On the Automated Transformation of Domain Models into Tabular Datasets

Alfonso de la Vega, Diego García-Saiz, Marta Zorrilla, and Pablo Sánchez

Dpto. Ingeniería Informática y Electrónica
Universidad de Cantabria, Santander (Spain)
{alfonso.delavega, diego.garcia,
marta.zorrilla, p.sanchez}@unican.es

Abstract. We are surrounded by ubiquitous and interconnected software systems which gather valuable data. The analysis of these data, although highly relevant for decision making, cannot be performed directly by business users, as its execution requires very specific technical knowledge in areas such as statistics and data mining. One of the complexity problems faced when constructing an analysis of this kind resides in the fact that most data mining tools and techniques work exclusively over tabular-formatted data, preventing business users from analysing excerpts of a data bundle which have not been previously traduced into this format by an expert. In response, this work presents a set of transformation patterns for automatically generating tabular data from domain models. The described patterns have been integrated into a language, which allows business users to specify the elements of a domain model that should be considered for data analysis.

Keywords: Data Mining · Model-Driven Engineering · Domain-Driven Design

1 Introduction

Currently, we rely on computer systems for most of the actions we carry out in our daily life. Consequently, these systems gather and store information that, if it is appropriately processed, can help to improve different kinds of systems or processes [7]. Nevertheless, as pointed out by Cao [4], there is a gap between data mining research and practice. According to Cao, the academic community has focused on improving the algorithms for data mining, but much less attention has been paid to how these algorithms can be deployed in real-life environments.

As a very first consequence of this gap, data coming from a certain domain need to be largely processed, formatted and reshaped in order to fit in with the requirements of each data mining algorithm. In general, most data mining algorithms require their input data to be arranged in a tabular format, such as a *CSV* (*Comma Separate Values*) file.

The transformation process is often carried out manually by data scientists. These data scientists create a set of processing scripts which extract data from

In: C. Cabanillas, S. España, S. Farshidi (eds.):
Proceedings of the ER Forum 2017 and the ER 2017 Demo track,
Valencia, Spain, November 6th-9th, 2017,
published at http://ceur-ws.org

its original source and reshape them into a tabular format. The creation of these scripts can be a time-consuming and error-prone process. Moreover, it requires specialised skills on data manipulation, which hampers that average decision-makers could analyse data by themselves, making difficult *data mining democratisation* [1].

To overcome this problem, this work presents a set of patterns for automating the transformation of data coming from an object-oriented domain model into tabular format. These patterns have allowed us to develop a high-level language, called *Lavoisier*, for specifying which elements of a domain model should be provided as input for a data mining algorithm.

This specification is then compiled and, using the transformation patterns, data is automatically retrieved from its source, processed and reshaped, providing an tabular representation of the requested data as output. Therefore, data scientists would not need to create scripts for data formatting and reshaping manually. Moreover, since *Lavoisier* is a high-level language, it might be used for people without specialised skills on data processing.

Expressiveness and effectiveness of our approach have been evaluated using different external case studies. In particular, two data mining open challenges [13, 3] have been used. The first challenge contains data collected by an online business review system, whereas the second one focuses on data extracted from continuous integration tools. These challenges made data publicly available and raised some questions to be answered through their analysis. For both challenges, *Lavoisier* was used to construct tabular representations that feed data mining algorithms, which tried to provide an answer to the proposed questions.

After this introduction, the work is structured as follows: Section 2 exposes the motivation, and formalizes the contributions of this paper. Some technical operations applied in the patterns are introduced in section 3. The paper continues with comments about related work in section 4. The transformation patterns are then described in section 5, while in section 6 an usage example of the *Lavoisier* language is presented. The paper finishes with a recapitulation and an enumeration of future objectives of this work in section 7.

2 Case Study and Motivation

This section explains in detail the motivation behind our work. To illustrate it, a case study based on the 9^{th} edition of the *Yelp Dataset Challenge* [13] is used through the rest of this work. This case study is described below.

Yelp is an American company which provides an online businesses review service for customers to write their opinions and for owners to describe and advertise their businesses. Moreover, additional features, such as events notification or social interaction between reviewers, are also supported. Yelp, during its regular operations, captures different kinds of data, which are being made available for academic usage through challenges. The objective of these challenges is to be able to discover some information hidden in these data which might be of interest for Yelp.

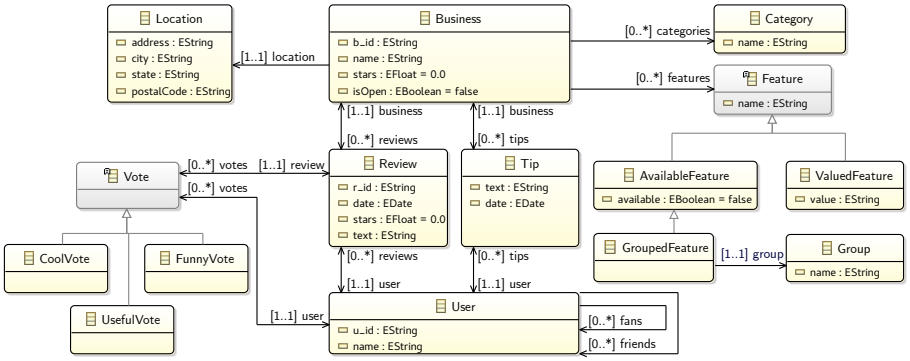

Fig. 1. Domain Model for the Yelp Dataset Challenge.

Yelp's challenge data is provided as a bundle of interconnected files in JSON (*JavaScript Object Notation*) format. From these files, we have abstracted the domain model depicted in Figure 1. According to it, detailed information for each *business* is stored, such as its *location*, an indication of provided *features* (for instance, if WiFi is available, if there is a smoking area, or the minimum required age to enter) and the *categories* which best describes it (Cafes, Restaurant, Italian, Gluten-Free, and so on). *Users* can make *reviews* of businesses, where they star-rate and introduce a text describing their experience. Additionally, user *tips* can get also registered. As Yelp provides some social network capabilities, users can have *friends* or *fans*, and can receive *votes* in their reviews emitted by other users in case they found the review funny, useful or cool.

As part of the challenge, Yelp proposes the participants to find issues that might lead to a successful business, beyond location. This information can be computed using different data mining techniques, such as classification. Nevertheless, as explained before, most algorithms of these techniques only accept input data in a specific sort of tabular format. Therefore, this constraint can be found in tools typically used for data mining, such as R [11] or Weka [10]. These tools only accept as input data arranged in a particular tabular format, such as *CSV* (*Comma Separated Values*) files or relational database tables. The tabular data problem is detailed with the help of Figure 2 in the following.

For the sake of simplicity, let us assume we want to find reasons behind successful business using just the information of Figure 2 (a), this is, stars rating and available features per business. Figure 2 (b) shows two objects representing two different businesses. As previously commented, our very first problem is that we need to rearrange these objects' data in a tabular format. This tabular representation must satisfy the following constraint: all data of each instance of the main class under analysis (in this case, *Business*) must be placed under the same row, as depicted in Figure 2 (d). Other alternative representations, such as the one in Figure 2 (c), which might be more easily produced, would not work properly, as the information of each business gets distributed in several rows.

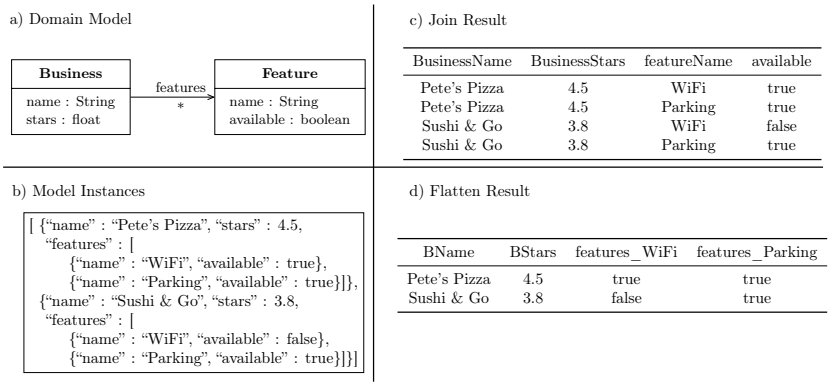

Fig. 2. (a) Excerpt of a ratings domain model; (b) Object graph with some instances of (a); (c) Business-Feature pair merging; (d) A column is created for each feature.

In the following, and inspired by the functional programming paradigm, we will refer to the operation that places all the information of a domain entity in a single row, as a *flattening*.

To implement this *flattening* operation, data scientists often create several scripts by hand. These scripts collect data from the sources and process them using several operations such as products, filters, aggregations and reshapes, until getting the data in the appropriate format. This is a time-consuming and error-prone process. Moreover, to produce these scripts, deep knowledge on data manipulation techniques and tools is required, which average business users often lack. Thus, the situation makes mandatory to hire and rely on the mentioned data scientists.

To overcome this problem, our work aims to provide an automated *flattening* operator. The operator will allow to specify the domain entities to be flattened. Next, the necessary data transformation scripts would be automatically generated from this specification.

This operator has been implemented relying on different data manipulation operations, more specifically, as a combination of *joins* and *pivots*. To make this work self-contained, these operators are described in the following section.

3 Background: Join and Pivot

3.1 Join

This operator, which can be typically found in relational databases, takes as input two domain entities A and B and a relationship r from A to B which connects them. Then, it calculates the Cartesian product between instances of A and B first, and then, filters out those tuples whose instances are not connected through the relationship r.

Figure 2 (c) shows the results of a join for `Business` and `Feature` through *features* (see Figure 2 (a)). Each instance of a *Business* is merged with its related *Feature* instances, generating a row in the resulting table for each pair. This table would not fulfil the constraints imposed by data mining algorithms, as the information of each business spreads over different rows.

3.2 Pivot

This operator is commonly used to rearrange into the same row data related to the same entity which appears, such as in Figure 2 (c), spread over a table. It can be found in some dialects of SQL (although it is not included in the official standards), in data analysis tools, such as R or Pandas, and even in some spreadsheets, like Excel.

A pivot accepts as input a table T, a set *pivotingColumns* of columns through which the table would be pivoted, and a set *pivotedColumns* of columns whose values will be pivoted. For instance, using table shown in Figure 2 (c), we might specify that we want to pivot the column *available* of that table using the *featureName* as pivoting column, arriving at the structure of Figure 2 (d).

Using this information, the pivot operation would work as follows:

1. A new table for holding the results is created. All columns from the original table which are not included in the pivoting or pivoted columns, are copied in the new table. In our case, these will be the *BusinessName* and *BusinessStars* columns (reduced in Figure 2 (d) to *BName* and *BStars* for space reasons).
2. Each instance with distinct values for the previous columns is added as a row to the resulting table. In our case, two different instances are detected, `(Pete's Pizza, 4.5)` and `(Sushi & Go, 3.8)`. It should be noticed as consequence of this step, several rows might be reduced to only one.
3. The set *pivotingValues* of distinct values that can be found in the *pivotingColumns* is calculated. In our example, the resulting set would be {*WiFi, Parking*}.
4. The Cartesian product between $pivotingValues$ and $pivotingColumns$ is calculated. In our example, this would be $\{WiFi, Parking\} \times \{available\} = \{WiFi\_available, Parking\_available\}$.
5. Then, a new column for each pair in the $pivotingValues \times pivotingColumns$ set is added to the resulting table.
6. Each new column is filled with the original values coming from the input table. For instance, in the `(Pete's Pizza, 4.5)` row, the *WiFi_available* column takes its value from the *available* column in the row with values `(Pete's Pizza, 4.5, Wifi)`. In our case, this value would be `true`. If several rows could be identified, the pivot operation would need as input an aggregation function to reduce all the collected values to just one.

As the reader can notice, we can go from Figure 2 (c) to Figure 2 (d), following the join operation by a pivot. This is, the *flatten* operator might be implemented as a combination of *joins* and *pivots*. Indeed, this is what a data

scientist will write by hand into a script for tabbing these data. However, it should be noticed that neither the *join* nor the *pivot* operators by themselves are able to produce a proper tabular format, they need to be used together. Moreover, as the domain model complexity grows, the concatenation of these operators also becomes more complex. Thus, our idea is that this concatenation of operators gets automatically generated.

Next section will analyse whether this issues have already been addressed in the literature.

4 Related Work

To the best of our knowledge, there is no work that provides a suitable implementation for our desired *flatten* operator. Nevertheless, technologies related to data management provide different kinds of operators which are worth to mention.

It is important to note that we are not trying to determine if by using a certain technology we can produce an appropriate tabular representation by hand. With enough effort, a data transformation script might be produced in practically any language. Therefore, we focus on analysing if there are concrete mechanisms into these technologies to automatically tabulate the data, without having to produce large chains of concatenated operators.

Firstly, we analysed whether our problem can be solved using just SQL or a SQL-like language, such as JPL (*Java Persistence Language*). These languages typically provides efficient implementations of the *join* operator. Moreover, some database management systems, such as SQL Server, offers limited versions of the *pivot* operator [5]. For instance, the pivot version of SQL Server requires to know the number and names of the columns that will be added as a result of the operation in advance, instead of being discovered dynamically as we have described in the previous section. Consequently, scripts combining standard and proprietary SQL need often to be written by hand to convert data into a proper tabular format. Writing these scripts is exactly what we want to avoid.

Secondly, data warehouses store high volumes of data which can be consulted for reporting and analytical purposes [12]. These data are manipulated through multidimensional models based in facts and dimensions. Facts store quantitative measures around a business concept, while dimensions offer different perspectives, such as space and time, from which obtain and analyse fact measures. Languages for data warehouse management typically provide operators such as *drill-down*, *drill-up*, *slice* or *pivot*, among others. Nevertheless, they do not offer an operator that can execute a flattening process by itself. As before, by means of concatenating by hand different operators, a flattening process can be achieved, but we desire to make unnecessary the manual production of these concatenated operations chain.

Finally, the process of transforming an object-oriented domain model into another representation (e.g., relational, XML) has been largely studied in the literature [2, 8, 9]. These works typically specify a set of patterns that can be used to transform step-by-step from a model representation into another one.

Review
r_id : string
stars : double
text : string

r_id	stars	text
R1	4.5	We were recommended this by ...
...	...	...
R2	3.7	The first impression was not ...

Fig. 3. (Left) Main class to be transformed; (Right) The resulting tabulated data.

These patterns are the basis, for instance, of current *Object-Relational Mappers (ORM)*. Nevertheless, to the best of our knowledge, there is no pattern that can be used by itself to perform a flattening process. Again, by means of a careful combination by hand of these patterns, a flattening process can be implemented. However, as the reader might have noticed, this is not what we are looking for.

Next section shows how we have been able to provide an implementation for our desired flattening operator.

5 Flattening Operator Implementation

From an abstract point of view, the flattening operation can be viewed as the problem of transforming a set of interconnected objects into a tabular representation where, in addition, all information related to each instance of a specific class must be placed in the same row. In the following, we will refer to this specific class as the *main class* or the *main entity*.

Our strategy for implementing the *flattening operator* is based on reducing complex cases to a trivial case, whose implementation is straightforward. Reductions are achieved by means of operations, such as *joins* and *pivots*, over the data as well as by applying transformation patterns frequently used by *Object-Relational Mappers* [8]. Next subsections describe this reduction process with the help of the Yelp case study. However, the described patterns would work with any domain model under analysis.

5.1 Trivial Case: Single Class

In this case, the main class contains just simple attributes and it is not part of any inheritance hierarchy. A *simple attribute* is an attribute whose type is a basic type. Thus, the main class contains no references to other classes. Figure 3 (left) shows an example for this trivial case.

In this situation, the flattening operation goes as follows: first, we create a table with one column per each attribute contained in the main class. Then, each instance of the main class is placed in a single row, placing its attribute value within their corresponding columns. Figure 3 (right) depicts the result of applying the operator to a set of `Review` instances.

More complex flattening cases are reduced to this trivial case as described in next subsections.

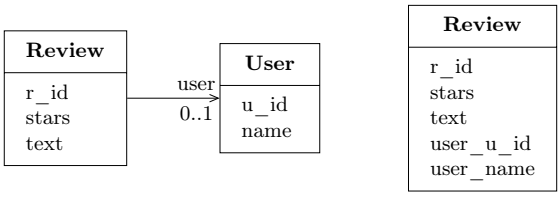

Fig. 4. One-bounded association.

5.2 One-Bounded Associations

This case happens when main class objects have a reference to a single object of another class, as shown in the Figure 4 (left). In addition to the *Review* class data, we want to include information of the user who has written each review.

As the upper bound of the reference is 1, to reduce this pattern to the trivial case, attributes of the referenced class can be simply copied into the main class, as if they were initially included in it. This can be easily achieved by means of a *join* operation between the main class and the referenced class. To avoid name collisions, the association's name is added as a prefix to each copied attribute.

Figure 4 shows how *User* attributes are included into the *Review* class, which later can be tabulated as described in the single class trivial case.

5.3 Unbounded Associations

In this case, objects of the main class refer to collections of objects of another class, instead of a single one as in the previous case. Figure 5 (left) shows an example which illustrates this case. We would like to analyse features influence in top-rated business, so we want to include information of the *Feature* class in the data analysis. For the sake of simplicity, we have skipped the inheritance hierarchy in which *Feature* class is involved (See Figure 1).

Because of the unbounded reference, the storage of multiple instances of the reference class must be allowed. Therefore, attributes of the referenced class will be included several times into the main class. Moreover, we would need a mechanism to distinguish between instances of the referenced class to, for instance, determine how many attribute copies must be included in the main class. This distinction mechanism would also allow to relate referenced instances from different instances of the main class, in order to place them into the same attribute set or, from the tabular format perspective, under the same column.

For this purpose, an attribute -or set of attributes- from the referenced class must act as identifier for their instances. This way, instances can be distinguished according to the value of their identifier, and information of referenced instances which share the same identifier can be placed into the same attributes set.

In order to make the reduction, we will perform a *pivot* operation. Linking with the terminology introduced in section 3, the attributes selected as identifier of the referenced class will act as *pivoting columns*, and the remaining attributes would be the *pivoted columns*.

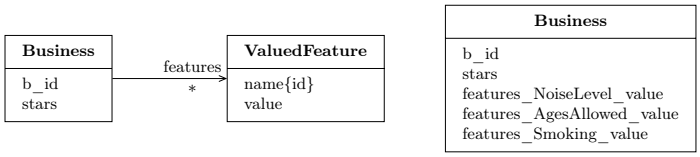

Fig. 5. Unbounded association.

As a result of the pivot operation, new sets of attributes would be added to the main class, one set per each distinct identifier found in the referenced objects. Each set will contain the attributes present in the *pivoted columns* set. This way, the information of the referenced instances gets condensed into each main class instance, added as a new set of attributes.

To avoid name collisions, attributes of each newly created attribute set are named according to the following pattern: <*referenceName*>_<*pivotingValues*>_ <*pivotedColumnName*>, where *referenceName* is the name of the reference, *pivotingValues* are values that can be found in the *pivoting columns* through which the pivot operation has been performed and, finally, *pivotedColumnName* is the name of the pivoted column.

For instance, in the case of Figure 5 (left), the attribute *name* from the class *ValuedFeature* is selected as identifier, which means that it will be used as *pivoting column*. Let us assume that {*NoiseLevel, AgesAllowed, Smoking*} is the set of distinct values for this attribute. Using this assumption, three new sets of attributes would be created, one per each distinct value. Each attribute set contains those attributes which will get pivoted. In this case, there is just an attribute to be pivoted, *value*, therefore only one attribute will be included on each set. The previously described pattern is used to denote the new attributes. For example, *features_NoiseLevel_value* represents the value of the *value* attribute for the feature *NoiseLevel* contained in the set of *features* of a specific *Business* instance.

Finally, it is worth to mention a special case of this pattern. It happens when all the attributes of the referenced class are used as identifiers, so there are no remaining attributes to be pivoted. One example of this situation would be the relationship between *Business* and *Category* (see Figure 6). In this case, the referenced class *Category* contains only the attribute *name*, which would be used as identifier. Therefore, there will not be any attribute to be pivoted, this is, the *pivotedColumns* set would be empty. In this particular case, a phantom attribute holding a boolean value will be pivoted. This attribute will indicate whether a particular instance of the referenced class appears or not in the collection of referenced objects attached to each instance of the main class.

For the case of Figure 6 (left), let us assume that {*buffet,dj,kosher*} is the set of all distinct names for the *Category* class. So, three different groups of attributes are created, one per each category. As there are no attributes to be pivoted, a boolean attribute is added to each group to indicate whether a specific *Business* belongs to a certain category, as shown in Figure 6 (right).

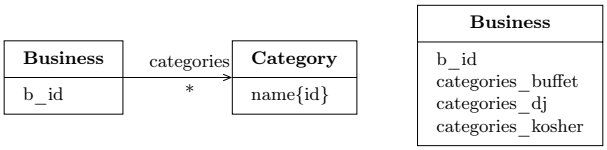

Fig. 6. Special unbounded case where all attributes from the reference are identifiers.

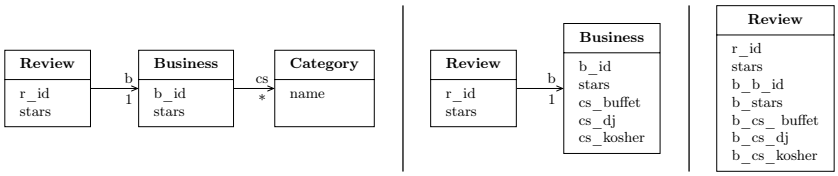

Fig. 7. Multilevel associations reduction.

5.4 Multilevel Associations

In this case, a referenced class has a reference to another class. An example is shown in Figure 7 (left)[1]. We would want to find patterns behind positive reviews, including information both from the reviewed *Business* and the *Category* each business belongs to.

It should be noted that the chain of references between classes has an unbounded length since, for instance, the *Category* class might have referenced another class and so on.

To reduce this pattern to the trivial case, the chain of references is recursively processed from tail to head. On each step, the deepest class is reduced using either the one-bounded or the unbounded pattern. After each step, the resulting chain of references is one level shorter than the previous one, so the process will end converging to the trivial case.

This process is illustrated in Figure 7. First, *Business* and *Category* are reduced to one class using the unbounded pattern (Figure 7 (middle)). Then, the resulting chain of references is reduced using the one-bounded pattern, reaching the trivial case (Figure 7 (right)).

5.5 Multiple Associations

This case happens when a class has several references to other classes. This pattern would appear if we combine Figures 5 and 6, because we want to analyse businesses considering category and feature information at the same time.

To reduce this pattern to the trivial case, each reference is reduced using the previous patterns. Since attribute ordering does not matter for the final tabular format, references might be processed in any order. Moreover, as references of a class must have different names, avoidance of name collisions is ensured.

[1] Reference names have been abbreviated for space reasons.

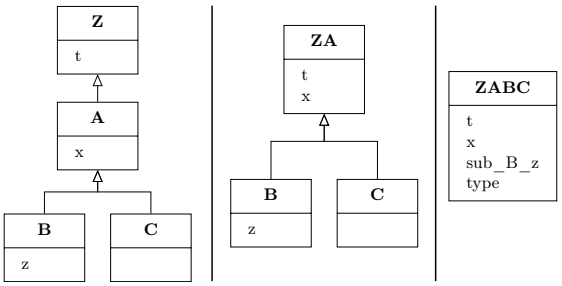

Fig. 8. Inheritance reduction by collapsing the attributes into main class A.

5.6 Simple Inheritance

In this section we will explain an inheritance reduction pattern which works for most found cases in a domain model. There exists two roles that inheritance can play in our patterns: (1) the main class belongs to an inheritance hierarchy, and (2) a referenced class resides in an inheritance hierarchy. These roles generate situations from which, through more advanced patterns, we can benefit of to achieve a more optimized reduction. We are currently working on these patterns but, for space and simplicity reasons, they have been left out of this paper.

The pattern performs inheritance reduction by *attributes collapse*. The inheritance gets compacted into a class from the hierarchy, including the attributes of all superclasses and subclasses of that class. This transformation process is inspired in the *single table* pattern used by Object-Relational Mappers [8].

The point of the inheritance from which we want to perform the reduction might be the root of the hierarchy, a leaf, or simply be in the middle. Figure 8 shows an example of this latter case, where class A is the main class. It has a superclass called Z, and two subclasses B and C. Reduction proceeds as follows:

- First, superclass attributes are included in the main class. For instance, attribute t from superclass Z is included into class A, conforming the temporary class ZA (Figure 8, (middle)). This is, attributes descend from the hierarchy root to the main class.
- Secondly, subclasses are folded towards the main class. Since we might need to tabulate information coming from any subclass, attributes of all subclasses are raised up to the main class. In Figure 8, main class instances can be of type A, B or C. Hence, it is necessary to include every attribute present in B and C into the class ZA, resulting in the type $ZABC$ (Figure 8, (right)). A special problem can be encountered when mixing attributes coming from different subclasses: their names may collide. To overcome this, attributes are renamed according to the following pattern: *sub_<className>_<attributeName>*. The *className* refers to the original class from which the attribute has been brought. In the example, attribute z, originally from class B, gets renamed to *sub_B_z* when placed in the final $ZABC$ class (Figure 8 (right)).

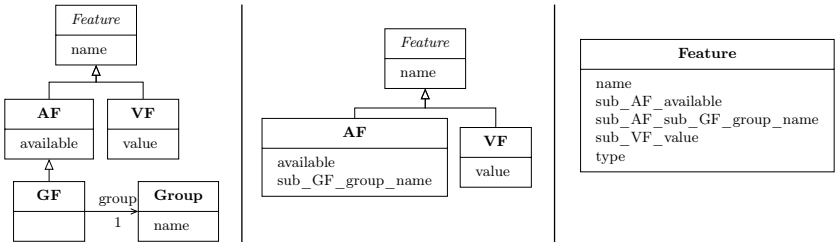

Fig. 9. Features inheritance reduction.

To be able to distinguish the type of each instance of the main class, a *type* attribute is included in the resulting class (see Figure 8 (right)). This attribute has as value the concrete type of each instance.

The other two extreme cases can be deducted from this general case, assuming that the main class has no superclasses (first step would be skipped) or that it has no subclasses (step two would be omitted). If multiple levels of inheritance are found, no matter their direction (up as superclasses or down as subclasses), they are evaluated recursively in the same manner as multilevel associations are, starting from the furthest to the main class one.

The reduction of the *Feature* inheritance from the domain model is shown in Figure 9. As before in other example, some names in the figure have been reduced for drawing purposes. A *Feature* can be found in three different flavours: *AvailableFeature* (AF), *ValuedFeature* (VF), and *GroupedFeature* (GF), which represents a special case of *AvailableFeatures* that relate conforming a group, therefore it has a reference to the *Group* class.

No superclasses have to be reduced in this case, as *Feature* locates in the root of the hierarchy (Figure 9 (left)). The reduction of the subclasses is depicted in two steps. First, attributes from *GroupedFeature* are merged into the *AvailableFeature* class (Figure 9 (middle)), being in this case the *name* obtained from the *group* reference. Then, both classes *AvailableFeature* and *ValuedFeature* are merged into the *Feature* class, and an additional *type* attribute is included (Figure 9 (right)). As the original *Feature* class was abstract, the set of values the *type* attribute might take would be {*AF, VF, GF*}.

6 Example: Usage in an Entities Extraction Language

Based on these patterns, we have developed a high-level language, *Lavoisier*[2], for automatically flattening fragments of a domain model. The language has been developed using *Xtext*[6]. This language offers a set of high-level primitives for specifying which parts of a domain model should be considered during a data analysis process. In *Lavoisier*, as conventionally in the data mining field, the tabular structure generated as a result of the flattening process is called a *dataset*.

[2] Current implementation is available at https://github.com/alfonsodelavega/lavoisier

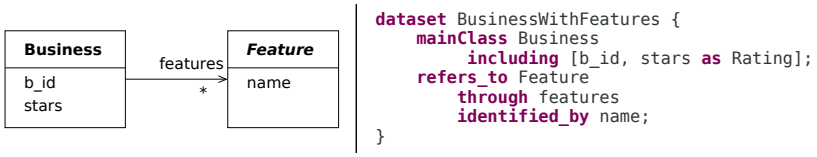

Fig. 10. Left. Fragment of the Yelp domain model. Right. A dataset specification written in *Lavoisier*

Figure 10 shows how *Lavoisier* can be used to solve the problem described in Section 2. The left of Figure 10 depicts the fragment of the domain model we wanted to use for a data analysis task.

Dataset definition starts by giving it a name, being in this case *BusinessWithFeatures* the introduced one. Then, the main class for building the dataset must be specified. *Business* class is selected as main class. Next, the set of attributes of the main class that will be included in the dataset are specified using the *including* keyword. This step is optional, and if this clause is omitted, all attributes of the main class would be included. Moreover, aliases for some attributes can be specified if it is desired through the keyword *as*.

Finally, classes which are referenced from the main class can also be included in the dataset by means of the *refers_to* keyword. In our case, the *Feature* class would be added through the *features* reference. As this is an unbounded reference, there exists the need to define the attributes of the class *Feature* that will be used as pivoting attributes. This is done with the keyword *identified_by*, which is used to select *name* as pivoting attribute in our case. The tool automatically obtains the set of *name* values that will be used as ids in the unbounded pattern.

It is worth to point out that the *Feature* class is included into an inheritance hierarchy. Therefore, as previously described, advanced transformations patterns would be required to correctly tabulate each instance depending on its type. However, the *Lavoisier* user does not need to know the transformations details, as the language will execute them transparently.

Through Xtext usage, we obtain a very capable editor, with easy inclusion of terms proposal and validation into the language. The user gets assisted through the dataset definition process, which is checked against the existent domain model to ensure correctness and to provide useful suggestions.

Next section recapitulates the contributions and concludes this paper.

7 Summary and Future work

This paper has presented, as main contribution, a set of patterns for automatically transforming a selection of interconnected objects, conforming to a domain model, into a particular kind of tabular format. This transformation process, named as a flattening operation, is mandatory when using these data as input for a data mining algorithm. These patterns has been integrated into a high-level language, called *Lavoisier*. Working with *Lavoisier*, the user just specifies,

using a set of high-level primitives, which part of a domain model should be considered for a data mining task, and the language automatically rearranges the corresponding data into an appropriate tabular format. This avoids that large and complex scripts to accomplish this task have to be created by hand, saving time and reducing errors.

In future works, we will perform a comprehensive description of *Lavoisier* capabilities, as well as the inclusion of more data selection mechanisms, such as aggregation functions or row filters. Moreover, the patterns will be formalized, which will allow us to develop further patterns and an easier study of how to work with other representations, such as entity-relationship or RDF models.

Acknowledgements. This work has been partially funded by the Government of Cantabria (Spain) under the doctoral studentship program from the University of Cantabria, and by the Spanish Government under grant TIN2014-56158-C4-2-P (M2C2).

References

1. Abadi, D., et al.: The beckman report on database research. SIGMOD Rec. 43(3), 61–70 (Dec 2014), http://doi.acm.org/10.1145/2694428.2694441
2. Atzeni, P., Cappellari, P., Torlone, R., Bernstein, P.A., Gianforme, G.: Model-independent schema translation. VLDB Journal 17(6), 1347–1370 (2008)
3. Beller, M., Gousios, G., Zaidman, A.: Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: Proceedings of the 14th working conference on mining software repositories (2017)
4. Cao, L.: Domain-Driven Data Mining: Challenges and Prospects. IEEE Transactions on Knowledge and Data Engineering 22(6), 755–769 (jun 2010)
5. Cunningham, C.: PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. Proceedings of the 30th International Conference on Very Large Data Bases pp. 998–1009 (2004)
6. Eysholdt, M., Behrens, H.: Xtext: Implement Your Language Faster than the Quick and Dirty Way. In: Companion to the 25th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA). pp. 307–309. Reno/Tahoe (Nevada, USA) (October 2010)
7. Fayyad, U., Piatetsky-Shapiro, G., Smyth, P.: From data mining to knowledge discovery in databases. AI magazine 17(3), 37 (1996)
8. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
9. Hainaut, J.L.: The Transformational Approach to Database Engineering. In: Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, pp. 95–143. Springer (2006)
10. Hall, M., et al.: The WEKA Data Mining Software: An Update. SIGKDD Explorations Newsletter 11(1), 10–18 (June 2009)
11. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2013)
12. Wrembel, R., Koncilia, C.: Data Warehouses And Olap: Concepts, Architectures And Solutions. IRM Press (2006)
13. Yelp: Yelp Dataset Challenge Round 9. https://www.yelp.com/dataset_challenge, [Online; accessed 30-March-2017]