Bogumila Hnatkowska
Zbigniew Huzar
Lech Tuzinkiewicz

EXTRACTING CLASS DIAGRAM FROM HIDDEN DEPENDENCIES IN DATA SETS

Abstract *A conceptual model is a high-level graphical representation of a specific domain that presents its key concepts and the relationships between them. In particular, these dependencies can be inferred from instances of concepts being a part of big raw data files. This paper aims to propose a method for constructing a conceptual model from data frames encompassed in data files. The result is presented in the form of a class diagram. The method is explained with several examples and verified by a case study in which the real data sets are processed. It can also be applied for checking the quality of a data set.*

Keywords conceptual model, class diagram, UML, data retrieval, raw data, csv

Citation Computer Science 21(2) 2020: 211–236

1. Introduction

One aim of big data analysis is to find out the relationships among the data. The data can be stored in a variety of formats, typically in the form of tables with only basic information like a column/row name or column/row type. Retrieving information about the entities and the relationships among these entities from these tables can be a challenging and time-consuming process; eventually, it may be presented in the form of a conceptual model. It is especially important to have the means to present these relationships in a meaningful and readable way [4]; one of these is a UML class diagram. Other possibilities include ERD diagrams or semantic networks.

A conceptual model is a high-level representation of a target problem made from the composition of concepts that are used to help people know, understand, or simulate the subject that a model represents. The term "conceptual model" may be used to refer to models that are formed after a conceptualization or generalization process [12]. Conceptual models are often abstractions of things in the real world, both physical and social.

It should be underlined that the set of data usually represents some application domain. So, when recreating a conceptual model based on the available data, it is necessary to keep its compliance with the application domain. Semantic correspondence between the conceptual model and the application domain is a crucial and very sensitive point of the modeling. In conceptual modeling (as opposed to the numerical analysis of big data), it is necessary to have some knowledge delivered by relevant documentation or by domain experts.

Conceptual modeling is one step in the process of data analysis. After delivering, the data should be cleaned and possibly preliminarily analyzed. When implemented correctly, a conceptual model should accomplish the following [8]:

- enhance an individual's understanding of the representative system;
- provide a point of reference for system designers to extract system specifications;
- document the system for future reference and provide a means for collaboration.

(provided that the model has acceptable fidelity to the modeled problem domain).

This paper aims to propose a method of revealing a conceptual data model (a structural perspective only) from data frames – raw data delivered as a dataset (e.g., a csv file). The proposed method can be applied for two purposes:

- discovering existing (in data) entity types (classes of objects) and the relationships among them; the result can be used for different purposes (for example, to visualize complex dependencies among the data) to document them (e.g., in the form of an ontology) [15];
- checking the quality of the data describing a specific domain if the data is to be used for different purposes; e.g., for validating the data against a real domain.

The proposed method discovers functional dependencies among the analyzed data, gathers the attributes (names for data values) into classes, and finds any relationships among them. The process itself is adapted to object-oriented modeling –

the result is represented as a UML class diagram – and takes the relationships specific to object models into consideration; e.g., associations, association classes, generalizations, and compositions. UML is now considered to be a classic modeling language, well-suited not only for the object-oriented paradigm but also used for conceptual data modeling (e.g., [10]).

This paper presents an extended and refined version of the algorithm first described in [6]. Also, the list of illustrating examples is outspread to cover all derivation rules.

The rest of the document is structured as follows. Section 2 brings the basic definitions necessary for understanding the algorithm of conceptual model creation. The next section gives a short review of the relevant literature, and Section 4 discusses the problems with data cleaning as the data pre-processing process. Section 5 presents the algorithm of conceptual model creation. Illustrating examples are given in Section 6, while a simple case study is presented in Section 7. Section 8 concludes the paper.

2. Basic definitions

This section introduces a list of basic definitions necessary for further considerations. Let us assume that a set of data is given in the form of a data frame $DF=\langle H, B\rangle$, where header H is a set of attribute names and B is a set of items (tuples). It is also assumed that data frame DF is associated with a given application area, which is the basis for the data frame interpretation.

Each attribute name $a \in H$ has a data type T_a assigned, noted as $a : T_a$. For a given attribute name a, its type is denoted by $type(a)$. Taking into account an undefined value of any attribute $\perp$, the header of the data frame may be considered as the set $\{\langle a : T_a \text{ or } a = \perp\rangle \mid a \in H\}$.

Each item of the data frame is a partial function from the attribute names into the respective data types. Notation $\langle a, v\rangle$ means value v is assigned to attribute a. Hence, an item is a set $\{\langle a, v\rangle \mid a \in H\}$. The set of items is denoted by B.

Undefined value $\perp$ is interpreted as missing or inapplicable. This means that the value $\perp$ of an attribute $a \in H$ in a given item of a data frame represents valuable information relating only to the set of attributes $H \setminus \{a\}$.

The projection of an item t into a subset of attribute names $A \subseteq H$ is defined as $t[A] = \{\langle a, v\rangle \mid a \in A\}$ ([9]). $B[A]$ denotes the set of all items $t[A]$ belonging to B.

Let X, $Y \subseteq H$; by $X \to Y$ a functional dependency (FD) is denoted, which means that, for any t_1, $t_2 \in B$ if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$; notation $X \not\to Y$ means that there is no FD between X and Y. A functional dependency $X \to Y$ is minimal if removing an attribute from X makes it invalid [9]. An attribute A is partially functionally dependent of set of attributes X, if there exists such an $X' \subset X$ that $X' \to A$ ([7, 9]). A subset X such that $X \subseteq Y \subseteq H$ is called a candidate key with respect to Y if $X \to Y$ and $X' \not\to Y$ for each $X' \subset X$ ([7]). By $CK(Y)$ is denoted the set of all candidate keys with respect to Y.

By $X \Rightarrow Y$, a weak functional dependency (*WFD*) is denoted, which means that there exists $X \rightarrow Y$ after removing such items from B for which $\perp$ is a value of any attribute in $B[X]$.

3. Related works

How the different types of dependencies among data can be retrieved is described in [9], for example. In our approach, we only concentrate on functional dependencies (and skip other types; e.g., inclusion, approximated *FD*, or conditional *FD*). We follow a top-down procedure in which candidate *FDs* $X \rightarrow Y$ are derived first and next examined (on real data) starting from X consisting of one attribute. Even for large data sets, we take all items into account. The list of *FDs* is limited by the use of pruning methods.

There is not much research addressing the same problem of inferring a conceptual model from data. The closest one is [15], which presents the TANGO (Table ANalysis for Generating Ontologies) method. This is 'a formalized method of processing the format and content of tables that can serve to incrementally build a relevant reusable conceptual ontology' [15]. The authors use different heuristics to build a table from partially unstructured data. The result is called a normalized table. The mini-ontologies retrieved from the tables are visually represented by an object in the Object-oriented Systems Model (OSM) notation. Similar to our approach, the elements of the model are mined from data based on the functional dependencies and inclusion dependencies, and the multiplicity is defined by observing the mandatory and optional patterns in the data. However, the detailed algorithms of data extraction are not given. Another problem is that the resulting OSM model is difficult to be interpreted in terms of the classes and their properties. All of the rectangles on the model represent separate data sets that are somehow connected; e.g., a 'country' and its 'population' are represented as separated but linked entities in OSM, while in the UML 'population' and 'country' (name) will be structural features of the same class. The other difference is that the tables in TANGO are generated automatically from data, which sometimes leads to a strange structure with columns not containing real data but serving for grouping purposes; e.g., a column containing in all rows the word 'Religions:' followed by several columns informing about the percentage of a specific religion in a country. We assume that the data is cleaned and any columns with no information are removed before processing.

Another interesting bit of research is [5], in which the authors distill class diagrams from spreadsheets using the so-called Gyro approach. This approach assumes that the data in spreadsheets is organized according to some patterns and separated with empty cells, which determine the relationships among the entities. The recognized pattern is translated into a parse tree and then into a class diagram. In the case when more than one pattern can be applied to a specific set of data, the algorithm returns the set of all. The method distinguishes between label cells, cells with values, and cells with formulas (which is not the case in our approach). The formulas are translated into methods.

In [14], the authors also concentrate on patterns that are potentially used for data in spreadsheets and propose a meta-model for their representation. Such a meta-model can also be used later in our method for data normalization.

The opposite approach in which a conceptual model (in the form of an ontology) is used for data extraction from web pages is presented in [2]. This approach starts with the definition of an ontology instance from which a database schema is generated, with matching rules for constants and keywords. After that, a record extractor is used for data cleaning, and a recognizer is applied to find the parts in unstructured chunks that match the rules. The last step is transforming the found data into a database using the defined heuristics, which makes the data querying possible.

4. Data preparation

Data cleaning is the process of analyzing, detecting, and correcting the errors and inconsistencies in a data set to improve the data quality [1,11,13]. Generating a conceptual data model (domain model) based on raw data requires a preliminary data analysis and, in most cases, improving its quality (in the process of data cleaning) to ensure compliance with the represented domain of the considered problem.

In our experience, the critical issue in data analysis is to understand and interpret a data set. Therefore, we believe that a preliminary assessment of the data sample should be made before proceeding with the process of generating a conceptual data model. This activity is particularly important when we do not know the attributes of the analyzed data sample in detail. The minimal scope of data analysis should include at least the following:

- number of attributes in data set with their value types;
- number of records (observations);
- number of undefined values in entire data set.

For numerical data, it is crucial to determine the number of unique values, the average value, and the minimum and maximum values. This allows us to evaluate the values of the attributes in the context of the domain. For string/text data (categorical attributes), it is recommended to obtain information on the number of unique values, missing (n/a, null, etc.) values, and a sample of the most common attribute values.

The completeness of data values is one of the crucial aspects that has a significant impact on the correct interpretation of potential dependencies among the data. The input values should be consistent with acceptable domain values. Also, it is essential that all values from these sets occur in the data sample in the case of finitely enumerated domains.

For conceptual model extraction purposes, this approach maybe not be deficient. The data set should be potentially enriched with a variety of possible variants of the attribute values. Furthermore, attributes that can play roles as candidate keys in the source dataset should have unique values, which can be checked by determining the value of the so-called 'unique strength' (as a percentage) of each candidate key.

A 'unique strength' of less than 100% indicates that duplicate values exist [16]. At this stage, it is also worth considering the case of attributes that do not have specific values or have a constant value. In the case when any attribute has no specified values or all of the values are the same in the considered data set, then this attribute should be rejected from the dataset.

In the end, it should be mentioned that, after presented an initial analysis, the criteria for assessing the data quality must be determined and adopted by the analyst (depending on the considered problem domain).

5. Algorithm of disclosing data conceptual model

This section outlines an algorithm that aims at a derivation of a conceptual model (in the form of a UML class diagram) from the given data frame DF. The algorithm is computationally very complex due to the large number of functional dependencies, which is usually greater than the number of dependencies existing in the considered domain. This is usually caused by the low quality of data that does not reflect all possible cases in the reality. We propose a simple metric to prioritize functional dependencies. Dependency $X \to Y$ has a higher rank over $X' \to Y$ if $|X| < |X'|$. Theoretically, the computational complexity of the algorithm is in the order of $O(m * n!)$, where n is the number of attributes in header H (describing a schema of the data frame), and m is the number of entities in the data frame. For the reason of the complexity, we apply some heuristics in the algorithm presented below to avoid the complete search of a space of solutions. The applied heuristics were determined based on previously made experiments.

5.1. Notation used

In further, the following notation is used:

- $Cl(X)$ represents a class where X is a set of its attribute names.
- $Cl(X, \mathit{root})$ represents a class with attributes X marked as root.
- $At(C)$ represents a set of attributes of class C.
- $As(C_1 n_1, \ldots, C_k n_k)$ represents an n-ary association among $C_1, \ldots, C_k$ classes with multiplicities $n_1, \ldots, n_k$ at respective association ends.
- $AC(X, C_1 n_1, \ldots, C_k n_k)$ represents an n-ary association class C with set of attributes X, associated with classes $C_1, \ldots, C_k$ where $n_1, \ldots, n_k$ are multiplicities at respective association ends.
- $Cm(C, C_1 n_1)$ represents a composition relationship, where C is a composite and C_1 is a component with multiplicity n_1 at its end.
- $Gen(C_1;\ C_2)$ represents a generalization relationship, where C_1 is a parent and C_2 is a child.
- Let fd be a functional dependency such that $fd : X \to Y$. We say that X is the *source* ($\mathit{source}(fd) = X$), and Y is the *target* ($\mathit{target}(fd) = Y$) of functional dependency fd. The number of elements in X is called the *grade* of fd ($\mathit{grade}(fd) = |X|$).

For a set of attributes $S \subseteq H$, we define two auxiliary functions:

- $S \rightarrow$ – represents a set of attributes (disjoint with S) being functionally determined directly or indirectly by S as a whole and not defined by anything outside S. More formally: $S \rightarrow = \{S'\colon\ S \rightarrow S'$ and $S \cap S' = \emptyset\ \}$
- $\rightarrow S$ – a flattened set of subsets of attributes (disjoint with S) that functionally determine S.
  Formally: $\rightarrow S = \{S'\colon\ S' \rightarrow S$ and $S \cap S' = \emptyset$ and $\forall S'' : S'' \subset S' \Rightarrow S \not\rightarrow S''\ \}$

5.2. Algorithm definition

5.2.1. Introduction

The top-down approach was used to present the algorithm. Main function *ModelGeneration* calls a number of sub-functions defined separately either in pseudocode or by an activity diagram. The sub-function description presents its goal and gives additional details. We assume global visibility of the data frame and its parts (H–header, B–body) within the functions as well as visibility of the function results. The activity flow of the *ModelGeneration* function is defined in (Fig. 1).

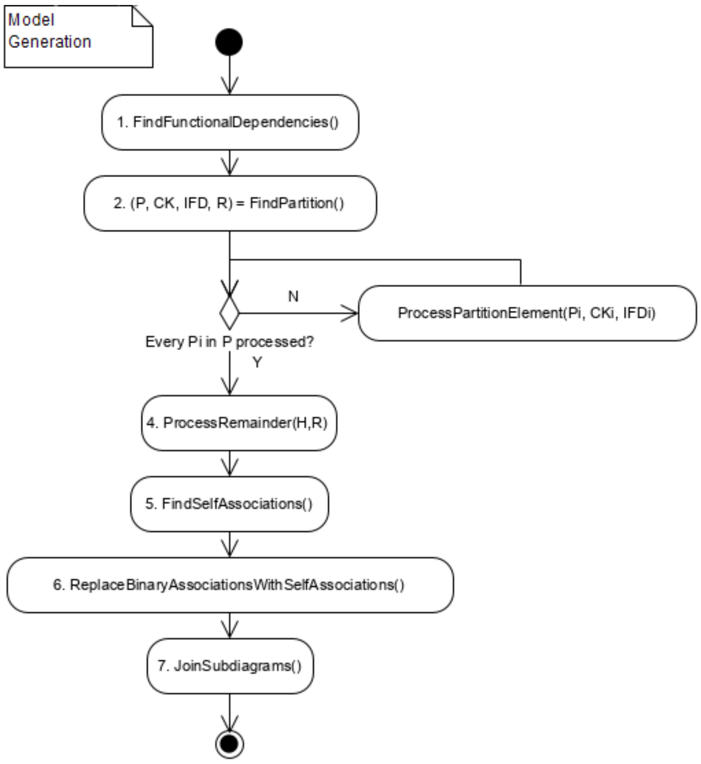
Model Generation
1. FindFunctionalDependencies()
2. (P, CK, IFD, R) = FindPartition()
N
ProcessPartitionElement(Pi, CKi, IFDi)
Every Pi in P processed?
Y
4. ProcessRemainder(H,R)
5. FindSelfAssociations()
6. ReplaceBinaryAssociationsWithSelfAssociations()
7. JoinSubdiagrams()

Figure 1. *ModelGeneration* function

The main function consists of the seven stages (sub-function calls) described below.

5.2.2. Stage 1 – finding functional dependencies

Functional dependencies are retrieved from data samples. We start with the selection of any singular attribute and check if it is a source for any dependency. If yes, the found dependency is minimal, and no other checks of supersets of the selected attribute are made. If not, we subsequently select sets of two, three, and so on attributes, checking whether they are sources of any minimal dependency. The computing complexity of this searching is $O(m * n!)$. Therefore, the length of the LHS (Left Hand Side) of *FD* is limited to four attributes for practical reasons.

5.2.3. Stage 2 – finding partition

Our algorithm follows the divide-and-conquer strategy. The functional dependencies found in the previous step determine a partial partition of set H; i.e., $P = P_1 \cup P_2 \cup \ldots \cup P_K$, where $P_i \cap P_j = \emptyset$, and $P_j \subseteq H$, and remainder $O = H \setminus P$. Each element P_i is processed later, giving a piece of the class diagram.

Following the grades of the functional dependencies, the partition is determined uniquely. It is calculated by the *FindPartition* function, which additionally identifies a set of candidate keys CK_i for each P_i. The next outcome is the remainder, and the last is a set IDF_i for each P_i, which contains the functional dependencies for the further processing defined within P_i.

The *FindPartition* function uses three auxiliary functions (defined below in a declarative way); i.e., *FindKeyFD*, *FindSupersetOf* and *JoinDependencies*. The first returns key functional dependencies from H; i.e., those dependencies with a grade equal to L (L is the function parameter) whose source is not defined functionally nor is defined cyclically by the dependent elements. Such dependencies are the starting points for class creation. The *FindSupersetOf* function selects (if they exist) from the key dependencies (the *keyFD* parameter) those dependencies that have the lowest grade (equal to K) and whose attributes partially define the attributes being the source of the *fd* dependency (the function parameter). The last *JoinDependencies* combines the found supersets (if any) into one more-complex functional dependency. More formally:

$FindKeyFD(L) = \{fd \in FD : grade(fd) = L$ and
$((\rightarrow source(fd) = \emptyset)$ or $(\rightarrow source(fd) \subseteq source(fd) \rightarrow))\}$
$FindSupersetOf(fd, K, keyFD) = \{fdx \in keyFD\text{: } grade(fdx) = K$ and
$source(fd) \cap source(fdx) \neq \emptyset\}$
$JoinDependencies(fdx) = \{\bigcup_{f \in fdx}(source(f)\} \rightarrow \{\bigcup_{f \in fdx}(target(f))\}$

Example. Let us assume that we have $H = \{A, B, C, D, E, F\}$, and $FD = \{\{B, C\} \rightarrow A,\ \{B, E\} \rightarrow A,\ B \rightarrow D,\ D \rightarrow F\}$. The *FindKeyFD* function will return the first three dependencies as interesting. The $B \rightarrow D$ is considered first, as its grade is equal to 1. The *FindSupersetOf* will return $\{B, C\} \to A$, and $\{B, E\} \to A$ as supersets of $B \to D$. Next, the *JoinDependencies* function will combine them with one more complex dependency $\{B, C, E\} \to A$. Finally, as a result of the *FindPartition* function, we obtain one partition element with one candidate key $\{B, C, E\}$, and the following dependencies: $IFD_1 = \{ \{B, C, E\} \to A, B \to D, D \to F\}$.

Algorithm 1: FindPartition

```
Data: $H$ – a set of attributes
Result: $P = \{P_1, \ldots, P_k\}$ – partition
$CK = \{CK_1, \ldots, CK_k\}$ – $CK_i$ is a set of candidates keys for $P_i$
$O$ – set of attributes outside $P$
$IDF = \{IDF_1, \ldots, IDF_k\}$ – functional dependencies left for further consideration
begin
    $i \longleftarrow 1$
    $P \longleftarrow \emptyset$
    $CK \longleftarrow \emptyset$
    $IDF \longleftarrow \emptyset$
    for $L \leftarrow 1$ to $min(4, |H| - 1)$ do
        $keyFD \longleftarrow FindKeyFD(L)$
        for $fd \in keyFD$ do
            for $K \leftarrow L + 1$ to $min(4, |H|)$ do
                $fdx \longleftarrow FindSupersetOf(fd, K, keyFD)$
            if $fdx \neq \emptyset$ then
                $fdxJoined \longleftarrow JoinDependencies(fdx)$
                $P_i \longleftarrow source(fd) \cup source(fd) \to \cup$
                  $source(fdxJoined) \cup (\forall Y \subseteq source(fdxJoined) : Y \cup Y \to)$
                $CK_i \longleftarrow \{source(fdxJoined)\}$
            else
                $P_i = source(fd) \cup source(fd) \to$
                $CK_i = \{X : X \subset source(fd) \to$ and
                  $X \subset\to source(fd)\} \cup \{source(fd)\}$
            $IDF_i \longleftarrow (fd \cup fdxJoined \cup (\forall x \in FD$ such that $source(x) \subseteq P_i$ and
              $target(x) \subseteq P_i : x)) \setminus fdx$
            if $\neg\exists P_j \in P : P_j \cap P_i \neq \emptyset$ then
                $P \longleftarrow P \cup P_i$
                $CK \longleftarrow CK \cup CK_i$
                $IFD \longleftarrow IFD \cup IDF_i$
                $i \longleftarrow i + 1$ // change the partition element
    $O \longleftarrow H \setminus P$
```

5.2.4. Stage 3 – processing partition

The goal of this stage is to generate a piece of a class diagram separately for each partition element P_i. The elements were constructed in such a way that ensures that the pieces are disjoint. The procedure reuses techniques known from relational databases and the normalization process; however, it adapts them to the object-oriented paradigm. The class(es) generated in one step are considered to be a context for the next-generation step in which the context is linked to the newly generated elements. The *ProcessPartitionElement* procedure (see Fig. 2) processes transitive dependencies recursively by the *ProcessTransitiveDep* function. *Rulex* functions perform simpler transformations – they are defined formally after the main function. The *FindPartialDep* function takes the first candidate key as a parameter for P_i and returns all dependencies in IFD_i, whose targets are partially dependent from this candidate key. The newly created elements (classes, relationships) are visible globally.

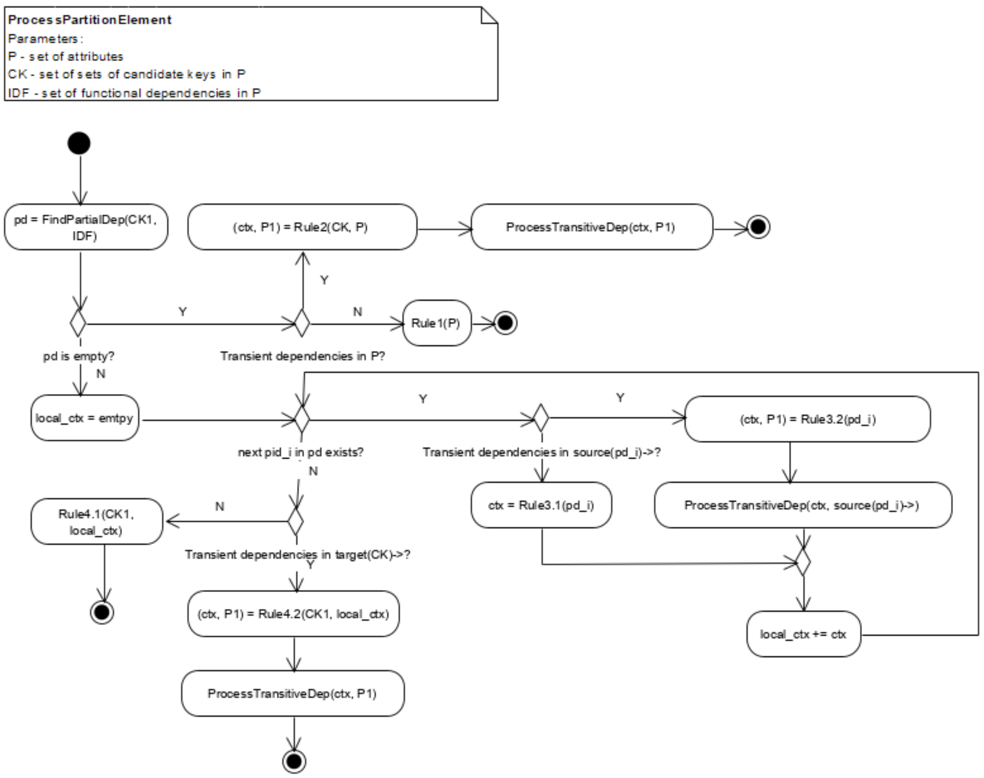
ProcessPartitionElement
Parameters:
P - set of attributes
CK - set of sets of candidate keys in P
IDF - set of functional dependencies in P
pd = FindPartialDep(CK1, IDF)
(ctx, P1) = Rule2(CK, P)
ProcessTransitiveDep(ctx, P1)
Y
Y
N
Rule1(P)
pd is empty?
Transient dependencies in P?
N
local_ctx = emtpy
Y
Y
(ctx, P1) = Rule3.2(pd_i)
next pid_i in pd exists?
Transient dependencies in source(pd_i)->?
N
ctx = Rule3.1(pd_i)
ProcessTransitiveDep(ctx, source(pd_i)->)
Rule4.1(CK1, local_ctx)
N
Transient dependencies in target(CK)->?
Y
(ctx, P1) = Rule4.2(CK1, local_ctx)
local_ctx += ctx
ProcessTransitiveDep(ctx, P1)

Figure 2. Definition of *ProcessPartitionElement* function

Rule 1 takes into a set of attributes as a parameter and creates a root class from all of it.

Rule 2 takes two parameters: a set of candidate key CK for partition element P and returns a newly created root class as a context and a reduced number of attributes for further consideration.

Rule 3.1 is called for a partial dependency *fd* when it is not a source for any transitive dependency. It creates a new root class that is returned as a context.

Rule 3.2 is called when functional dependency *fd* (the parameter) is a source for a transitive dependency. It creates a new root class (*ctx*) and returns a reduced number of attributes for further processing (P').

Rule 4.1 creates a class or association class to be linked (by composition or association class) with the classes obtained from processing of the partial dependencies.

Rule 4.2 works in a similar fashion as *Rule 4.1*. The only difference is that it is called in the context of transitive dependencies; so, this is why the rule returns acontext (a class to which something will be connected in the next stage) and a reduced number of attributes.

Rule 5.1 and *Rule 5.2* are called internally within the *ProcessTransitiveDep* function. Both create classes to be linked with the context by an association or generalization (depending on the case). The second returns a new context for the recursive calls.

Algorithm 2: ProcessTransitiveDep

```
Data: $ctx$ – a set of classes, $H$ – a set of attributes
begin
    $(P, CK, O, IDF) = FindPartition(H)$
    for $P_i \in P$ do
        if there exists any transitive dependency in $IDF_i$ then
            $(ctx', H') \longleftarrow Rule5.2(ctx, CK_i, P_i)$
            $ProcessTransitiveDep(ctx', H')$
        else
            $Rule5.1(ctx, CK_i, P_i)$
```

$FindPartialDep(CK, IDF) =$
$\{fd \in IDF : source(IDF)$ is partially dependent from $CK\}$

$Rule1(P) = Cl(P, root)$

$(ctx, P')\ Rule2(CK, P) =$
$C \longleftarrow Cl(\bigcup_{K \in CK : K \to Y and \nexists Y \to Z} K \cup Y, root)$
$ctx \longleftarrow C$
$P' \longleftarrow P \setminus At(C)$

$ctx\ Rule3.1(fd) =$
$C \longleftarrow Cl(source(fd) \cup target(fd), root)$
$ctx \longleftarrow C$

$(ctx, P')\ Rule3.2(fd) =$
$ctx \longleftarrow Cl(source(fd), root)$
$P' \longleftarrow fd \to$

```
$Rule4.1(CK, localCtx) =$
    $FlatSet \longleftarrow \bigcup_{c \in localCtx}(At(c))$
    **if** $localCtx = \{C_1\}$ **then**
        $C \longleftarrow Cl(CK \rightarrow \backslash At(C_1))$
        $Cm(C_1,\ C\ n)$
    **else if** $localCtx = \{C_1, \ldots, C_k\}$ and $CK \subseteq FlatSet$ **then**
        $C \longleftarrow Cl(CK \rightarrow)$
        $AC(At(C),\ C_1\ n_1, \ldots, C_k\ n_k)$
    **else**
        $Z \longleftarrow Cl(CK \rightarrow)$
        $C \longleftarrow Cl(CK \setminus FlatSet)$
        $AC(At(Z), C_1\ n_1, \ldots, C_k\ n_k, Cn)$

$(ctx, P')\ Rule4.2(CK, localCtx) =$
    $FlatSet \longleftarrow \bigcup_{c \in localCtx}(At(c))$
    **if** $localCtx = \{C_1\}$ **then**
        $C \longleftarrow Cl(CK \setminus At(C_1))$
        $Cm(C_1,\ C\ n)$
    **else if** $localCtx = \{C_1, \ldots, C_k\}$ and $CK \subseteq FlatSet$ **then**
        $C \longleftarrow Cl(Y : CK \rightarrow Y directly)$
        $AC(At(C),\ C_1\ n_1, \ldots, C_k\ n_k)$
    **else**
        $Z \longleftarrow Cl(Y : CK \rightarrow Y directly)$
        $C \longleftarrow Cl(CK \setminus FlatSet)$
        $AC(At(Z), C_1\ n_1, \ldots, C_k\ n_k, Cn)$
    $ctx \longleftarrow C$
    $P' \longleftarrow CK \rightarrow$

$Rule5.1(ctx, CK, P) =$
    $C \longleftarrow Cl(P)$
    **if** $CK_1$ weakly defines any attribute from $ctx$ **then**
        $Gen(ctx, C)$
    **else**
        $As(Cn_1, ctxn_2)$

$(ctx', H')\ Rule5.2(ctx, CK, P) =$
    $C \longleftarrow Cl(CK_1)$
    **if** $CK_1$ weakly defines any attribute from $ctx$ **then**
        $Gen(ctx, C)$
    **else**
        $As(Cn_1, ctxn_2)$
    $ctx' \longleftarrow C$
    $H' \longleftarrow P \setminus At(C)$
```

5.2.5. Stage 4 – processing remainder

In the fourth stage, the attributes outside the partition of initial set H are processed. These belong to the results of the *FindPartition* function and could be somehow connected to existing partition elements (by incoming dependencies). The processing is done by the *ProcessRemainder* function, which works recursively until there is no functional dependency between the input parameters.

Algorithm 3: ProcessRemainder

```
Data: $H$ – a set of attributes
$R$ – a subset of $H$
begin
    $rootClassesH$ = find all root classes created by $ProcessPartitionElement$ for set $H$
    if there exists $fd \in FD$ such that $source(fd) \subseteq R$ and $target(fd) \subseteq R$ then
        $(P, CK, O) \longleftarrow FindPartition(R)$
        for $P_i \in P$ do
            $ProcessPartitionElement(P_i, CK_i)$
        $rootClassesR$ = find all root classes created by $ProcessRemainder$ for set $R$
        $Rule6.1(rootClassesR, rootClassesH, R, H)$
        $ProcessRemainder(R \setminus P, O)$
    for each $A \in R :\rightarrow A \subset (H \setminus R)$ and $A \rightarrow = \emptyset$ do
        $Rule6.2(rootClassesH, A, R, H)$
```

Rule 6.1 takes all root classes created by the specific recursive run of the *ProcessReminder* function and links them to the classes created for the partition of H. *Rule 6.2* takes a singular attribute A, creates a class for it, and links this class to each class whose attributes define A functionally.

```
$Rule6.1(ClassesR, ClassesH, R, H) =$
    for each $C_i \in ClassesR$ do
        for each $C_j \in ClassesH$ such that $At(C_j) \subset (H \setminus R)$
        and there exists $X \subset At(C_i) : X \rightarrow Y$ and $Y \subseteq At(C_j)$, do
            $As(C_i\ n,\ C_j\ 1)$
```

```
$Rule6.2(rootClassesH, A, R, H) =$
    $C \longleftarrow Cl(A)$
    for each $C_i \in rootClassesH : At(C_i) \subset (H \setminus R)$
        and there exists $X \subset At(C_i) : X \rightarrow A$, do
        $As(C_i\ n,\ C\ 1)$
```

5.2.6. Stage 5 – self associations

In the fifth stage, the associations within this class are identified for each previously determined class. An identified association enables the modification of a class by reducing a set of its attributes and replacing the reduced attributes by the association.

Let C be such a class and for $K = \{a_1, \ldots, a_n\}$, where $K \in CK(C)$, there exists a set of attributes $\{b_1, \ldots, b_n\}$ such that the two conditions are satisfied:

(a) $\{b_1, \ldots, b_n\} \subseteq At(C)\backslash K$ such that $type(b_i) = type(a_i)$, and the meanings of a_i and b_i are the same (see Subsection 5.2.9) for $i = 1, \ldots, n$,

(b) $t[b_1, \ldots, b_n] \subseteq t[a_1, \ldots, a_n]$.

If the above conditions are satisfied, this is interpreted as:

(Rule 7.1) The set of attributes $\{b_1, \ldots, b_n\}$ is redundant for class C, which means that this class may be replaced by a class $C'' = Cl(At(C)\ \{b_1, \ldots, b_n\})$.

(Rule 7.2) There is an association $As(C'\ m_1, C'\ m_2)$ where multiplicities m_1, m_2 are to be determined on the basis of tuples from DF that represent instances of C'. What is more, one end of the association will be given the $\{b_1, \ldots, b_n\}$ role name to make tracing the source of the self-association possible.

5.2.7. Stage 6 – self-associations instead of binary associations

In the sixth stage, we consider pairs of previously identified classes. The aim is a refactorization of the association by discovering possible hidden generalizations.

Let C_1 and C_2, and $As(C_1\ m_1,\ C_2\ m_2)$, where $m_1 = 1$ or 0..1 and $m_2 =$ 0..* or 1..*, are given. Moreover, let $K_i \in CK(C_i)$ for $i = 1, 2$. The further proceedings make sense if $K_1 \not\rightarrow K_2$,$K_2 \rightarrow K_1$,and $type(K_1) = type(K_2)$ and the meanings of K_1 and K_2 are the same.

If $K_2 = At(C_2)$, then:

(Rule 8.1) The associated pair of classes is replaced by class C_1 and association $As(C_1\ m_1,\ C_1\ m_2)$ (see Fig. 3a). The ‘many’ end of the association is given the K_2 role name.

(Rule 8.2) Class C_2 is removed, and their associations/generalizations (if any) are moved to class C_1.

If $K_2 \subset At(C_2)$, then a more complex analysis is required. Let us define $A_i = At(C_i)\backslash K_i$ for i =1,2. Now, we analyze the semantics of subsets A_1 and A_2.

Let us assign the name A_{12} for the set of attributes in A_1 and A_2 that have the same meaning (their names may be different, but the types must be the same). If A_{12} is nonempty, then:

(Rule 8.3) The associated pair of classes is replaced by two or three classes (Fig. 3b): a superclass $C_{12} = Cl(K_1 \cup A_{12})$ and association $As(C_{12}\ m_1,\ C_{12}\ m_2)$, where multiplicities m_1, m_2 are to be determined from an analysis of the set of respective tuples in DF. The names for those attributes with the same semantics are taken from C_1.

Additionally:

(Rule 8.4) a subclass $C_{11} = Cl(A_1\ A_{12})$ of C_{12} provided that set $At(C_{11})$ is not empty

(Rule 8.5) a subclass $C_{22} = Cl(A_2\ A_{12})$ of C_{12} provided that set $At(C_{22})$ is not empty

(Rule 8.6) The existing associations/generalizations where C_1 is placed on the end are to be moved to that class in the hierarchy, which contains the attributes that are the cause for the association/generalization.

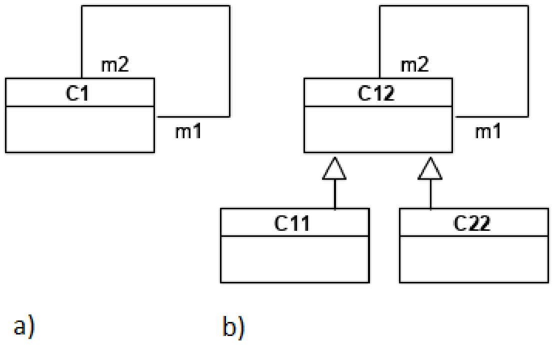

Figure 3. Possible model refactorizations for binary associations

5.2.8. Stage 7 – post-processing

The seventh phase is a post-processing stage that connects separate subgraphs created for the partition elements. In this phase, the classes resulting from the previous phases are examined. For each pair of root classes C_i, C_j $(i \neq j)$:

(Rule 9.1) a new many-to-many association $As(C_i\ m_i,\ C_j\ m_j)$ is created with m_i, m_j derived from an analysis of the set of tuples from DF that represent instances of C_j and C_j.

5.2.9. Attempts to check meaning equivalence among attributes

Some transformation rules (e.g., in the sixth phase) require checking whether the semantics of two attributes is the same. This could be done in different ways; e.g., by asking an expert. The problem with this solution is that the number of questions directed to experts grows exponentially with the number of attributes, even if the necessary conditions (e.g., type equivalences) are met. The decision process could be supported with the calculation of some base measures, including:

- The ratio of the shared values in two attributes (or the set of attributes in the case of complex candidate keys). If the value exceeds some predefined threshold (to be set), either it is assumed that the semantics of the attributes is the same, or an expert is asked for confirmation. This approach should be especially effective for textual (categorical) values; e.g., towns, countries, first names.

- The hierarchy depth for the transitive dependency (if any) between values in the considered attributes. Let us illustrate this method with a simple example. Assume that we have two attributes ($EmployeeId$ and $BossId$) with a functional dependency between them ($EmployeeId \rightarrow BossId$). Some bosses play the role of employees and have their bosses, which is reflected in the values of the attributes. If the hierarchy depth is greater than a specified threshold (e.g., 2), one can assume that a kind of interesting dependency between attributes exists. This approach is recommended for number values, especially those used as identifiers.

6. Illustrative examples

This section helps in understanding the algorithm details. The examples were intentionally prepared for this purpose with the use of test data (data sets containing from several to several dozen items). The functional dependencies among the data instances do not need to be fully conformant with commonly known domains.

The first explains the concept of partition elements – the result of the second phase and how any discovered functional dependencies (in the first phase) help in finding them. Assume we have the source data that is shown in Table 1.

Table 1
Source data for first example

Name [Text]	Age [Number]	IsFemale [Boolean]
John Kowalski	10	No
Ann Nowak	12	Yes
Agatha Smith	10	Yes

As it is easy to observe, the *Name* attribute is the only one with unique values; therefore, it is a key candidate. This also functionally defines the other attributes (*Name* → *Age*, *Name* → *IsFemale*). This means that all of them belong to the same partition element. As there are no transitive nor partial dependencies, all of the attributes will constitute one class (see Rule 1, Fig. 4).

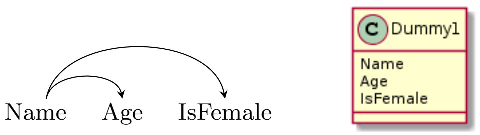

Figure 4. Class generation – example 1

The second example shows how a class is created with two candidate keys read from functional dependencies (*Group*, *Time*). The partitioning process returns one partition element with both attributes. The remainder parts contain the *Something* attribute. Again, Rule 1 produces one class. The *Something* attribute is not covered, as there is no dependency that points to it – see Figure 5.

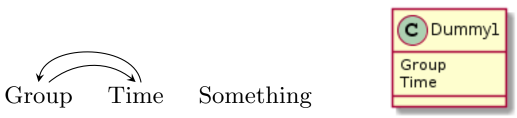

Figure 5. Class generation – example 2

The third example illustrates how to serve a transitive functional dependency (derived from *Group* → *Course* → *CourseName*, and *Group* → *CourseName* → *Course*) – see Fig. 6. The partitioning process returns one partition element with one candidate key (*Group*). The first run of the *ProcessPartitionElement* function creates the *Dummy1* class (Rule 2). The class contains the key as well as the attributes defined by it (*Room*) unless they are involved in any transitive dependency. Such attributes are processed by recursive function *ProcessTransitiveDep*. The first call of it creates the *Dummy2* class and links it with the one-to-many association with *Dummy1* (Rule 5.1).

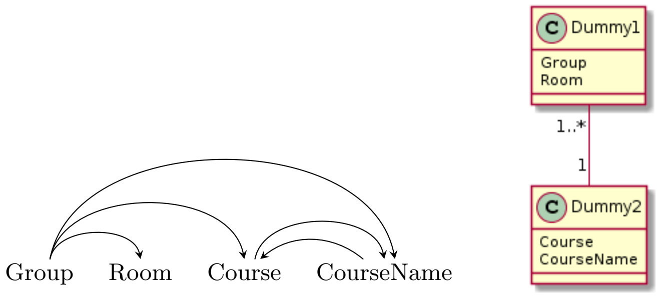

Figure 6. Generation of associations resulting from transitive *FD*

The next example shows how to create a generalization relationship based on a weak functional dependency (see Fig. 7, WFD).

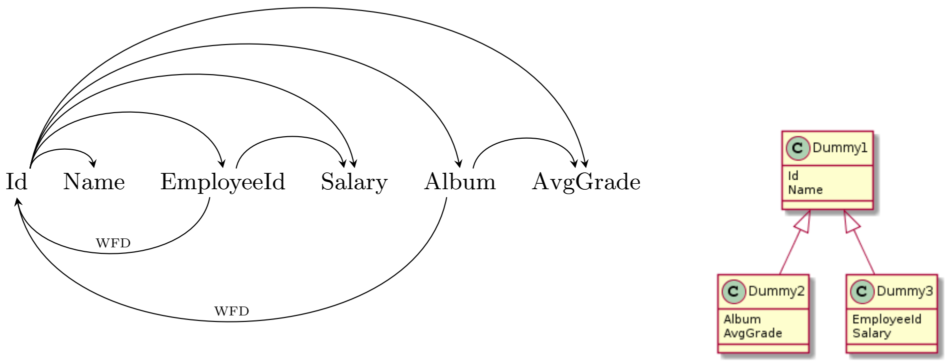

Figure 7. Generation of generalization relationship

The data represents information about a person (Id, $Name$) and two subtypes: *Student* (represented by the *Dummy2* class) and *Employee* (represented by the *Dummy3* class). The application of Rule 2 produces the *Dummy1* class. The *ProcessTransitiveDep* function starts with $P1 = \{EmployeeId, Salary, Album, AvgGrade\}$ in which two partition elements are found, each of which is the source of a new class. These new classes are connected with a generalization (Rule 5.1) with *Dummy1*. The rules are applied for disjoint and incomplete data.

The presence of partial dependencies activates another group of transformation rules. Let us present the following example. The file contains data about invoices, including $InvoiceNr$, $Date$, and rows defined by: $RowNr$, $Product$ (name), product $Quantity$ – see Fig. 8. The partition contains only one element, with pair $\{InvoiceNr, RowNr\}$ as a candidate key. The *FindPartialDep* function returns one functional dependency ($InvoiceNr \to Date$) for which Rule 3.1 is run. This creates the *Dummy1* class, which is returned as the local context. Next, Rule 4.1 creates a new class ($Dummy2$) and links it via composition with *Dummy1*.

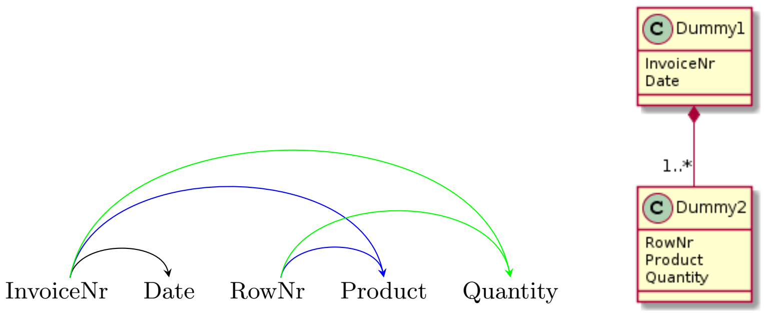

Figure 8. Generation of composition

The next example covers the case in which an association class is created as a result. The test data is typical: we have *Groups* (of students) taking their classes for a specific $Course$ at a specific time. The students identified by *Album* have $Name$. Students are given *Grades* – see Fig. 9. All attributes belong to the same partition element, with pair $\{Group, Album\}$ as a candidate key. Now, two partial functional dependencies are present in the set of attributes. For each, a singular class is created (the application of Rule 3.1) and added to the local context. Rule 4.1 creates the $Dummy3$ association class (as the candidate key of the partition element is covered by the attributes of the contextual classes) with its ends at $Dummy1$ and $Dummy2$.

Below, another example is shown that demonstrates the application of Rule 4.1. The initial dependencies among the set of attributes are presented in Figure 10. There are partial dependencies from candidate key $K = \{Album, Course, Date\}$ in H. Rule 3.1 is applied twice and produces a set of classes $\{Dummy1 = \{Album, StudentName\}, Dummy2 = \{Course, CourseName\}\}$ as a local context.

Rule 4.1 produces a new class ($Dummy3 = \{Date\}$) and links all of the previously mentioned classes via n-ary association class $Dummy4 = \{Grade\}$. In Figure 10, the association class is represented by a casual class because of the limitations of the notation used.

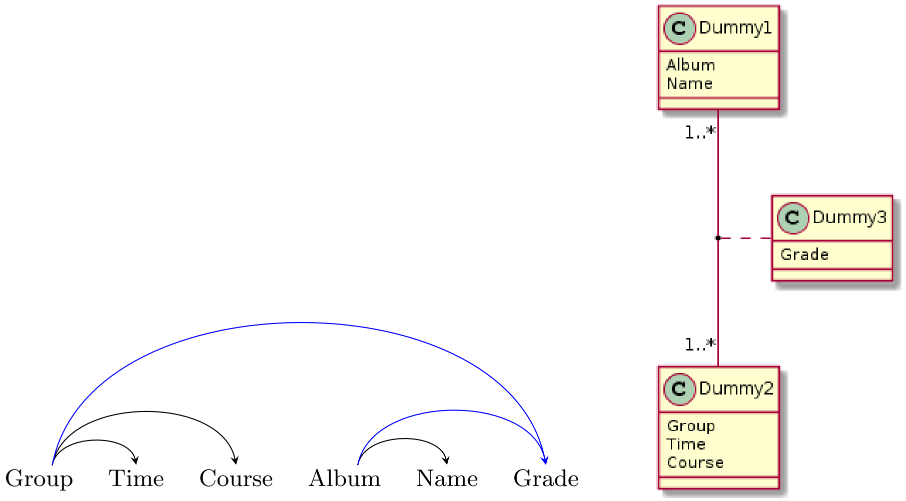

Figure 9. Generation of association class – example 1

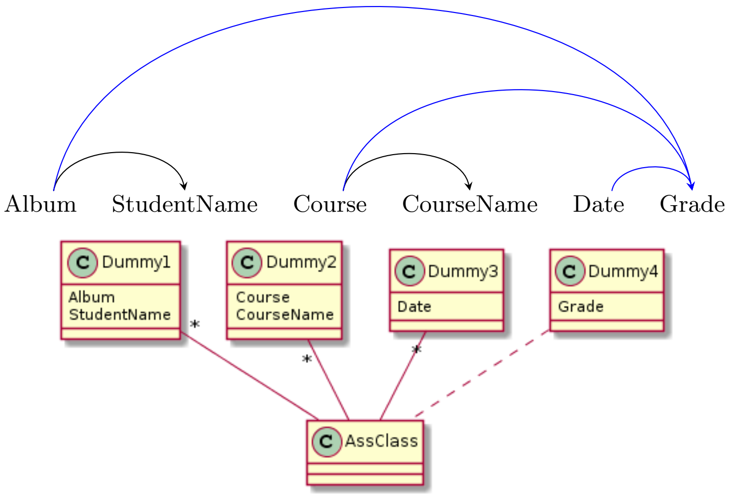

Figure 10. Generation of association class – example 2

Figure 11 shows an example in which a cascade of transitive dependencies is served ($Name \rightarrow Birthdate \rightarrow Age \rightarrow Experience$). First, the $Dummy1$ class is created (Rule 2) with the $Name$ attribute. The rest of the attributes are passed to the $ProcessTransitiveDep$ function. The attributes form one partition element. Because it still contains transition dependencies, Rule 5.2 creates the $Dummy2$ class with the $Birthdate$ attribute (the class is linked via association with $Dummy1$), and the function is called recursively with the limited set of attributes. At the end, Rule 5.1 creates the $Dummy3$ class and links it with $Dummy2$.

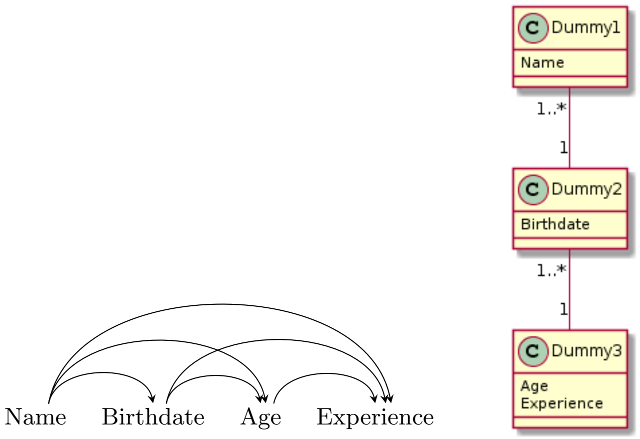

Figure 11. Servicing cascade of transition dependencies

The next example demonstrates the result of the fourth phase. Let us assume that we have discovered the functional dependencies that are shown in Figure 12. The partitioning process returns a partition that consists of two elements ($Group$ is a candidate key in the first, and Car is in the second). These two partition elements are sources for the $Dummy1$ and $Dummy2$ classes. The $Something$ attribute lies outside the partition. As the remaining part does not contain any functional dependency, Rule 6.2 is run immediately. This creates a new $Dummy3$ class and links it via one-to-many associations with the previously created classes.

The model could be refactored within the fifth and sixth phases of the algorithm. The following example demonstrates how a self-association replaces one of the class attributes. Let us consider the functional dependencies from Figure 13. The application of Rule 1 creates one $Dummy1$ class with the $EmployeeId$ key. Now, we check whether such an attribute exists with the same type ($Number$) and the same meaning as the key has. We find one ($BossId$). After this, we check whether the values of the $BossId$ set are included in the values of the $EmployeeId$ set. The answer is positive, so the $BossId$ attribute is removed (Rule 7.1) and replaced with a self-association (Rule 7.2).

The following example demonstrates another refactorization (the result of the sixth phase) within which two classes that are connected via the binary association are replaced by one class with a self-association. Let us consider the functional dependencies in Figure 14. Stages 2–4 produce the $Dummy1$ class with one $EmpId$ attribute, the $Dummy2$ class with the $BossId$ and $BossName$ attributes, and the one-to-many associations between them. The key of the $Dummy1$ class defines the key of the $Dummy2$ class (their types and meanings are the same); the key of $Dummy1$ is its only attribute, so the associated pair of classes is removed. In its place, a copy of the $Dummy2$ class is used with a self-association (Rule 8.1).

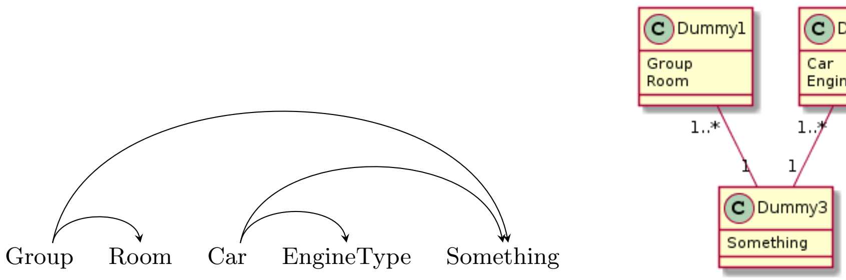

Figure 12. Generation of shared class with associations

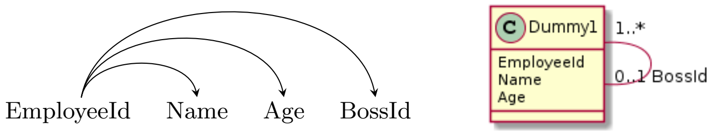

Figure 13. Generation of self-association

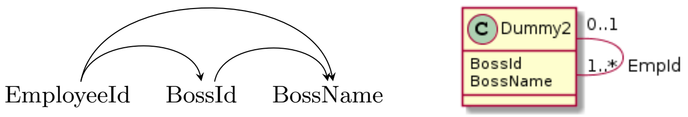

Figure 14. Generation of self-association instead of binary association

Let us now present a more complex version of the previously examined reality (see Fig. 15). Here, we have more attributes that describe both employees and bosses. Before we start the sixth phase, two classes had been created: $Dummy1(EmpId, EmpName, EmpAge)$ associated via many to one association with $Dummy2(BossId, BossName, BossSalary)$. The key of $Dummy1$ – $EmpId$ – defines the key of $Dummy2$ – $BossId$ – functionally. So, the entry conditions for the transformation rules are met. As the $Dummy1$ class not only contains the key, we have to find out A_{12} – the set of attributes in both classes (without keys) that share the same semantics. In this case, $A_{12} = \{BossName(= EmpName)\}$. Rule 6.3 creates the $Dummy3$ (C_{12}) class with attributes $\{BossId, BossName\}$ and a self-association for it. Rule 8.2 creates the $Dummy4$ (C_{11}) subclass with attribute $\{BossSalary\}$, and Rule 8.3 creates the $Dummy5(C_{22})$ subclass with attribute $\{EmpAge\}$.

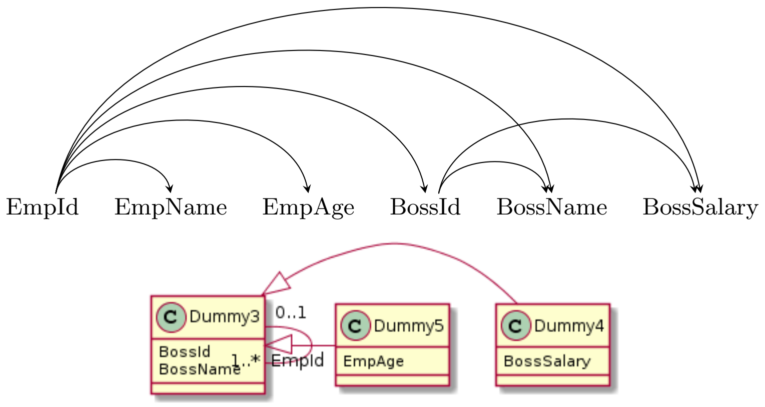

Figure 15. Generation of self-association instead of binary association

The last example illustrates the application of the seventh phase. Let us assume that we have data about students and courses not related by any functional dependency (see Fig. 16). The set of attributes creates a partition with two elements – each of which is the source of one root class: $Dummy1$ (for the courses), and $Dummy2$ (for the students), respectively. In the post-processing stage, a new many-to-many association is created between them (Rule 9.1).

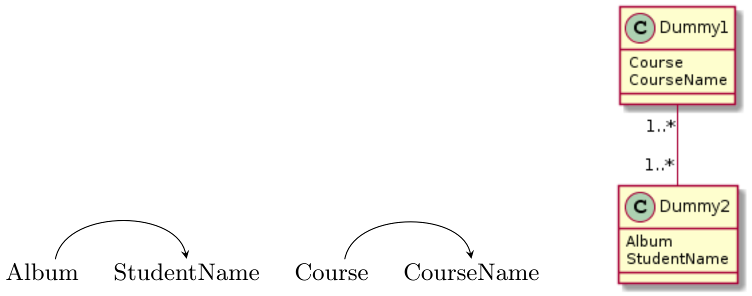

Figure 16. Generation of self-association instead of binary association

The method for the conceptual model extraction from the data frames was implemented in a prototype tool written in Java, which can read and interpret csv files. The tool produces the resulting model using plantUML syntax (http://plantuml.com/), which can be easily translated to a visual form. The implementation only produces binary relationships (plantUML lacks support for n-ary associations) and considers the composite keys of three attributes at most (in practice, it happens rather rarely that more items are necessary). The tool was tested with many simple examples (including all of those presented above) and a few larger for which the authors know the class diagram. In all of the considered cases, the implementation returned a correct or at least acceptable solution (e.g., inheritance would be better, but the binary association can be used instead).

7. Case study

The aim of this section is to illustrate an application of the proposed method to a real example. To make the illustration reliable and verifiable, we decided to embed it in a well-known university domain. We gradually completed data that documented the results of student evaluations prepared by different teachers and observed how these increase would influence the obtained conceptual model. Finally, we gathered the data from four teachers within one academic year.

The full list of attributes appears as follows: {*Album*, *Surname*, *FirstNames*, *Year of study*, *Semester*, *GroupID*, *CourseId*, *CourseName*, *SemType*, *AcademicYear*, *Grade*, *Date*, *EmployeeId*, *EmployeeData*, and *Teacher title*, where 'Album' is a unique identifier for each student; 'Surname' is the student surname; 'FirstNames' is the list of concatenated student's names; 'Year of study' represents a value from 1 to 4 (the actual year the student is in); 'Semester' represents a value from 1 to 7 (the actual semester the student is in); 'GroupID' is a unique identifier of a group of students that take a specific 'CourseId' (this identifier determines the form of a course; e.g., lecture, lab) of a specific 'CourseName' (the same course name can be applied for many course ids); 'SemType' is an enumeration with two literals only ('summer,' 'winter'); 'AcademicYear' is a string with two numbers (e.g., 2017/2018); 'Grade' represents a student's grade for a specific course id in a specific semester and academic year (can be empty); 'Date' informs when the grade was registered by an academic teacher identified by 'EmployeeId' and described by 'EmployeeData' (this attribute contains a concatenation of the teacher surname and name). The last attribute ('Title') is an enumeration representing the formal title of the academic teacher (e.g., prof., Ph.D).

During the first stage, we merge the data from four teachers (about 500 rows) without any grades being included. This was the reason why the algorithm was not able to determine a class for attributes 'Grade' and 'Date' – see Fig. 17a. After adding grades to some courses, the model changed – see Fig. 17b. The program correctly recognized an association class between *Dummy2* (a class representing students) and *Dummy3* (a class representing a specific student group taught by a specific academic teacher).

In the next step, we added data with grades for the second semester (almost 400 rows). Now, the algorithm was able to correctly separate a class representing teachers (*Dummy5*) and courses (*Dummy4*) – see Fig. 18. The course can be assigned to many groups (*Dummy3*). One group can gather students being in different semesters/years of study and must be run in a specific semester type (association to *Dummy7*) and within a specific academic year (association to *Dummy8*). It happened that a few students had to retake selected courses – such a fact was correctly identified by the algorithm – *Dummy2* is connected via association class (*Dummy6*) to a specific group – *Dummy3*.

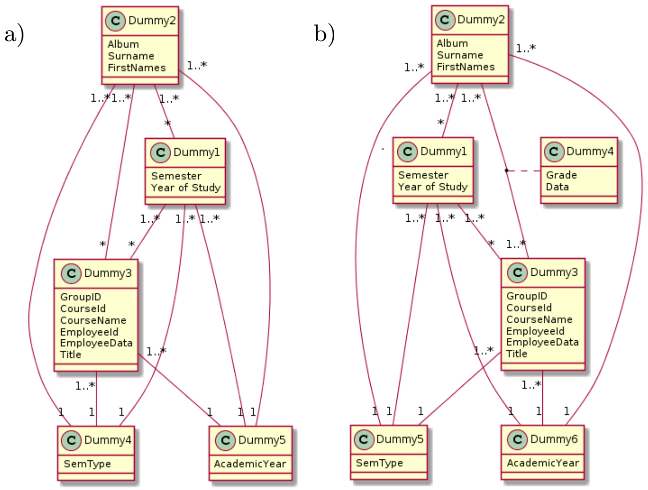

Figure 17. Conceptual model resulting from 1st stage: a) data without grades; b) data with grades

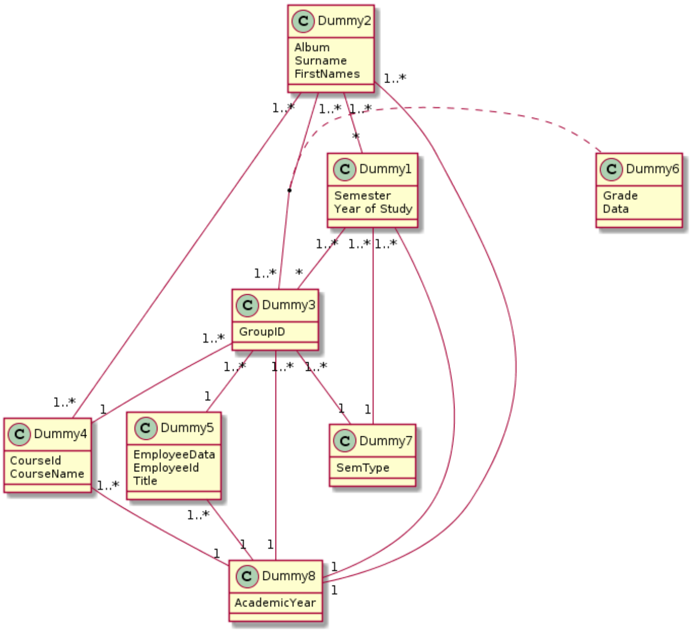

Figure 18. Conceptual model resulting from 3rd stage: data from whole academic year with student grades

8. Conclusions

This paper presents an approach to conceptual data modeling inferred from data frames. The approach reuses known techniques from a database domain related to the database normalization process; however, it adapts them to an object-oriented paradigm and extends them with some additional rules (e.g., generation of composition, self-association, and generalization relationships, generation of sharable classes). The main difference is that the set of functional dependencies is not explicitly defined but is created during the data analysis process. The approach is very sensitive to data quality, which makes data preparation a crucial preprocessing step. Better data results in a better conceptual model.

Among other uses, the proposed method may be used to:

- Interpret a set of data in the absence of knowledge of the problem domain.
- Assess the quality of the sample data.
- Prepare data that should be compatible with the problem domain in question as a software testing set.
- Support didactics in the field of data modeling.

The list of the known limitations of the proposed approach is as follows:

- Attributes are grouped in anonymous classes without meaningful names; e.g., *Dummy1*.
- Generalization relationship is recognized only if it is incomplete; i.e., some data rows have keys of the children instances undefined for parent object.
- Data structure must not contain any patterns that influence data interpretation (compare, e.g., [5]).

The limitations are going to be addressed in the near future; e.g., the names of classes could be defined by reference to a domain ontology or a kind of universal glossary (Wordnet). Another direction of potential research is to extend the consideration for multi-file input or include quality measures for a sample of data and supplement the missing acceptable data cases based on their preliminary assessment.

References

[1] Data Cleansing: Care for most valuable business asset. https://www.hitechbpo.com/data-cleansing.php.

[2] Embley D., Campbell D., Jiang Y., Liddle S.W., Lonsdale D.W., Ng Y.-K., Smith R.D.: Conceptual-model-based data extraction from multiple-record Web pages, *Data & Knowledge Engineering*, vol. 31(3), pp. 227–251, 1999. https://doi.org/10.1016/S0169-023X(99)00027-0.

[3] Embley D., Kurtz B.D., Woodfield S.N.: *Object-Oriented Systems Analysis: A Model-Driven Approach*. Prentice Hall, USA, 1992.

[4] Embley D., Liddle S.: *Conceptual Modeling*, chap. Big Data – Conceptual Modeling to the Rescue. Springer, Heidelberg, 2013.

[5] Hermans F., Pinzger M., Deursen van A.: *ECOOP 2010 – Object-Oriented Programming*, chap. Automatically Extracting Class Diagrams from Spreadsheets, pp. 52–75, Springer, Heidelberg, 2010.

[6] Hnatkowska B., Huzar Z., Tuzinkiewicz L.: *Integrating research and practice in software engineering*, chap. A data-driven conceptual modeling, pp. 97–109, Springer, Cham, 2020.

[7] Kedar S.: *Database Management System*. Technical Publications, USA, 2011.

[8] Kung C., Sölvberg A.: Activity modeling and behavior modeling. In: *Proceedings of the IFIP WG 8.1 Working Conference on Information Systems Design Methodologies: Improving the Practice*, pp. 145–171, North-Holland Publishing Co., Amsterdam, 1986. http://dl.acm.org/citation.cfm?id=20143.20149.

[9] Liu J., Li J., Liu Ch., Chen Y.: Discover Dependencies from Data – A Review, *IEEE Transactions on Knowledge and Data Engineering*, vol. 24(2), pp. 251–264, 2012. http://dx.doi.org/10.1109/TKDE.2010.197.

[10] Ma Z.: Fuzzy Database Modeling with XML, Springer, Boston, 2005. https://doi.org/10.1007/b104945.

[11] McKinney W.: *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, 2nd Edition*. O'Reilly Media, USA, 2017.

[12] Ross R.: Conceputal Model vs. Concept Model: Not the Same!, *Business Rules Journal*, vol. 20, 2019. http://www.brcommunity.com/a2019/b977.html.

[13] Svolba G.: *Data Quality for Analytics Using SAS*, SAS Institute Inc., USA, 2012.

[14] Teixeira R., Amaral V. (2016) On the Emergence of Patterns for Spreadsheets Data Arrangements. In: P. Milazzo, D. Varró, M. Wimmer (eds.), *Software Technologies: Applications and Foundations. STAF 2016*, Lecture Notes in Computer Science, vol. 9946, pp. 333–345, Springer, Cham, 2012.

[15] Tijerino Y., Embley D., Lonsdale D., Ding Y.: Towards Ontology Generation from Tables, *World Wide Web*, vol. 8, pp. 261–285, 2005.

[16] Veerman E., Moss J., Knight B., Hackney J.: *SQL Server 2008. Integration Services. Problem-Design-Solution.* O'Reilly Media, USA, 2010.

Affiliations

Bogumila Hnatkowska
Wroclaw University of Science and Technology, Wyb. Wyspiańskiego 27, 50-370 Wroclaw, Bogumila.Hnatkowska@pwr.edu.pl, ORCID ID: https://orcid.org/0000-0003-1706-0205

Zbigniew Huzar
Wroclaw University of Science and Technology, Wyb. Wyspiańskiego 27, 50-370 Wroclaw, Zbigniew.Huzar@pwr.edu.pl, ORCID ID: https://orcid.org/0000-0002-4362-3502

Lech Tuzinkiewicz
Wroclaw University of Science and Technology, Wyb. Wyspiańskiego 27, 50-370 Wroclaw, Lech.Tuzinkiewicz@pwr.edu.pl, ORCID ID: https://orcid.org/0000-0002-6378-766X

Received: 28.10.2019
Revised: 14.02.2020
Accepted: 17.02.2020