (HistoInformatics 2016) co-located with Digital Humanities 2016 conference (DH 2016), Krakow, Poland, July 11, 2016 (CEUR Workshop Proceedings)*, Marten Düring, Adam Jatowt, Johannes Preiser-Kappeller, and Antal van den Bosch (Eds.), Vol. 1632. CEUR-WS.org, 37–46. http://ceur-ws.org/Vol-1632/paper_5.pdf

[28] Lien Fu Lai, Chao-Chin Wu, Pei-Ying Lin, and Liang-Tsung Huang. 2011. Developing a fuzzy search engine based on fuzzy ontology and semantic search. In *FUZZ-IEEE 2011, IEEE International Conference on Fuzzy Systems, Taipei, Taiwan, 27-30 June, 2011, Proceedings*. IEEE, 2684–2689. https://doi.org/10.1109/FUZZY.2011.6007378

[29] Nuno Laranjeiro, Seyma Nur Soydemir, and Jorge Bernardino. 2015. A Survey on Data Quality: Classifying Poor Data. In *21st IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2015, Zhangjiajie, China, November 18-20, 2015*, Guojun Wang, Tatsuhiro Tsuchiya, and Dong Xiang (Eds.). IEEE Computer Society, 179–188. https://doi.org/10.1109/PRDC.2015.41

[30] Ora Lassila. 1999. *Resource Description Framework (RDF) Model and Syntax Specification.* W3C Recommendation. W3C. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[31] Pablo N. Mendes, Hannes Mühleisen, and Christian Bizer. 2012. Sieve: linked data quality assessment and fusion. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops, Berlin, Germany, March 30, 2012*, Divesh Srivastava and Ismail Ari (Eds.). ACM, 116–123. https://doi.org/10.1145/2320765.2320803

[32] Nebras Nassar. 2020. *Consistency-by-Construction Techniques for Software Models and Model Transformations.* Ph.D. Dissertation. Philipps-Universität Marburg, Germany.

[33] Ivo Oditis, Janis Bicevskis, and Zane Bicevska. 2017. Domain-Specific Characteristics of Data Quality. In *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017, Prague, Czech Republic, September 3-6, 2017 (Annals of Computer Science and Information Systems)*, Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki (Eds.), Vol. 11. 999–1003. https://doi.org/10.15439/2017F279

[34] Paulo Oliveira, Fátima Rodrigues, Pedro Henriques, and Helena Galhardas. 2005. A taxonomy of data quality problems. In *2nd Int. Workshop on Data and Information Quality*. 219–233.

[35] Paulo Oliveira, Fátima Rodrigues, and Pedro Rangel Henriques. 2005. A Formal Definition of Data Quality Problems. In *Proceedings of the 2005 International Conference on Information Quality (MIT ICIQ Conference), Sponsored by Lockheed Martin, MIT, Cambridge, MA, USA, November 10-12, 2006*, Felix Naumann, Michael Gertz, and Stuart E. Madnick (Eds.). MIT. http://mitiq.mit.edu/iciq/iqdownload.aspx?ICIQYear=2005&File=AFormalDefinitionofDQProblems.pdf

[36] Erhard Rahm and Hong Hai Do. 2000. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.* 23, 4 (2000), 3–13. http://sites.computer.org/debull/A00DEC-CD.pdf

[37] Hans-Juergen Rennau. [Online]. Combining graph and tree: writing SHAX, obtaining SHACL, XSD and more. https://www.parsqube.de/publikationen/combining-graph-and-tree-writing-shax-obtaining-shacl-xsd-and-more/

[38] Michael Sperberg-McQueen, Jean Paoli, Tim Bray, François Yergeau, and Eve Maler. 2008. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation. W3C. http://www.w3.org/TR/2008/REC-xml-20081126/.

[39] Josh Spiegel, Jonathan Robie, and Michael Dyck. 2017. *XML Path Language (XPath) 3.1.* W3C Recommendation. W3C. https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

[40] Yufei Sun, Liangli Ma, and Shuang Wang. 2015. A comparative evaluation of string similarity metrics for ontology alignment. *Journal of Information &Computational Science* 12, 3 (2015), 957–964.

[41] TEI Consortium. [Online]. TEI P5: Guidelines for Electronic Text Encoding and Interchange. http://www.tei-c.org/Guidelines/P5/

[42] Amrapali Zaveri, Anisa Rula, Andrea Maurino, Ricardo Pietrobon, Jens Lehmann, and Sören Auer. 2016. Quality assessment for Linked Data: A Survey. *Semantic Web* 7, 1 (2016), 63–93. https://doi.org/10.3233/SW-150175

# A METAMODEL

The complete metamodel for representing patterns is shown in Fig. 6. In the following, we will briefly explain its differences compared to the condensed version depicted in Fig. 4.

The class TrueElement is used to indicate the most inner levels of the nested condition. Hence, it does not contain further conditions.

Instead of considering Elements contained in different Graphs as equivalent if their names are equal, we actually express this equivalency via ElementMappings contained in Morphisms. An ElementMapping is specified by a source and a target Element. Analogously, the class RelationMapping allows defining two Relations that are contained in different graphs as equivalent. A Morphism is specified by a source and a target Graph and is contained in a MorphismContainer, which is either a CountPattern or a QuantifiedCondition.

The interface Adaptable represents items in generic patterns that can be adapted to a specific database technology by being replaced by an instance of a corresponding subclass.

The class BooleanOperator represents operators that return a boolean value. The included list elements holds the Elements whom the BooleanOperator serves as a predicate. This association simplifies the translation of nested operators into queries. Only BooleanOperators that are not an argument of a Comparison can serve as predicates. Such a BooleanOperator is a predicate of those Elements that directly or indirectly (via nested operators) serve as its arguments or that contain the Properties that directly or indirectly serve as its arguments. The elements association is updated automatically when the arguments of an operator are modified.

The class Comparison includes an attribute type which specifies the types of the values that are compared. It determines which casting functions are applied to the values in the data prior to the comparison.

Instead of implementing the generic class OptionParam<T> and passing ComparisonOperator, RelationKind and PropertyKind as the type argument T, we implemented ComparisonOptionParam, RelationOptionParam and PropertyOptionParam as we consider these the only cases in which we need to express parameters whose domain is defined via an enumeration of allowed values.

Fig. 6 shows more subclasses of ParameterValue than Fig. 4. They allow expressing boolean values, lists of string values as well as some of the XML Schema data types used in XQuery (i.e. date, time and dateTime).

## A.1 Constraints

Besides the *multiplicity constraints* depicted in Fig. 6, our metamodel includes further constraints. They are implemented in the isValidLocal method of the corresponding PatternElement. Since some constraints may only apply to patterns of a certain level of abstraction (i.e. generic, abstract or concrete), the method has a parameter of type AbstractionLevel. In the following, we will specify the constraints in natural language grouped by package.

### *A.1.1 Constraints of the Package* patternstructure.

- Each Morphism must reference the preceding Graph in the nested condition via the association source and the Graph of its MorphismContainer via the association target.
- The source Element of an ElementMapping must be contained in the source Graph of the Morphism that contains the ElementMapping. The target Element must be contained in the target Graph of this Morphism. Analogously, the source and target Relation of a RelationMapping must be contained in the source or target Graph of the containing Morphism.
- Two Mappings contained in the same Morphism must not have the same source.

### *A.1.2 Constraints of the Package* graphstructure.

- The list of returnElements of a Graph must not be empty.
- Each Element contained in the list of returnElements of a Graph must be contained in this Graph.
- The returnElements of all Graphs directly contained in a Pattern or directly contained in any of its QuantifiedConditions must be equivalent (specified via ElementMappings).
- For generic patterns: a graph must not contain items of the package adaptionxml.
- For XML-adapted abstract and concrete patterns only: Each Graph must contain exactly one XmlRoot.
- For XML-adapted abstract and concrete patterns: a graph must not contain instances of Element, Relation or Property.
- The Elements and Relations contained in the graph of a CompletePattern must not be the target of a Mapping.
- For each RelationMapping there must be two ElementMappings which indicate that the source Elements of both Relations are equivalent and that the target Elements of both Relations are equivalent.

### *A.1.3 Constraints of the Package* operators.

- For each BooleanOperator that does not serve as an argument of a comparison the set of Elements that directly or indirectly serve as its arguments must be equal to the set elements.
- For each BooleanOperator that serves as an argument of a comparison its association elements must be empty.
- The return types of both arguments of a Comparison must be equal to the type of the Comparison. For a concrete pattern this type must not be UNSPECIFIED.
- A Comparison that has two Elements as arguments may be specified by EQUAL or NOTEQUAL only.
- Each ComparisonOperator must be free of cycles: it must not have another ComparisonOperator as an argument that it also directly or indirectly serves as an argument.
- Each Operator referenced within a Graph must be contained in this graph's operatorList. Each Operator contained in a graph's operatorList must be referenced from within this Graph.

### *A.1.4 Constraints of the Package* parameters.

- Each Parameter referenced within a CompletePattern must be contained in this pattern's parameterList. Each Parameter contained in a pattern's parameterList must be referenced from within this CompletePattern.
- For a concrete pattern each ParameterValue must be specified. Thus, its value (or values) attribute must not be null (or empty). For Date, Time and DateTime the value must satisfy a specific regular expression given in the specification of the XML Schema language [5].

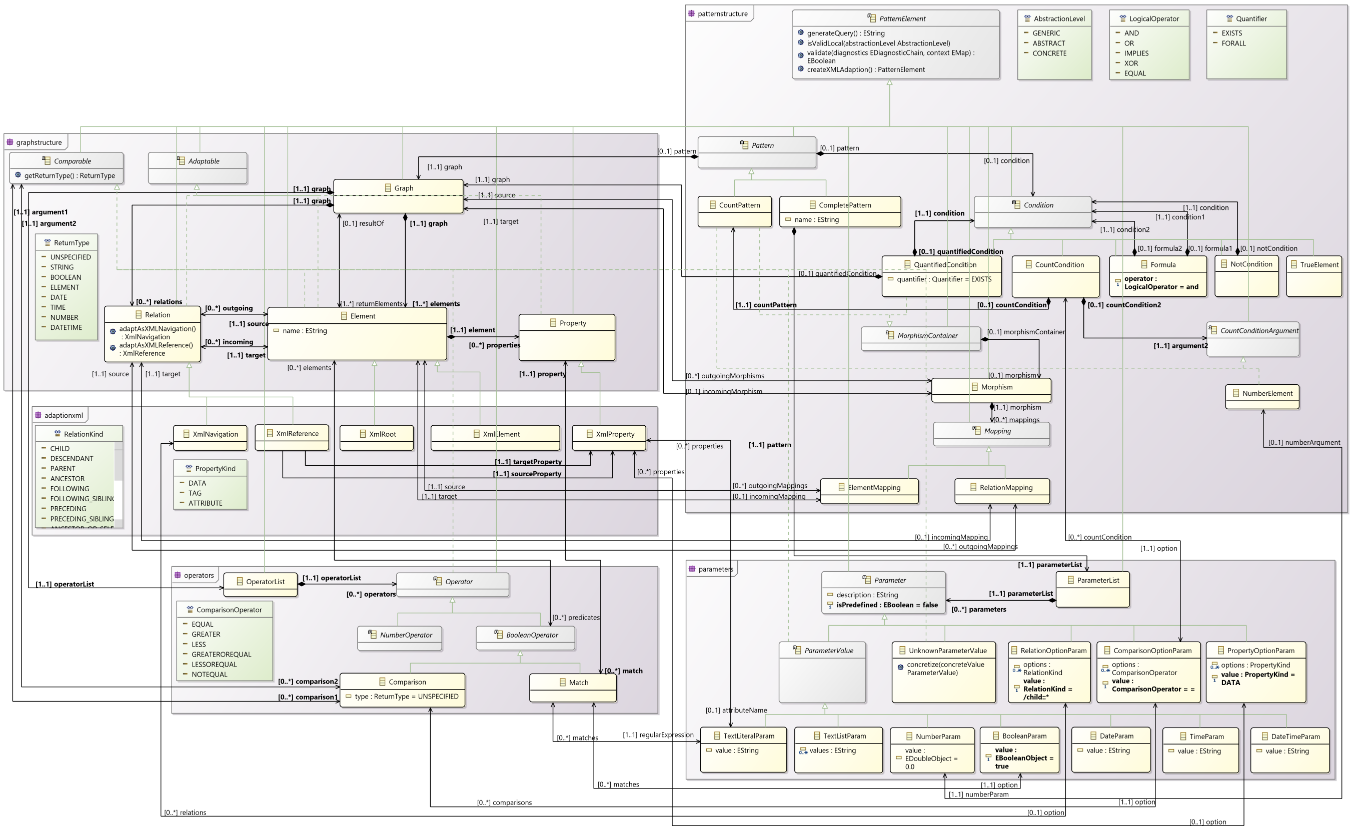

**Figure 6: Complete metamodel**

- A concrete pattern must not contain an `UnknownParameterValue`.
- For a concrete pattern each `ComparisonOptionParam`, `RelationOptionParam` and `PropertyOptionParam` must be specified. The `option` attribute must give at least one choice. Further, the attribute `value` must not be null and it must be one of the values listed in the `option` attribute.

### A.1.5 *Constraints of the Package* `adaptionxml`.

- Each `XmlElement` must have exactly one incoming `XmlNavigation`.
- An `XmlRoot` must not have any incoming `Relations`.
- An `XmlRoot` must not have any outgoing `XmlReferences`.
- The association `predicates` of an `XmlRoot` must be empty.
- The `option` of an `XmlNavigtion` must be non-null if and only if it is not the target of a `RelationMapping`.
- An `XmlNavigation` may only be mapped to `Relations` of type `XmlNavigation` and an `XmlReference` may only be mapped to `Relations` of type `XmlReference`.
- The `sourceProperty` of an `XmlReference` must be contained in its `source Element` and the `targetProperty` of an `XmlReference` must be contained in its `target Element`. These are automatically generated during the adaptation.
- If an `XmlProperty` is specified to be of type `ATTRIBUTE`, its `attributeName` of type `TextLiteral` must not be null. In the concrete pattern, its `TextLiteral` must contain a non-empty string as `value` attribute.

## B FURTHER EXAMPLE

To get a better understanding of our metamodel, we will present one further example pattern in the following. First we modify the running example. In the version depicted in Listing 5 `building` elements may contain a `creator` element containing a reference to its associated architect.

Note, that in this example we have no `architect` with `ID 4`, however such a data record is referenced by the `building` with `ID 2`. Hence, this is a referential integrity violation. Such problems usually do not arise during data creation as this is controlled by database management systems. Instead, they often stem from deletions or transfer of parts of the database.

The *abstract pattern* `REFINT` shown in Fig. 7 detects such referential integrity violations. The right part of this pattern specifies the condition that determines whether referential integrity has been violated. The return element has to contain another element (called Element 1, identified by Property1 and Value1). Its PropertyA1 refers to some PropertyA2 of an Element 2 (identified by Property2 and Value2) which is not contained in the root element as indicated and thus does not exist.

Note that Fig. 7 shows three graphs which are included into each other from left to right. Pattern elements are bound from left

**Listing 5: Running example: data describing paintings, buildings and artists which includes 3 of the quality problems listed in Table 1.**

```
<data>
	<building id="1">
		<name>Empire State Building</name>
		<city>New York City</city>
		<country>USA</country>
		<creator ref="3"/>
	</building>
	<building id="2">
		<name>Chrysler Building</name>
		<city>New York City</city>
		<country>unknown</country>
		<creator ref="4"/>
	</building>
	<architect id="3">
		<name>William F. Lamb</name>
		<birthyear>1883</birthyear>
		<birthyear>1884</birthyear>
	</architect>
</data>
```

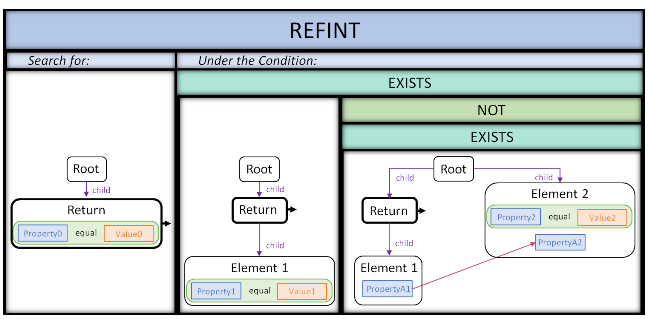


**Figure 7: Abstract pattern REFINT for detecting referential integrity violations; XML-adaption**

to right. This allows expressing relations and conditions between elements already existing in a previous graph (to the left) and new elements that are added to the pattern in this graph.

To detect the referential integrity violation in the example data in Listing 5, the parameters need to be specified as shown in Listing 6.

**Listing 6: Example concretisation of the pattern REFINT**

```
Property0 = name, Value0 = "building"
Property1 = name, Value1 = "creator"
Property2 = name, Value2 = "architect"
PropertyA1 = attribute "ref", PropertyA2 = attribute "id"
```

*The concrete REFINT pattern detects all* building *elements whose* creator *element contains a reference to a non-existent* architect *element.* When applied to the example data depicted in Listing 5, the pattern returns the building with ID 2.

A reviewer pointed out that this pattern detects also cases in which Element 1 does not have the PropertyA1, which is actually not a referential integrity violation. We will address this issue in near future. The issue, however, does not affect our evaluation as in the chosen database Element 1 exists only if it also includes a reference to an element Element 2, thus has PropertyA1.

## B.1 An Abstract Pattern as Instance Model

Fig. 8 shows the abstract pattern REFINT (visualised in Fig. 7) as an instance model of our metamodel as depicted in Fig. 4. The instance model represents the internal structure of the pattern, which is mostly hidden behind the graphical representation as shown in Fig. 7. The red objects are the surrounding Pattern and the Graphs, which are shown as square white blocks in the graphical representation. The structure around the blocks represents conditions; corresponding objects are represented in light blue. Each Graph contains Elements, which are the nodes of the graph. When adapted to XML these are represented as XmlElements. The element names in Fig. 8 match with those in Fig. 7. Elements that are contained in different graphs but have the same name correspond to each other.

The Elements are connected by Relations, which in the XML adaption are represented as XmlNavigation and XmlReference contained in Graph. It has to be assured, that the Relations of elements with the same name occurring in different Graphs are equivalent. To guarantee that, only the first XmlNavigation has an OptionParam attached. All corresponding XmlNavigations in other graphs (identified via equivalent element names) have implicitly the same specification.

The Parameters are shown in orange at the bottom of the model. The first group represents 8 TextLiteralParams. 3 of them are used in comparisons, while the other 5 are used in an XmlProperty each; they are required only in case the option *PropKind = attribute* is chosen. The other groups consist of 12 OptionParams: 3 of them for Relations, 5 specify Propertys and the other 4 specify Comparisons. Each of them has a variable value, which represents the chosen value. In this example, the values of OptionParam<RelKind> and OptionParam<CompOp> are predefined. The values of OptionParam<PropKind> and TextLiteralParam have to be specified during the concretisation. Meanwhile, they are represented as null.

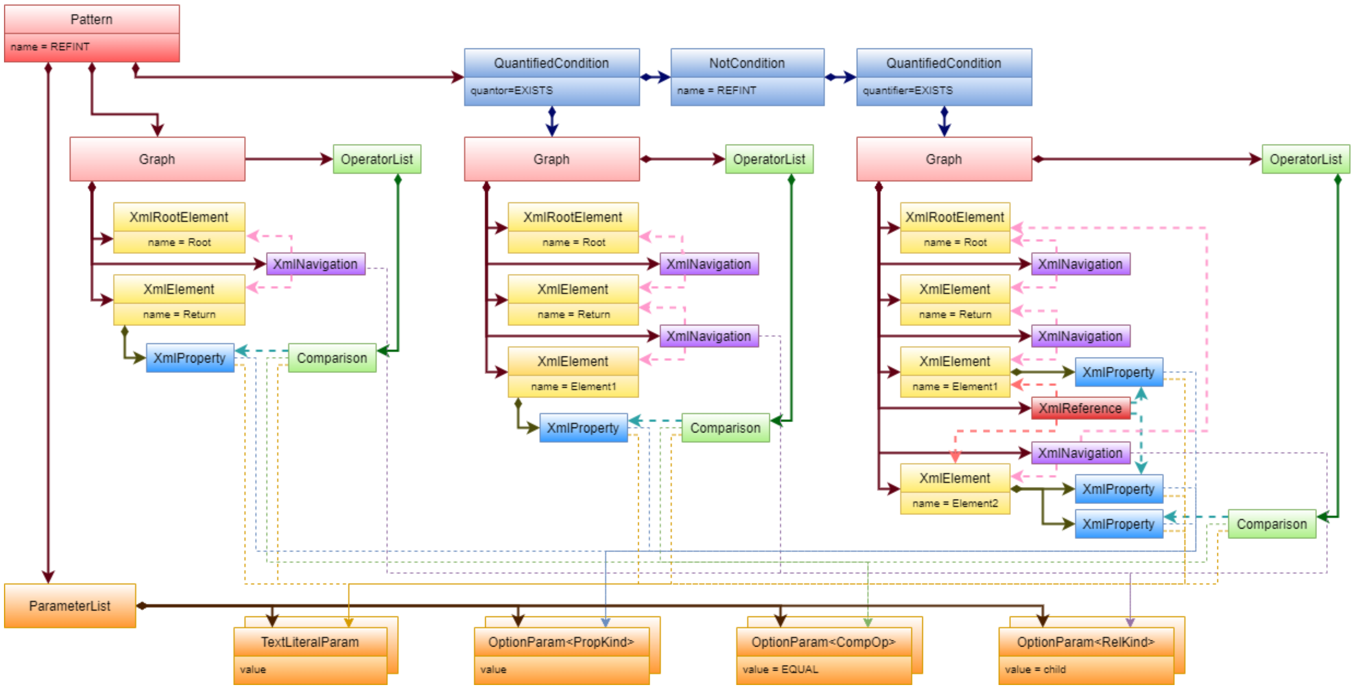


**Figure 8: Abstract pattern REFINT as instance model [solid arrows: containment, dashed arrows: non-containment]**

## C GENERATED QUERIES

The following listings show the XQuery expressions generated by our tool for the concrete example pattern CARD presented in Section 3.2 and REFINT presented in Section B of the appendix.

**Listing 7: XQuery for the concrete CARD pattern presented in Section 3.2**

```
for $var1 in /child::*[./name()="artist"]
where count(
 for $var2 in /child::*[./name()="name"]
 where true()
 return $var2
) > 1.0
return $var1
```

**Listing 8: XQuery for the concrete REFINT pattern presented in Section B**

```
for $var1 in /child::*[./name()="building"]
where some $var2 in $var1/child::*[./name()="creator"]
 satisfies not(
  some $var3 in /child::*[./name()="artist"]
     [$var2/@ref=./@id]
    satisfies true()
 )
return $var25
```

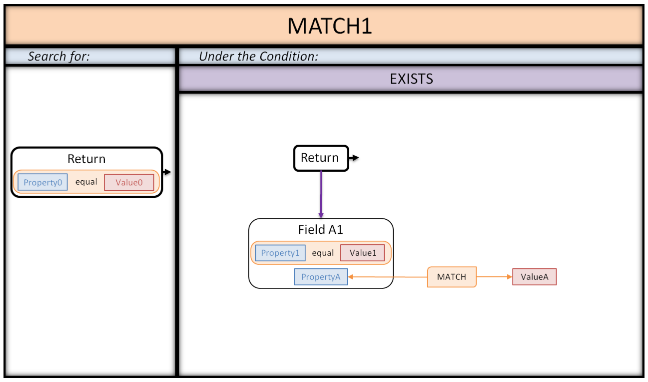


Figure 9: Generic pattern MATCH1

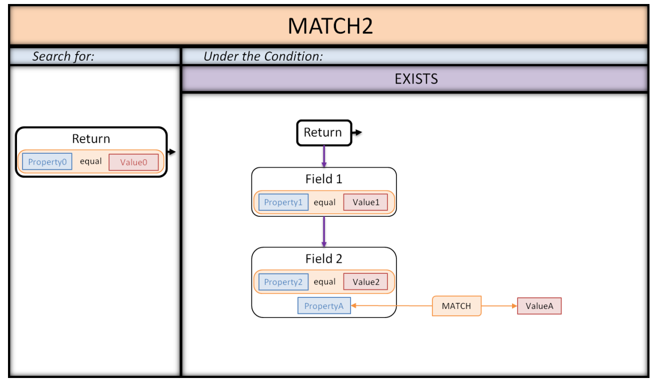


Figure 11: Generic pattern MATCH2

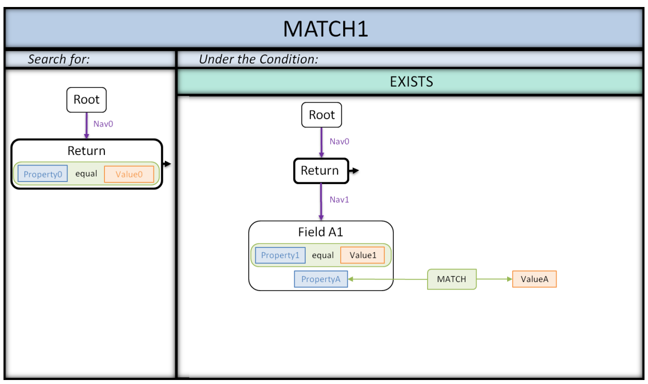


Figure 10: Abstract pattern MATCH1

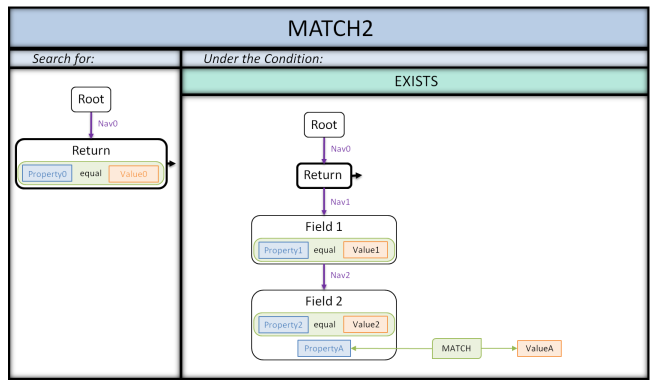


Figure 12: Abstract pattern MATCH2

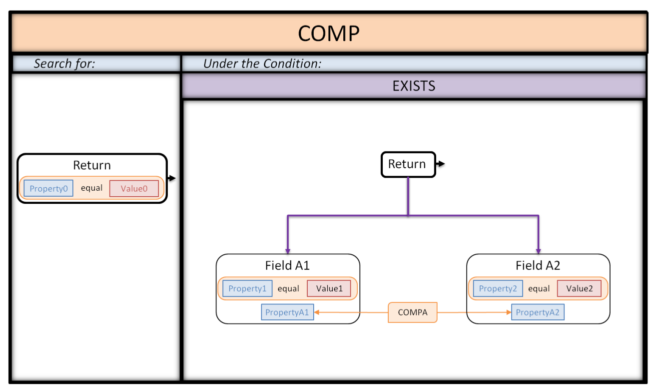


Figure 13: Generic pattern COMP

# D GENERIC AND ABSTRACT PATTERNS

In the following, we will present the generic and XML-adapted abstract patterns that were used for the evaluation presented in Section 5. Patterns that have a similar purpose are grouped together. For each pattern we will briefly explain its purpose, present the diagram of the generic pattern and finally present the diagram of the XML-adapted abstract version of that pattern. Concretisations of these patterns will be presented in Section E.

The patterns are depicted via diagrams that are structured as those presented in Section 3.2. Note that there is the possibility to use the same parameter multiple times. For example, if we want to determine that two elements of a pattern represent the same XML elements we can use the same properties, values and XPath axes for both pattern elements. In the presented patterns, the concrete operator of a comparison is often predefined in the generic pattern as "equal". In this case the comparison serves for identifying certain elements. Some patterns contain further comparisons that are not predefined. They allow expressing conditions to the values of the selected elements.

## D.1 MATCH

The MATCH patterns (Figures 9-12) check a value in the data (PropertyA) against a given regular expression (ValueA).

## D.2 COMP

The COMP pattern (Figures 13, 14) compares two values that are indirectly related to one record.

## D.3 COMPVAL

The COMPVAL patterns (Figures 15-18) compare a value in the data with a literal value via a comparison operator (COMP2 and COMP3, respectively).

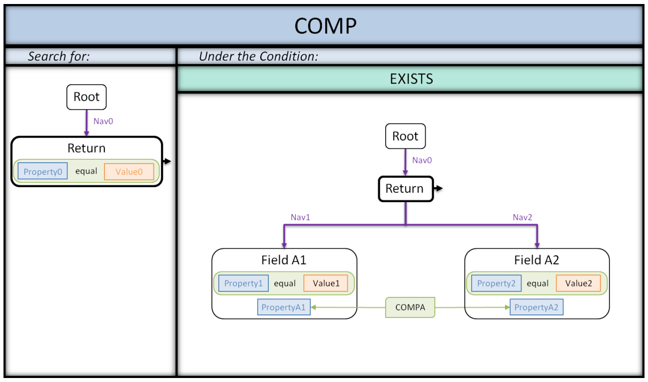


Figure 14: Abstract pattern COMP

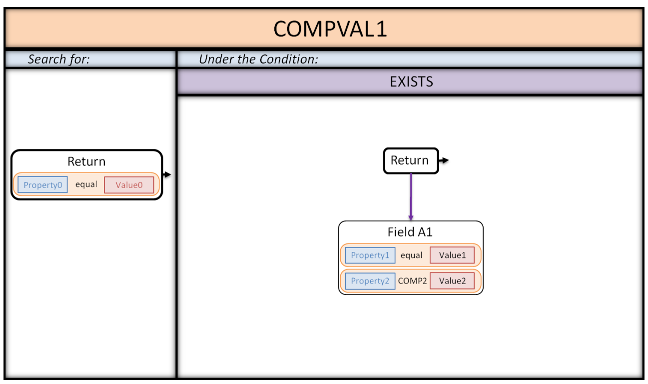


Figure 15: Generic pattern COMPVAL1

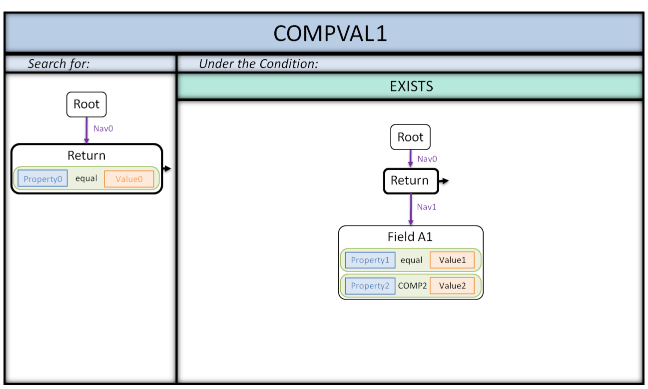


Figure 16: Abstract pattern COMPVAL1

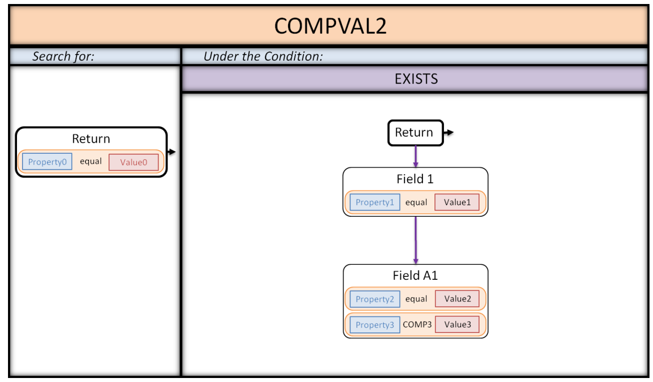


Figure 17: Generic pattern COMPVAL2

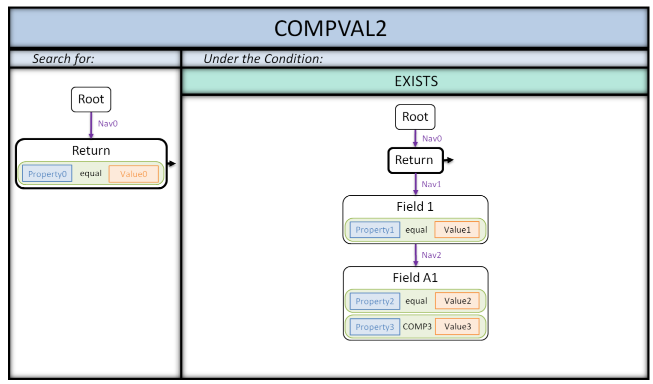


Figure 18: Abstract pattern COMPVAL2

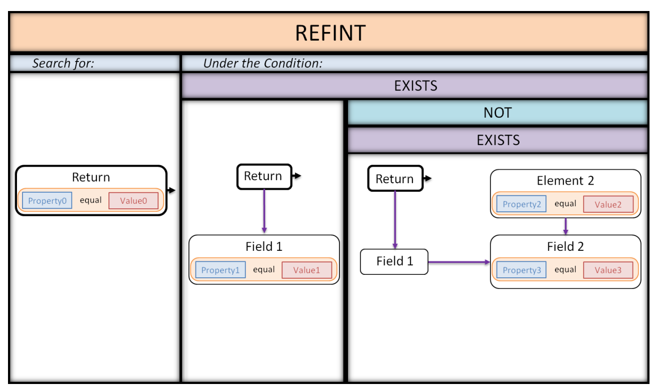


Figure 19: Generic pattern REFINT

## D.4 REFINT

The REFINT pattern (Figures 19, 20) detects referential integrity violations. A reviewer pointed out that this pattern detects also cases in which the element called Field 1 does not have the PropertyA, which is actually not a referential integrity violation. We will address this issue in near future. The issue, however, does not affect our evaluation as in the chosen database the element Field 1 exists only if it also includes a reference to an element Field 2, thus has PropertyA.

## D.5 CARD

The CARD patterns (Figures 21-24) check whether a certain structure occurs more than once in the data. The first CARD pattern is explained in detail in Section 3.2.

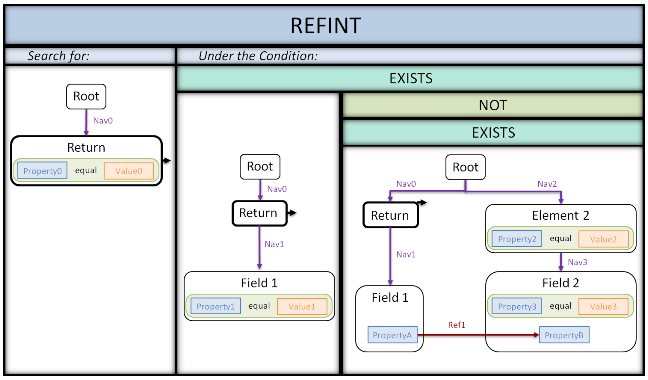


Figure 20: Abstract pattern REFINT

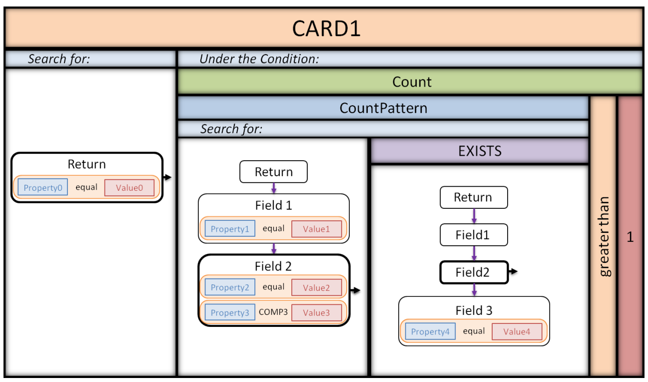


Figure 21: Generic pattern CARD1

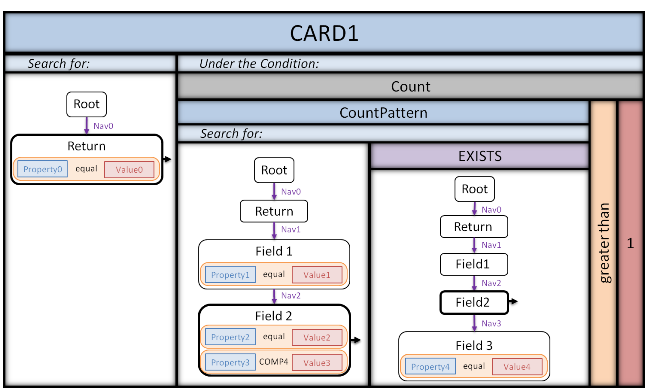


Figure 22: Abstract pattern CARD1

## D.6 FUNC

The FUNC patterns (Figures 25, 28) detect violations of functional dependencies. The patterns check whether the values of a field in two data records do not match (Field B1 and B2), even though another (generally more specific) field (Field A1 and A2) has the same value in both records. The first FUNC pattern is explained in detail in Section 3.2.

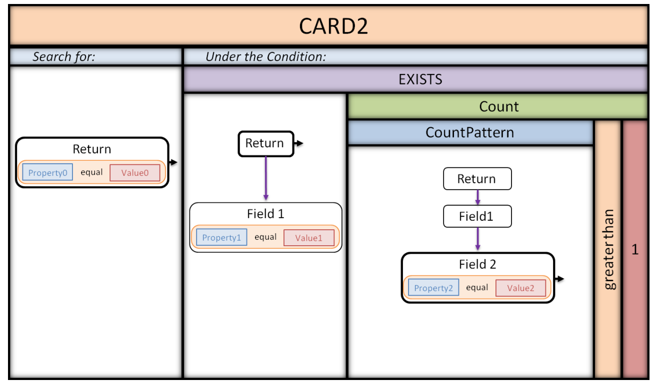


Figure 23: Generic pattern CARD2

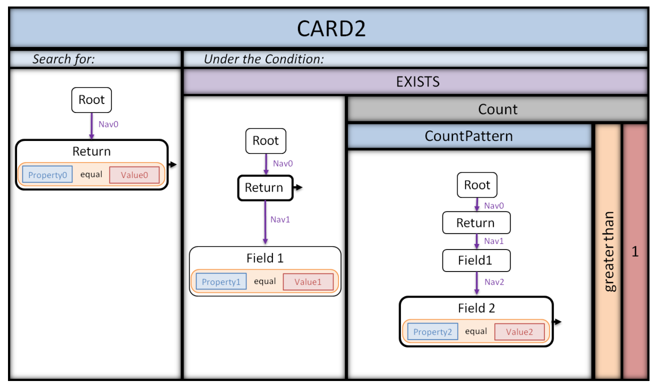


Figure 24: Abstract pattern CARD2

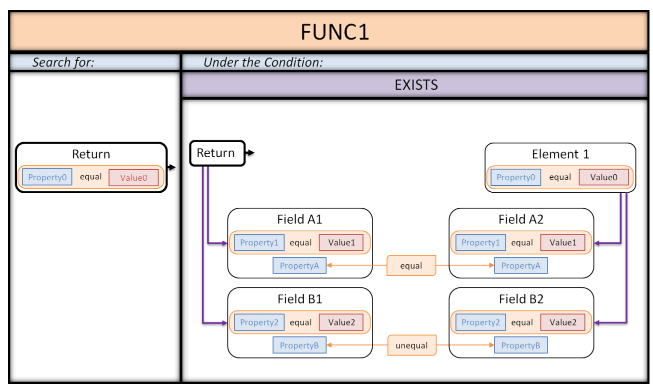


Figure 25: Generic pattern FUNC1

## D.7 UNIQUE

The UNIQUE patterns detect unique value violations across records (UNIQUE, Figures 29, 30) or within a record (UNIQUE2, Figures 31, 32).

## D.8 MAND

The MAND patterns (Figures 33-36) detect missing mandatory attributes. The patterns match if a specific structure does not exist

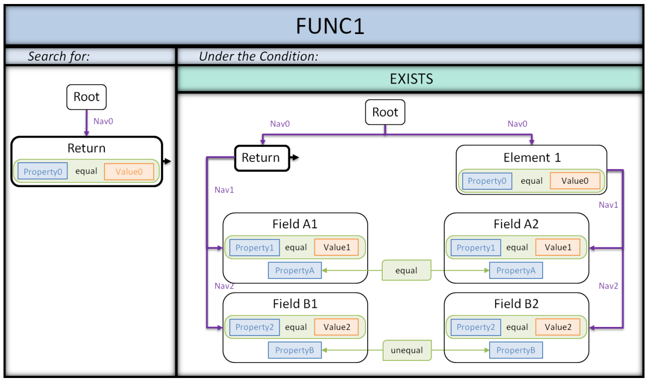


Figure 26: Abstract pattern FUNC1

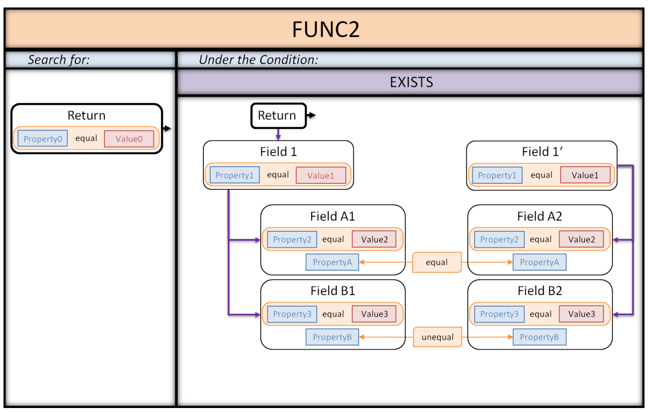


Figure 27: Generic pattern FUNC2

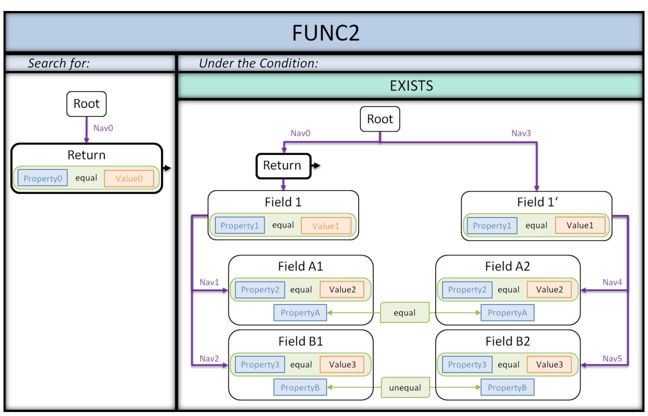


Figure 28: Abstract pattern FUNC2

or if it exists but a specific included attribute is equal to one of several given dummy values.

## D.9 MANDSTRUC

The MANDSTRUC patterns detect missing mandatory structures in the data. MANDSTRUC1 (Figures 37, 38) detects data records,

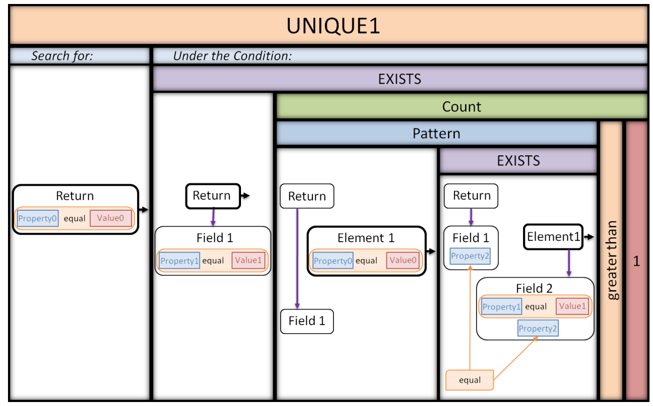


Figure 29: Generic pattern UNIQUE1

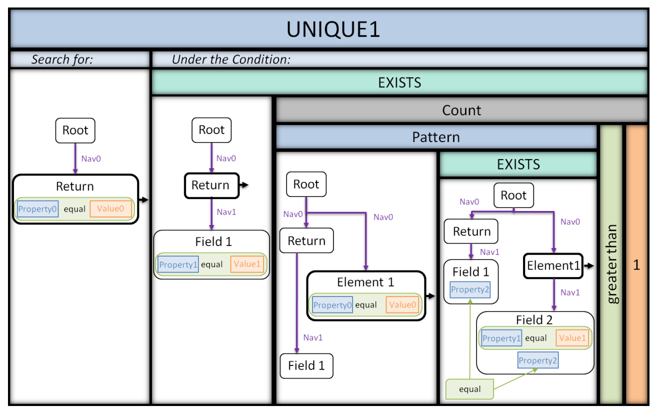


Figure 30: Abstract pattern UNIQUE1

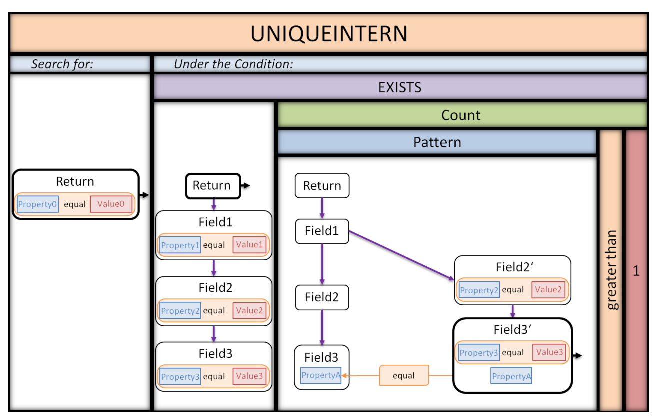


Figure 31: Generic pattern UNIQUE2

where Field 1 does not exist or its related Field 2 does not exist. MANDSTRUC2 (Figures 39, 40) tests the existence of Field 2 under the precondition that the related Field 1 exists in the record.

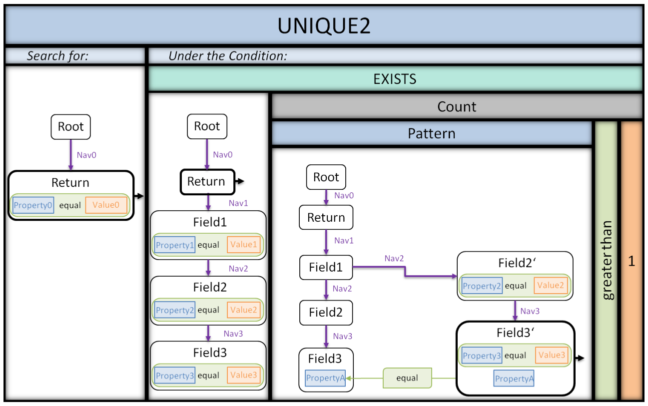


Figure 32: Abstract pattern UNIQUE2

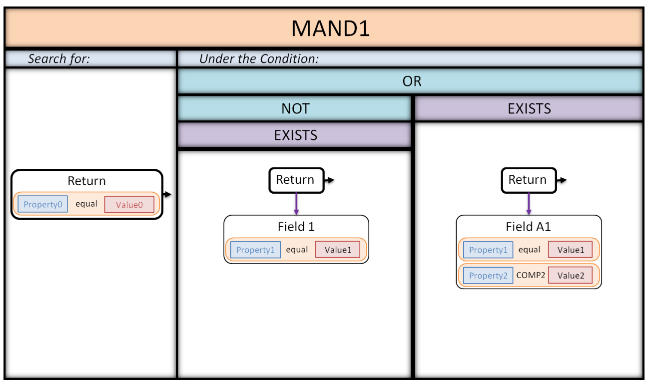


Figure 33: Generic pattern MAND1

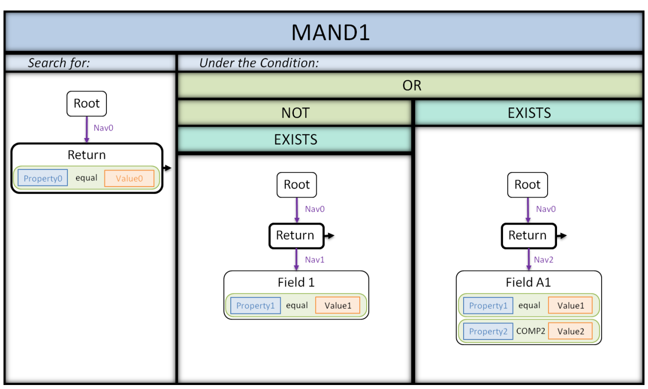


Figure 34: Abstract pattern MAND1

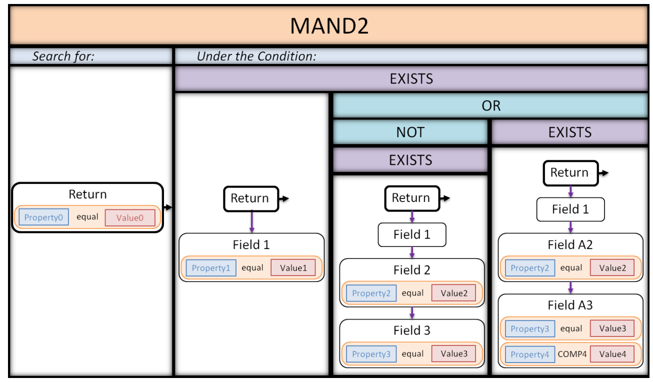


Figure 35: Generic pattern MAND2

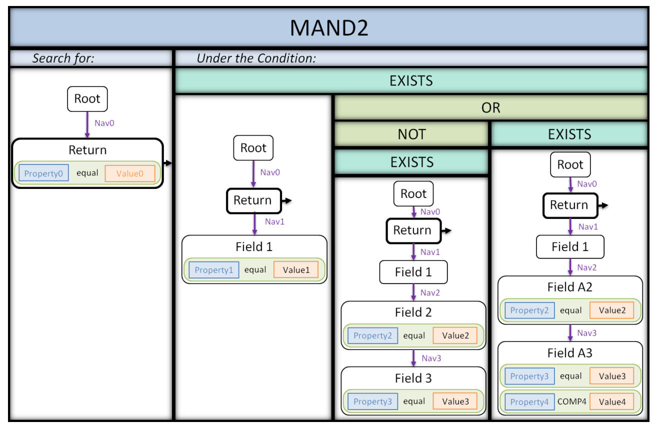


Figure 36: Abstract pattern MAND2

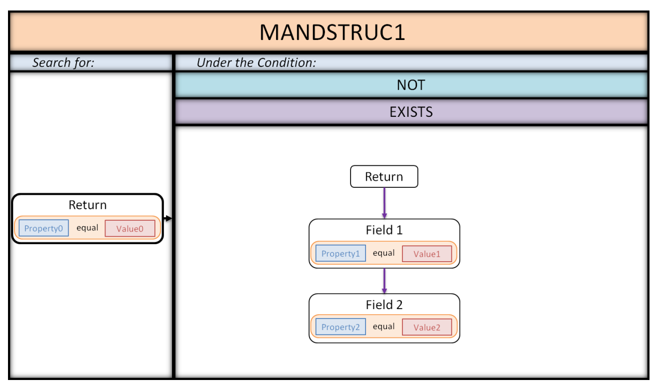


Figure 37: Generic pattern MANDSTRUC1

## D.10 CONTREL

The CONTREL pattern (Figures 41, 42) detects contradictory relationships of two records, more precisely violations of a symmetry constraint. If Field 2 contains a reference to another record which contains a reference to the first record, the values of a specific field must satisfy a given comparison relation. Otherwise the relationships are contradictory.

## D.11 EXDUP

The EXDUP pattern (Figures 43, 44) detects exact duplicate records via a direct comparison between elements.

## D.12 APPDUP

The APPDUP pattern (Figures 45, 46) detects approximate duplicate records by comparing three distinguishing attributes.

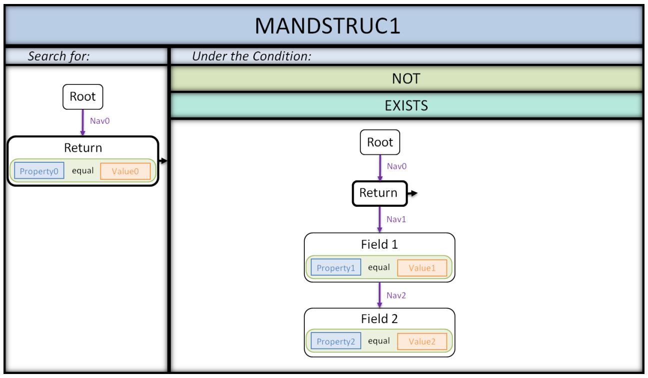


Figure 38: Abstract pattern MANDSTRUC1

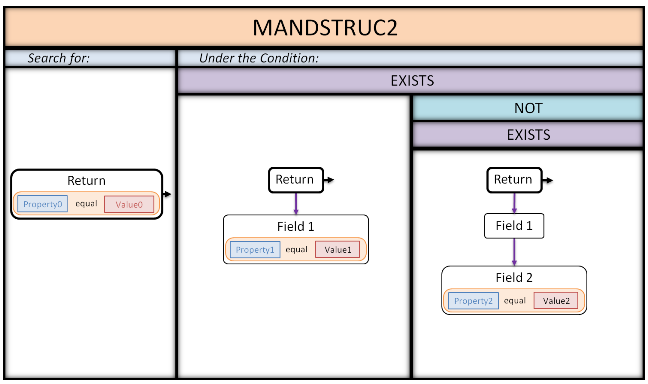


Figure 39: Generic pattern MANDSTRUC2

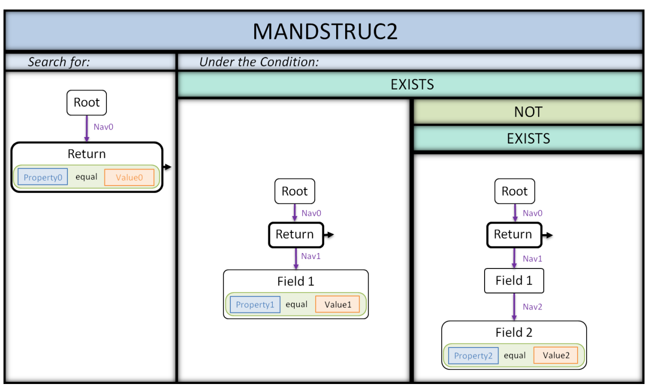


Figure 40: Abstract pattern MANDSTRUC2

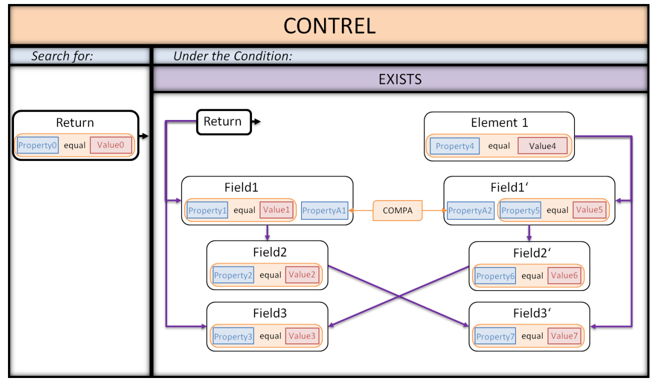


Figure 41: Generic pattern CONTREL

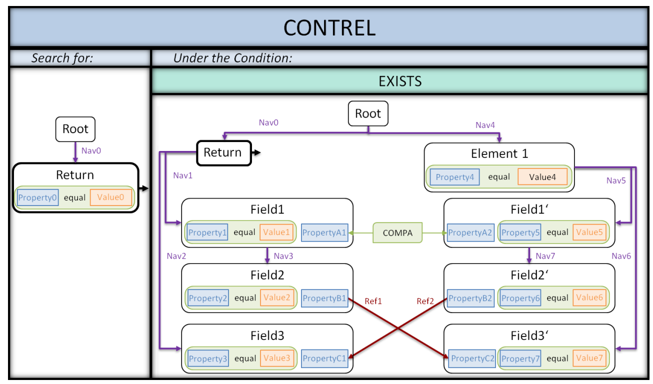


Figure 42: Abstract pattern CONTREL

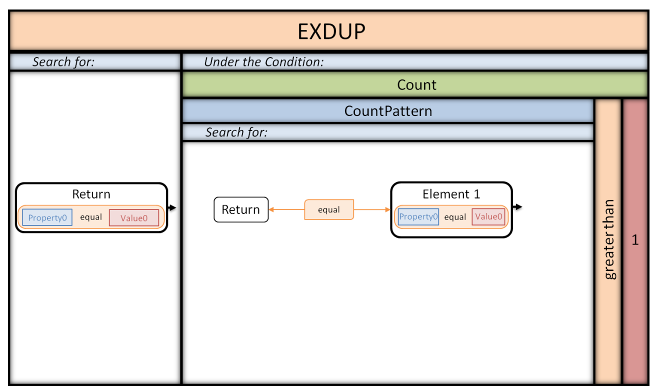


Figure 43: Generic pattern EXDUP

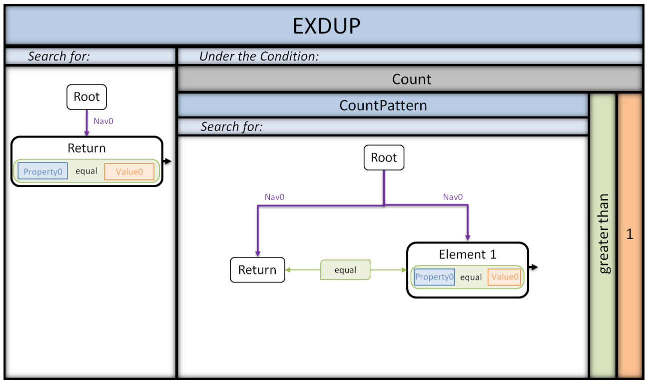


**Figure 44: Abstract pattern EXDUP**

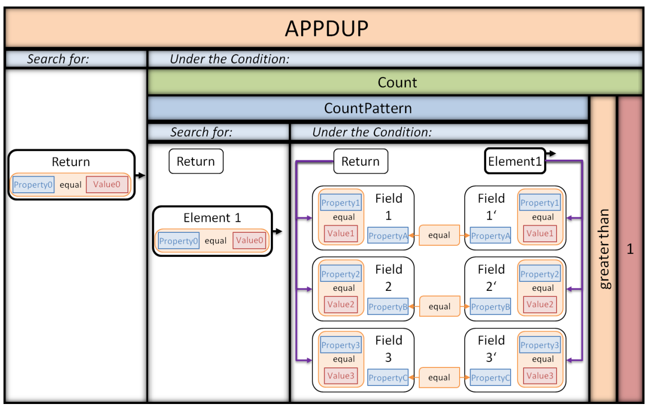


**Figure 45: Generic pattern APPDUP**

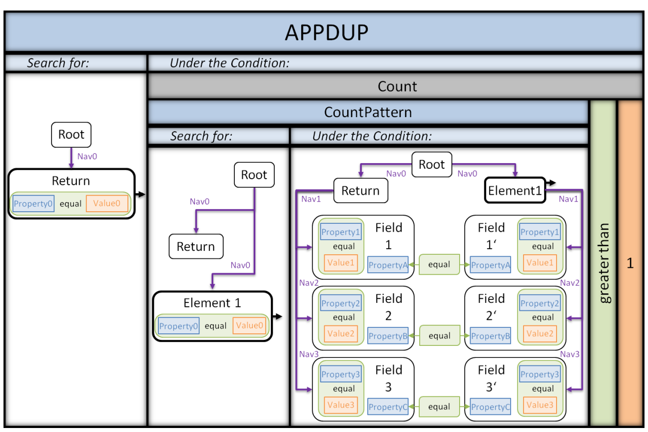


**Figure 46: Abstract pattern APPDUP**

# E REPRESENTATIVE QUALITY PROBLEMS

In the following, we will present the concrete quality problem instances for the MIDAS and LIDO data set that were covered by patterns in the course of the evaluation. In some cases, we checked the LIDO data for the same concrete instance of the quality problem as the MIDAS data. For each problem we will give a brief description, name the corresponding abstract pattern depicted in Section D, give the runtime and list the parameter values that were necessary to concretise the corresponding XML-adapted abstract pattern for the MIDAS and LIDO format, respectively. For each concretisation we specify properties, comparison operators, parameter values and relations. If related properties, comparison operators and values shown in the generic diagram are not specified in the concretisation, this comparison is ignored.

## E.1 Illegal Values

### *E.1.1 Wrong Datatype.*

*MIDAS.* An XML element indicating the relation between an object and an artist contains an element of a wrong type. The contained elements should describe the artist, but that is not the case here.
Runtime: 6653 ms

**Listing 9: Concretisation of the pattern MATCH2**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "ob30"
Nav2 = child, PropertyA = attribute "Value", ValueA = "^[12456789]"
```

*LIDO.* A measurement value which should be given as a whole number or decimal fraction contains letters.
Runtime: 7206 ms

**Listing 10: Concretisation of the pattern MATCH1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child7, Property1 = name, Value1 = "lido:measurementValue"
PropertyA = data, ValueA = "[a-zA-ZüöäÜÖÄ]"
```

### *E.1.2 Domain Violation.*

*MIDAS.* A set violation regarding an artist's gender.
Runtime: 770 ms

**Listing 11: Concretisation of the pattern COMPVAL1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "kue"
Nav1 = child, Property1 = attribute "Type", Value1 = "3140"
Property2 = attribute "Value", COMP2 = unequal, Value2 =
      ("m","f","unbekannt","m?","f?","?")
```

*LIDO.* See above.
Runtime: 8021 ms

**Listing 12: Concretisation of the pattern COMPVAL1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child8, Property1 = name, Value1 = "lido:genderActor"
Property2 = data, COMP2 = unequal, Value2 =
      ("male","männlich","weiblich","female","unknown","not
      applicable")
```

## E.2 Missing Data

### *E.2.1 Missing Values.*

*MIDAS.* The profession of an artist is not given.
Runtime: 1217 ms

**Listing 13: Concretisation of the pattern MAND1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "kue"
Nav1 = child, Property1 = attribute "Type", Value1 = "3162"
Property2 = attribute "Value", COMP2 = equal, Value2 = ""
```

*LIDO.* An actor without a name specification.
Runtime: 8296 ms

**Listing 14: Concretisation of the pattern MAND2**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child7, Property1 = name, Value1 = "lido:actor"
Nav2 = child, Property2 = name, Value2 = "lido:nameActorSet"
Nav3 = child, Property3 = name, Value3 = "lido:appellationValue"
Property4 = data, COMP2 = equal, Value4 = ""
```

### *E.2.2 Missing References.*

*MIDAS.* An object for which no artist is specified.
Runtime: 6635 ms

**Listing 15: Concretisation of the pattern MANDSTRUC1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "ob30"
Nav2 = child, Property2 = attribute "Type", Value2 = "3100"
```

*LIDO.* The role of an actor is not supplemented by a reference to a published controlled vocabulary.
Runtime: 6356 ms

**Listing 16: Concretisation of the pattern MANDSTRUC2**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child7, Property1 = name, Value1 = "lido:roleActor"
Nav2 = child, Property2 = name, Value2 = "lido:conceptID"
```

*E.2.3 Missing Records.* As explained in Section 5.2 this problem could not be covered.

### *E.2.4 Dummy Values.*

*MIDAS.* The birthdate of an artist is one of: "x", "y", "?", "unbekannt" (German for "unknown").
Runtime: 926 ms

**Listing 17: Concretisation of the pattern COMPVAL1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "kue"
Nav1 = child, Property1 = attribute "Type", Value1 = "3270"
Property2 = attribute "Value", COMP2 = unequal, Value2 =
      ("x","y","?","unbekannt")
```

*LIDO.* An appellation is stated to be one of: "unbekannt" (German for "unknown"), empty string, "?", "x", "unknown".
Runtime: 9194 ms

**Listing 18: Concretisation of the pattern COMPVAL1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child5, Property1 = name, Value1 = "lido:appellationValue"
Property2 = data, COMP2 = unequal, Value2 =
      ("unbekannt","","?","x","unknown")
```

## E.3 Referential Integrity Violation

*MIDAS.* An object record containing a reference to a non-existent atelier record.

Runtime: 197244 ms

**Listing 19: Concretisation of the pattern REFINT**

```
Nav0 = child3, Property0 = attribute "Type", COMP0 = equal, Value0
      = "obj"
Nav1 = child2, Property1 = attribute "Type", COMP1 = equal, Value1
      = "3600"
Nav2 = child3, Property2 = attribute "Type", COMP2 = equal, Value2
      = "wer"
Nav3 = child, Property3 = attribute "Type", COMP3 = equal, Value3 =
      "3600"
PropertyA = attribute "Value", PropertyB = attribute "Value"
```

*LIDO.* As explained in Section 5.1 this problem variant could not be covered for the LIDO data.

## E.4 Unique Value Violation

*MIDAS.* The atelier name is not unique even though it is used as an identifier.

Runtime: 90836 ms

**Listing 20: Concretisation of the pattern UNIQUE1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "wer"
Nav1 = child, Property1 = attribute "Type", Value1 = "3600"
Property2 = attribute "Value"
```

*LIDO.* A non-unique record ID.

Runtime: estimated 1.8 weeks (not finished)

**Listing 21: Concretisation of the pattern UNIQUE1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child, Property1 = name, Value1 = "lido:lidoRecID"
Property2 = data
```

## E.5 Violation of a Functional Dependency

*MIDAS.* Two atelier records with equal atelier names but different active years. The atelier name should be unique and thus determine the active years.

Runtime: 64299 ms

**Listing 22: Concretisation of the pattern FUNC1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "wer"
Nav1 = child, Property1 = attribute "Type", Value1 = "3600"
Nav2 = child, Property2 = attribute "Type", Value2 = "3680"
PropertyA = attribute "Value"
PropertyB = attribute "Value"
```

*LIDO.* The functional dependency between a concept ID linking to a published controlled vocabulary and the corresponding name of the concept is violated for the description of a material or technique.

Runtime: 21474836 ms

**Listing 23: Concretisation of the pattern FUNC2**

```
Nav0 = child2, Property0 = name, COMP0 = equal, Value0 = "lido:lido"
Nav1 = child7, Property1 = name, COMP1 = equal, Value1 =
      "lido:termMaterialsTech"
Nav2 = child, Property2 = name, COMP2 = equal, Value2 =
      "lido:conceptID"
Nav3 = child, Property3 = name, COMP3 = equal, Value3 = "lido:term"
PropertyA = data
PropertyB = data
```

## E.6 Contradictory Relationships

*MIDAS.* If an artist record references an atelier record and the atelier record references the artist record, both specified relation types must be equal, which is not the case here.

Runtime: 18957 ms

**Listing 24: Concretisation of the pattern CONTREL**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "kue"
Nav1 = child, Property1 = attribute "Type", Value1 = "ku35"
Nav2 = child, Property2 = attribute "Type", Value2 = "3600"
Nav3 = child, Property3 = attribute "Type", Value3 = "3100"
Nav4 = child3, Property4 = attribute "Type", Value4 = "wer"
Nav5 = child, Property5 = attribute "Type", Value5 = "we30"
Nav6 = child, Property6 = attribute "Type", Value6 = "3100"
Nav7 = child, Property7 = attribute "Type", Value7 = "3600"
PropertyA1 = attribute "Value", COMPA = unequal, PropertyA2 =
      attribute "Value"
PropertyB1 = attribute "Value", PropertyC2 = attribute "Value"
PropertyC1 = attribute "Value", PropertyB2 = attribute "Value"
```

*LIDO.* As explained in Section 5.1 this problem variant could not be covered for the LIDO data.

## E.7 Imprecise Data

### *E.7.1 Alternative Possible Values.*

*MIDAS.* Multiple possible artists are listed for an artwork.

Runtime: 9813 ms

**Listing 25: Concretisation of the pattern CARD1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = descendant-or-self, Property1 = name, Value1 = "h1:Block"
Nav2 = child, Property2 = attribute "Type", Value2 = "ob30"
Nav3 = child, Property3 = attribute "Value", COMP3 = equal, Value3
      = "Herstellung"
Property4 = attribute "Type", Value4 = "ob30rl"
```

*LIDO.* The place of an event is marked as "alternative".

Runtime: 5351 ms

**Listing 26: Concretisation of the pattern COMPVAL1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child5, Property1 = name, Value1 = "lido:eventPlace"
Property2 = attribute "lido:type", COMP2 = equal, Value2 =
      "alternative"
```

### *E.7.2 Imprecise Numerical Values.*

*MIDAS.* An interval is given as the date of birth of an artist.

Runtime: 965 ms

**Listing 27: Concretisation of the pattern MATCH1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "kue"
Nav1 = child, Property1 = attribute "Type", Value1 = "3270"
PropertyA = attribute "Value", ValueA = "[0-9]/[0-9]"
```

*LIDO.* The field for specifying the earliest possible date of something contains only the specification of a year.
Runtime: 4871 ms

**Listing 28: Concretisation of the pattern MATCH1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child7, Property1 = name, Value1 = "lido:earliestDate"
PropertyA = data, ValueA = "^[0-9]{4}$"
```

#### *E.7.3 Abstract Terms.*

*MIDAS.* The type of the described artwork is specified as "Objekt" (German for "object").
Runtime: 8296 ms

**Listing 29: Concretisation of the pattern COMPVAL1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "5230"
Property2 = attribute "Value", COMP2 = equal, Value2 = "Objekt"
```

*LIDO.* See above.
Runtime: 5630 ms

**Listing 30: Concretisation of the pattern COMPVAL2**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child4, Property1 = name, Value1 = "lido:objectWorkType"
Nav2 = child, Property2 = name, Value2 = "lido:term"
Property3 = data, COMP3 = equal, Value3 = "Objekt"
```

#### *E.7.4 Ambiguous Values.*

*MIDAS.* The type of the described artwork is specified as "Schloss" (German for "castle" and "lock").
Runtime: 7435 ms

**Listing 31: Concretisation of the pattern COMPVAL1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "5230"
Property2 = attribute "Value", COMP2 = equal, Value2 = "Schloss"
```

*LIDO.* See above.
Runtime: 6011 ms

**Listing 32: Concretisation of the pattern COMPVAL2**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child4, Property1 = name, Value1 = "lido:objectWorkType"
Nav2 = child, Property2 = name, Value2 = "lido:term"
Property3 = data, COMP3 = equal, Value3 = "Schloss"
```

#### *E.7.5 Abbreviations.*

*MIDAS.* The name of an artist contains a dot.
Runtime: 1178 ms

**Listing 33: Concretisation of the pattern MATCH1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "kue"
Nav1 = child, Property1 = attribute "Type", Value1 = "3100"
PropertyA = attribute "Value", ValueA = "\."
```

*LIDO.* See above.
Runtime: 8729 ms

**Listing 34: Concretisation of the pattern MATCH2**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child8, Property1 = name, Value1 = "lido:nameActorSet"
Nav2 = child, Property2 = name, COMP2 = equal, Value2 =
      "lido:appellationValue"
PropertyA = data, ValueA = "\."
```

## E.8 Misplaced Information

#### *E.8.1 Misfielded Values.*

*MIDAS.* A field for specifying the kind of date given in another field contains only digits. This indicates that the date is given in the wrong field.
Runtime: 7445 ms

**Listing 35: Concretisation of the pattern MATCH1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "5060"
PropertyA = attribute "Value", ValueA = "^[0-9/]+$"
```

*LIDO.* The measurement unit field contains digits probably representing the actual measurement value which, however, should be given in another field.
Runtime: 7424 ms

**Listing 36: Concretisation of the pattern MATCH1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child7, Property1 = name, Value1 = "lido:measurementUnit"
PropertyA = data, ValueA = "[0-9]"
```

#### *E.8.2 Extraneous Data.*

*MIDAS.* The field for specifying the dating of an artwork contains more than 10 letters. This indicates that more than the expected information is given.
Runtime: 10235 ms

**Listing 37: Concretisation of the pattern MATCH1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "5064"
PropertyA = attribute "Value", ValueA = "[a-zA-Z ]{10}"
```

*LIDO.* The political entity is additionally given in a field for specifying the name of a geographic place. It follows after a comma.
Runtime: 6453 ms

**Listing 38: Concretisation of the pattern MATCH1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child7, Property1 = name, Value1 = "lido:namePlaceSet"
PropertyA = data, ValueA = ","
```

## E.9 Redundant Data

#### *E.9.1 Exact Duplicate Records.*

*MIDAS.* Exact duplicate atelier records.
Runtime: 39046 ms

**Listing 39: Concretisation of the pattern EXDUP**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "wer"
```

*LIDO.* Exact duplicate object records.
Runtime: estimated 4.1 weeks (not finished)

**Listing 40: Concretisation of the pattern EXDUP**

```
Nav0 = child2, Property0 = name, COMP0 = equal, Value0 = "lido:lido"
```

#### E.9.2 *Approximate Duplicate Records.*

*MIDAS.* Multiple atelier records with equal location, type and active time may be approximate duplicates.
Runtime: 121129 ms

**Listing 41: Concretisation of the pattern APPDUP**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "wer"
Nav1 = child, Property1 = attribute "Type", Value1 = "3560"
Nav2 = child, Property2 = attribute "Type", Value2 = "3580"
Nav3 = child, Property3 = attribute "Type", Value3 = "3680"
```

*LIDO.* Multiple records reference the same unique, published identification of the described object. This indicates duplicate records describing the same object.
Runtime: 4330299 ms

**Listing 42: Concretisation of the pattern UNIQUE1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child, Property1 = name, Value1 = "lido:objectPublishedID"
Property2 = data
```

#### E.9.3 *Information Placed in Multiple Locations.*

*MIDAS.* The given first and middle names of an artist are equal.
Runtime: 1380 ms

**Listing 43: Concretisation of the pattern COMP**

```
Nav0 = child3, Property0 = attribute "Type", COMP0 = equal, Value0
      = "kue"
Nav1 = child, Property1 = attribute "Type", COMP1 = equal, Value1 =
      "3100"
Nav2 = child, Property2 = attribute "Type", COMP2 = equal, Value2 =
      "3105"
PropertyA = attribute "Value", COMPA = equal, PropertyB = attribute
      "Value"
```

*LIDO.* The same name is given multiple times in one actor element.
Runtime: 10029 ms

**Listing 44: Concretisation of the pattern UNIQUE2**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child7, Property1 = name, Value1 = "lido:actor"
Nav2 = child, Property2 = name, Value2 = "lido:nameActorSet"
Nav3 = child, Property3 = name, Value3 = "lido:appellationValue"
PropertyA = data
```

## E.10 Heterogeneous Data

#### E.10.1 *Heterogeneous Measure Units.*

*MIDAS.* The size of an artwork must be given as height times width in cm without stating the measure unit explicitly. The pattern detects violations.
Runtime: 6445 ms

**Listing 45: Concretisation of the pattern MATCH1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "5360"
PropertyA = attribute "Value", ValueA = "^[0-9]+(,[0-9]+)?( x [0-
      9]+(,[0-9]+)?)? (m|mm)( \([a-zA-ZäüöÄÜÖ ]+\))?$"
```

*LIDO.* LIDO explicitly allows heterogeneous measure units. This is why we do not consider heterogeneous measure units to be a problem in the LIDO data and did not create a corresponding pattern.

#### E.10.2 *Heterogeneous Value Representations.*

*MIDAS.* The type of the described artwork is specified as “Print” even though the equivalent German word “Druck” is preferred.
Runtime: 7564 ms

**Listing 46: Concretisation of the pattern COMPVAL1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "5230"
Property2 = attribute "Value", COMP2 = equal, Value2 = "Print"
```

*LIDO.* See above.
Runtime: 5652 ms

**Listing 47: Concretisation of the pattern COMPVAL2**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child4, Property1 = name, Value1 = "lido:objectWorkType"
Nav2 = child, Property2 = name, Value2 = "lido:term"
Property3 = data, COMP3 = equal, Value3 = "Print"
```

#### E.10.3 *Heterogeneous Structural Representations.*

*MIDAS.* If no artist can be specified for an artwork, then the XML element indicating a relation to an artist must be omitted in the records describing the artwork. The pattern detects violations.
Runtime: 6503 ms

**Listing 48: Concretisation of the pattern MANDSTRUC2**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "obj"
Nav1 = child, Property1 = attribute "Type", Value1 = "ob30"
Nav2 = child, Property2 = attribute "Type", Value2 = "3100"
```

*LIDO.* An element for specifying an actor’s name contains multiple names instead of the actor record containing multiple of such elements.
Runtime: 7809 ms

**Listing 49: Concretisation of the pattern CARD2**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = child8, Property1 = name, Value1 = "lido:nameActorSet"
Nav2 = child, Property2 = name, Value2 = "lido:appellationValue"
```

## E.11 Misspellings

As explained in Section 5.2 this problem could not be covered.

## E.12 Semantically Incorrect Data

*E.12.1 False Values.* As explained in Section 5.2 this problem variant could not be covered.

*E.12.2 False References.* As explained in Section 5.2 this problem variant could not be covered.

#### *E.12.3 Doubtful Data.*

*MIDAS.* The given birth date of an artist is followed by a question mark indicating uncertainty.
Runtime: 1592 ms

**Listing 50: Concretisation of the pattern MATCH1**

```
Nav0 = child3, Property0 = attribute "Type", Value0 = "kue"
Nav1 = child, Property1 = attribute "Type", Value1 = "3270"
PropertyA = attribute "Value", ValueA = "\?$"
```

*LIDO.* A given appellation value is followed by a question mark indicating uncertainty.
Runtime: 10046 ms

**Listing 51: Concretisation of the pattern MATCH1**

```
Nav0 = child2, Property0 = name, Value0 = "lido:lido"
Nav1 = descendant, Property1 = name, Value1 =
      "lido:appellationValue"
PropertyA = data, ValueA = "\?$"
```