Proceedings Paper:
De La Vega, Alfonso, Sanchez, Pablo and Kolovos, Dimitrios S. orcid.org/0000-0002-1724-6563 (2018) Pinset : A DSL for extracting datasets from models for data mining-based quality analysis. In: Proceedings - 2018 International Conference on the Quality of Information and Communications Technology, QUATIC 2018. 11th International Conference on the Quality of Information and Communications Technology, QUATIC 2018, 04-07 Sep 2018 Proceedings - 2018 International Conference on the Quality of Information and Communications Technology, QUATIC 2018 . Institute of Electrical and Electronics Engineers Inc. , PRT , pp. 83-91.

https://doi.org/10.1109/QUATIC.2018.00021

Pinset: A DSL for Extracting Datasets from Models for Data Mining-Based Quality Analysis

Alfonso de la Vega
Software Engineering and Real-Time
Universidad de Cantabria
Santander, Spain
alfonso.delavega@unican.es

Pablo Sánchez
Software Engineering and Real-Time
Universidad de Cantabria
Santander, Spain
p.sanchez@unican.es

Dimitrios S. Kolovos
Department of Computer Science
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

***Abstract*—Data mining techniques have been successfully applied to software quality analysis and assurance, including quality of modeling artefacts. Before such techniques can be used, though, data under analysis commonly need to be formatted into two-dimensional tables. This constraint is imposed by data mining algorithms, which typically require a collection of records as input for their computations. The process of extracting data from the corresponding sources and formatting them properly can become error-prone and cumbersome. In the case of models, this process is mostly carried out through scripts written in a model management language, such as EOL or ATL. To improve this situation, we present Pinset, a domain-specific language devised for the extraction of tabular datasets from software models. Pinset offers a tailored syntax and built-in facilities for common activities in dataset extraction. For evaluation, Pinset has been used on UML class diagrams to calculate metrics that can be employed as input for several fault-prediction algorithms. The use of Pinset for this calculations led to more compact and high-level specifications when compared to equivalent scripts written in generic model management languages.**

***Index Terms*—Data Mining; Software Quality; Model-Driven Engineering; Domain-Specific Languages**

I. Introduction

Data mining techniques [1] are being employed to improve different aspects of software quality assurance processes [2]–[5]. Among other issues, these techniques are being used to: (1) predict the existence of software *bugs* [2]; (2) detect patterns or *smells* that might affect software quality [3]; (3) obtain intelligent metrics that provide better insights for quality analysis [4]; and (4) improve the efficiency of mutant execution for analysing the quality of a test suite [5]. These techniques are applied to different kinds of software artefacts, including source code [6] and test reports [7]. Software models are also included among these artefacts [8].

In general, the objective of these approaches is to develop prediction models, which can help in the automatic detection of complex or hidden issues that might affect the quality of a software product. These prediction models are constructed by different algorithms, which use information extracted from existing software repositories and based on a curated history of previous software projects and their corresponding outcomes. These algorithms require the provided data to be formatted as a two-dimensional or tabular dataset. Several examples of these datasets can be found in D'Ambros et al. [6], who made publicly available metrics and historical data about five open source software systems, so that they could be used to, for instance, train fault-detection predictors for software products.

To build such datasets, software engineers write scripts that access software repositories, retrieve the required information, execute computations to calculate some metrics and, finally, arrange all the gathered data in a tabular form. In the case of models, these scripts are written using model management languages, such as OCL [9], ATL [10] or those provided by Epsilon [11]. As further discussed below, the development of these scripts can be a tedious and cumbersome process.

To improve this situation, we present Pinset, a Domain-Specific Language (DSL) for the generation of tabular datasets from models. Pinset offers high-level primitives that simplify the specification and computation of datasets, which can be used as inputs for data mining algorithms. The use of Pinset leads to more high-level and concise specifications of data acquisition processes when compared to existing alternatives.

Pinset is implemented as an extension of the Epsilon [11] suite, making use of the facilities this platform provides, such as OCL-like expressions. Using this implementation, we created scripts to build datasets from UML class models. These datasets can be used for fault prediction analysis by following several approaches available in the current literature [6], [12].

As input for these scripts, we used third-party UML models coming from a large public repository [13]. The scripts that make use of Pinset are more compact than their counterparts written in other languages, such as ATL or ETL [14]. This is mainly due to the use of high-level primitives specifically tailored for the data acquisition and transformation tasks.

The rest of the paper is organised as follows. Section II extends the motivation behind this work. Then, Section III shows how to extract datasets with state-of-the-art tools. The functionality and implementation of Pinset are described in Sections IV and V, respectively. Pinset is compared against a generic model transformation language in Section VI. Finally, Section VII comments on future work and concludes the paper.

II. Motivation and Running Example

This section details the motivation behind this work, describing first how our approach might be used inside a specific data-mining process for model quality analysis. Then, we

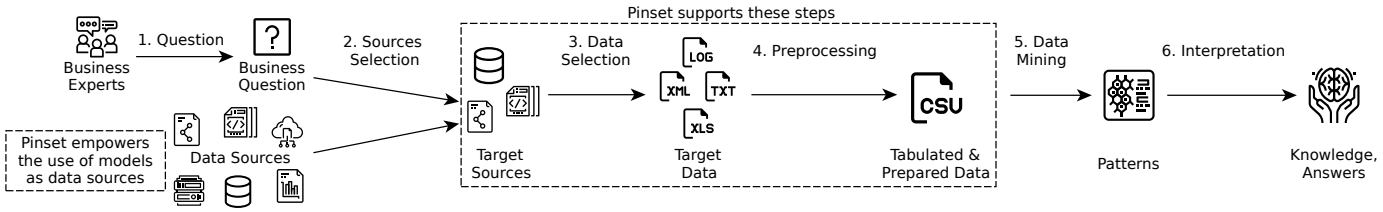

Fig. 1. Role of Pinset inside a generic data mining process. Source: [15]; expanded for clarity. Icons designed by Smartline From Flaticon.

introduce the running example that is used throughout the paper to describe the features of Pinset.

A. Data Mining Processes

A data mining process can be defined as a succession of steps, where data are analysed with the objective of discovering patterns that can be used for different purposes, such as predicting future behaviours [15]. Figure 1 depicts a generic data mining process. The process starts with a business question that must be answered (Figure 1, Step 1). In a software development context, it could be asked whether those modules whose maintenance tasks take more time to finish share some features that might be causing these longer durations.

Then, a set of data sources to answer these questions must be selected (Figure 1, Step 2). In the case of software quality analysis, these data sources might include source code hosted in a repository, modeling artefacts, requirements specifications or bug reports, among others. For instance, to find if there are features that are common to those modules whose maintenance effort is larger, we might decide to analyse information obtained from the source code of these modules, as well as data retrieved from the bug tracking system used to manage the maintenance tasks.

As a third step, specific data are selected and extracted from these sources. For instance, several parameters of a maintenance task, such as number of modified classes, number of added lines of code (LOC), number of removed LOC, and time spent to complete the task might be retrieved from the bug tracking system.

In the preprocessing step (Figure 1, Step 4), two different tasks are carried out. Firstly, to be used as input for data mining algorithms, the gathered data need to be arranged as records of information (rows), where each record provides values of a set of properties (columns) from an instance of the entities being analysed. That is, each record represents values for some features of, for instance, a class or a maintenance task, depending on which elements we are analysing in each case. Secondly, the selected data is adapted to any specific requirements of the applied algorithms. For instance, some numeric values might need to be normalised to fit between 0 and 1, whereas other values might need to be discretised prior to being used as input of a specific algorithm.

Next, data mining algorithms are executed (Step 5). The results of these algorithms are analysed to reach some conclusions, which is accomplished in the *interpretation* phase (Step 6). These conclusions should help answer the initial business question, which might assist in decision making.

As shown in Figure 1, Pinset, the language described in this work, aims to help with the data acquisition and transformation phases (steps 3 and 4, respectively) when the sources from which data are retrieved are well-defined models, this is, models following a model-driven perspective.

TABLE I
OBJECT-ORIENTED METRICS USED THROUGHOUT THE PAPER EXAMPLES.

Metric	Description
CK_WMC	Weighted method count
CK_DIT	Depth position on the inheritance tree
CK_NOC	Number of children
CK_CBO	Coupling between objects
OO_FanIn	Number of other classes that reference the class
OO_FanOut	Number of other classes referenced by the class
OO_NOF	Number of features
OO_NOA	Number of attributes
OO_NOPA	Number of public attributes
OO_NOPRA	Number of private attributes
OO_NOIA	Number of inherited attributes
OO_NOM	Number of methods
OO_NOPM	Number of public methods
OO_NOPRM	Number of private methods
OO_NOIM	Number of inherited methods

B. Running example

To develop our language, we were inspired by the work of Osman et al. [16]. This work evaluates how different automatic feature selection techniques affect the accuracy of bug predictors. A *bug predictor* is a tool that uses a data mining process to determine whether a piece of code might be a potential source of bugs, usually within some confidence range. In this case, the business question is whether a class can be considered as potentially buggy or not.

As input for the bug predictors, Osman et al. rely on an external dataset, which is provided by D'Ambros et al. [6] and which has been widely used in the bug prediction literature [17], [18]. This dataset collects data at the class level, by analysing source code hosted in software repositories augmented with some change metrics coming from other sources, such as configuration management systems. Therefore, the main entities under analysis are classes. For each class, a set of metrics, including the ones defined by Chidamber and Kemerer [12], are calculated. Table I lists these metrics.

However, these works usually omit the process through which the values of the datasets are calculated. It is assumed

that a script crawls a software repository, extracts the data and formats it. Pinset aims to complement all these works, by providing mechanisms to retrieve and format these data more easily when the data sources are models. To illustrate how Pinset works, we will show how it can be used to extract the metrics contained in Table I from UML class models. It should be noted that some of these metrics do not apply to the model level, so they have been adapted or just skipped. These adaptations are commented in the paper when required. Moreover, other metrics are used in some specific places, to better illustrate some of the features offered by Pinset.

III. Related Work

In the modeling community, the extraction of metrics from models is typically accomplished by means of generic model transformation/management languages, such as OCL, ATL or Epsilon's EOL/ETL. Indeed, the ATL documentation, which includes a wide range of model-to-model (M2M) transformation examples, contains a specific entry for a table extraction scenario[1]. In that example, the model of a Java program is transformed into an instance of a metamodel that represents a table. It is assumed that a code generation process is executed next, where the model is transformed into an appropriate textual format, such as CSV (Comma Separated Values).

Since Pinset is implemented over the Epsilon platform, for the sake of consistency, we will show how a similar transformation can be expressed in ETL [14], i.e., the M2M transformation language of Epsilon. This transformation takes place in the context of the running example introduced previously, that is, it will extract a dataset from a UML class diagram. We have slightly modified the original table metamodel of the ATL transformation to (1) allow the definition of several datasets in the same model; and (2) to explicitly store the column headers of each dataset, as usual in the data-mining community. This new metamodel is depicted in Figure 2.

A model conforming to this metamodel contains *Dataset* instances, i.e., one or more datasets. A *Dataset* is composed of a set of *Column* headers, plus a set of *rows*. Each *Row* stores *Cell* values, one for each column of the dataset. Each cell indicates to which column it corresponds. The relationship between *Cell* and *Column* might be avoided by imposing an order both to the cells of each row and to the column headers. This way, cells in a certain position inside a row would correspond to the column header in the same position. Nevertheless, this solution makes instances and model transformations harder to maintain because of the required attention to ordering. For instance, if we removed a column of a dataset, we would need to update, in addition to the values of that column, how the values of subsequent columns are assigned, since these values would now correspond to a lower position. Keeping each cell associated to its column avoids this problem.

Listing 1 shows how some basic metrics of a class can be computed using ETL. This transformation extracts the following information from each class: (1) its name; (2) whether it is abstract; (3) the name of its parent class, if any; (4) its number of attributes (*NOA*); (5) its number of methods (NOM); (6) its number of features, i.e., the sum of attributes and methods; and (7) its depth inside an inheritance tree (DIT).

[1]https://www.eclipse.org/atl/atlTransformations/Java2Table/ExampleJavaSource2Table[v00.01].pdf

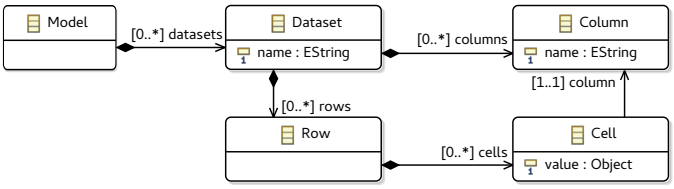

Fig. 2. Dataset Metamodel used as output in the M2M transformations.

Listing 1. ETL transformation that extracts basic class metrics.

```
1  pre {
2    var modelRoot = new Dataset!Model();
3    var metricsDataset = createDataset("BasicMetrics");
4    modelRoot.datasets.add(metricsDataset);
5    var bmd_c_name = createColumn("name");
6    var bmd_c_isAbstract = createColumn("isAbstract");
7    var bmd_c_parentName = createColumn("parentName");
8    var bmd_c_OO_NOA = createColumn("OO_NOA");
9    var bmd_c_OO_NOM = createColumn("OO_NOM");
10   var bmd_c_OO_NOF = createColumn("OO_NOF");
11   var bmd_c_CK_DIT = createColumn("CK_DIT");
12   metricsDataset.columns =
13   Collection {bmd_c_name, bmd_c_isAbstract,
14     bmd_c_parentName, bmd_c_OO_NOA,
15     bmd_c_OO_NOM, bmd_c_OO_NOF,
16     bmd_c_CK_DIT};
17 }
18
19 rule BasicMetricsClass2Row
20 transform class : Model!Class
21 to row : Dataset!Row {
22   metricsDataset.rows.add(row);
23   row.cells.add(createCell(bmd_c_name, class.name));
24   row.cells.add(createCell(bmd_c_isAbstract,
25   class.isAbstract));
26   var parentName = "";
27   if (not class.superClass.isEmpty()) {
28     parentName = class.superClass.first().name;
29   }
30   row.cells.add(createCell(bmd_c_parentName,
31   parentName));
32   var OO_NOA = class.attributes.size();
33   var OO_NOM = class.operations.size();
34   var OO_NOF = OO_NOA + OO_NOM;
35   row.cells.add(createCell(bmd_c_OO_NOA, OO_NOA));
36   row.cells.add(createCell(bmd_c_OO_NOM, OO_NOM));
37   row.cells.add(createCell(bmd_c_OO_NOF, OO_NOF));
38   row.cells.add(createCell(bmd_c_CK_DIT, class.dit()));
39 }
40
41 operation Class dit(): Integer {
42   var dit = 0;
43   var node = self;
44   while (not node.superClass.isEmpty()) {
45     node = node.superClass.first();
46     dit += 1;
47   }
48   return dit;
49 }
```

First, the dataset and its columns are instantiated in a *pre* block (lines 1-17). In ETL, a *pre* block contains code that is executed before any transformation rule. These blocks usually set up certain elements that need to be configured before running the transformations. In our case, the use of a *pre* block makes the defined datasets globally accessible during the execution process, which allows the transformation rules

to populate them with rows. Thus, in lines 2-4 a new dataset called *BasicMetrics* is defined and added to the model root element. Then, in lines 5-16, the columns of the *BasicMetrics* dataset are defined and assigned. For the creation of datasets and columns, we rely on helper functions named accordingly (*createDataset*, *createColumn*). For the sake of simplicity, these helper functions have been left out of this listing, but they can be consulted in an external repository[2].

Once we have created the dataset schema, it is populated by defining a transformation rule. The rule transforms each instance of the selected model type to a row in the dataset (lines 19-39). In our case, for each *Class* in the input metamodel, a new *Row* is created in the target metamodel (lines 20-21). This row is added to the *BasicMetrics* dataset (line 22), and then it is populated with a cell for each column. Thus, first of all, the name of the class is extracted (line 23) and assigned to the corresponding column header by means of the *createCell* helper function. Then, in the same way, the value for the *isAbstract* column is taken from the *isAbstract* class attribute (lines 24-25). Next, the *parentName* column value is computed (lines 26-31). To do it, we check if the class has a superclass. If so, the corresponding name is extracted from the parent class, otherwise this value is set to an empty string. Other values, like *NOA*, *NOM* or *NOF* are obtained using the same techniques (lines 32-34). Finally, to calculate the *DIT* value for each class (line 38), due to its complexity, we have opted for extracting the code that computes it to an external function (lines 41-49).

We can see how the management of the dataset structure introduces extra verbosity and complexity in the extraction process. This verbosity obfuscates the final goal of the transformation, which is how elements from the input model get transformed into rows of the resulting datasets. We tried to alleviate this obfuscation by defining some helper functions and providing several global variables. This kind of variables, as it is known, might lead to some undesired side effects.

It is worth mentioning that the described problems are not due to the use of a transformation language. If we have opted for using a general-purpose programming language such as Java, the problem would be worse, since this language does not offer facilities for manipulating models.

Therefore, after experimenting with the current state-of-the-art solutions, we came up with the following idea: to create a domain-specific language that provides high-level primitives for the extraction of metrics and values from models. With such a language, we could avoid the obfuscation and verbosity witnessed when using a general-purpose model transformation language for the dataset extraction task.

With this idea in mind, we checked the literature to know whether this objective had been already addressed. The closest approach is the work of López-Fernández et al. [19], who provide two languages for quality assurance in metamodels. These languages allow developers to specify constraints over EMF metamodels that check different issues, such as, for instance, that a design rule is not violated, or that a specific quality indicator does not exceed a certain threshold.

These languages provide some high-level primitives to navigate through a metamodel, and to select sets of metamodel elements that satisfy some conditions. The languages also support the computation of metrics, but they are not devised to construct datasets. Therefore, building datasets using these languages is not a straightforward task. Moreover, these languages only work at the metamodel level, so they cannot be applied to model instances or to other kinds of models, such as state machines or business process models. With this context, we borrowed some ideas from this work and we decided to develop *Pinset*, a domain-specific language for extracting datasets from models. The language is presented in the following section.

[2]https://github.com/alfonsodelavega/pinset-examples/blob/master/es.unican.istr.pinset.examples.etlComparison/etl/basicClassMetrics.etl

IV. Solution Description

We start by presenting a basic Pinset example, followed by more in-depth descriptions of some Pinset features.

A. Syntax Overview

Listing 2 shows the ETL script of Listing 1 rewritten using Pinset. As it can be seen, in Pinset, a dataset is specified as a set of column definitions that capture data from instances of a type included in an input model. These definitions might have different flavours: in their most basic form, they allow to specify a column declaratively through OCL-like expressions. However, if a column is complex enough to require several steps to calculate it, Pinset offers imperative language structures, such as conditions, loops or calls to external functions, to achieve this task. Moreover, Pinset also provides high-level constructs, not included in this first example, to facilitate the definition of complex columns.

Column definitions are organized in *dataset rules*. In its simplest form, a dataset rule consists of (i) a name, (ii) a typed parameter, and (iii) a set of column generators. The name identifies the dataset. The typed parameter specifies which type of the input model is going to be processed when populating the rows of the output dataset. This means that a row will be generated in the resulting dataset each time a new instance of the specified type is found in the input model. The column generators are used to define the columns that the final dataset will have, and how the values of these columns are calculated.

Listing 2. A Pinset dataset rule that extracts the metrics of Listing 1.

```
1  dataset basicClassMetrics over class : Class {
2    column name : class.name
3    column isAbstract : class.isAbstract
4    column parentName {
5      var name = null;
6      if (not class.superClass.isEmpty()) {
7        name = class.superClass.first().name;
8      }
9      return name;
10   }
11   column OO_NOA : class.attributes.size()
12   column OO_NOM : class.operations.size()
13   column OO_NOF : OO_NOA + OO_NOM
14   column CK_DIT : class.dit()
15 }
```

Listing 2 provides a dataset rule example, which is denoted as *basicClassMetrics*. Its parameter name is *class* (line 1), whose type is *Class*. This parameter is specified using the *over* keyword. The type selection means that a row will be created in the target dataset each time a Class instance is found in the input model. Finally, lines 2-14 contain the column generators that specify the contents of the dataset.

Different generators can be used to define the columns of a dataset. In this first example, the *Column* generator is employed. This generator requires a *name*, which defines the header of the column to be generated; plus a piece of code that specifies how this column must be calculated for each instance of the typed parameter. This piece of code can be defined using different styles. In the simplest version, column values are obtained through an EOL expression. EOL (Epsilon Object Language) [20] is an OCL-like language from the Epsilon suite, with capabilities for manipulating models conforming to a metamodel structure. The expression is invoked over each instance of the selected type that is found in the input model.

Listing 2, line 2 shows an example of the *Column* generator. This example specifies that the output dataset has a column called *name*, which contains the name of each class, retrieved through the expression *class.name*. It should be noticed that these EOL expressions have access to the instance being processed through the name of the dataset parameter, which is *class* in this case. A similar strategy is applied in the second column definition (line 3).

As commented, more complex expressions can be employed if needed. The column *parentName* is defined with an EOL block (lines 4-10), which is composed of a set of instructions, and ends returning the value that will be used to populate the column. In the example, the parent name is searched through the *superclass* feature of the class. If it is not found, a blank name is returned.

Column values are calculated in the same order they appear in the dataset rule. This way, previously calculated column values can be used in new column definitions. For instance, two columns holding the number of attributes (*OO_NOA*) and methods (*OO_NOM*) of the class are declared first (lines 11 and 12 respectively). Then, in line 13, the number of features (*OO_NOF*) is obtained through the sum of these two previously calculated values.

Finally, line 24 shows that it is possible to call external functions from column expressions. The depth of inheritance (*CK_DIT*) metric is obtained through the external *dit()* operation, which is defined outside of the rule. The definition of this operation has been omitted here, as it is present in Listing 1.

As output, the execution of the rule from Listing 2 generates a CSV file for each specified dataset rule. Each generated file has the same name as its corresponding dataset. In our example, a *basicClassMetrics.csv* file is created. The first row of this file contains the name of the defined columns separated by commas. Then, a row is included for each element of type *Class* in the input UML model. Each row contains appropriate values for their columns, being these values separated with commas. As it can be noticed, Pinset provides the final datasets in one step, unlike the M2M and M2T transformation process of the previously described state-of-the-art approach.

The following sections describe more advanced mechanisms provided by the language.

B. Properties Accessors

When we want to define columns that only hold values coming from properties of the selected type, the *Column* generator syntax can become too verbose and redundant. For example, in the *name* and *isAbstract* column definitions of Listing 2 (lines 2 and 3), the name of the column matches the name of the retrieved property. Therefore, this name could be easily deducted from that property, such as in the expression `column name : class.name`. For these cases, Pinset provides shorthand constructs that allow to define columns for simple properties in a more concise way.

Listing 3. Dataset extraction that employs *properties* and *reference* helpers.

```
1  dataset classBasicInfo over class : Class {
2    properties [name as class_name, isAbstract]
3    reference package[name]
4    ...
5  }
```

Listing 3 shows how this syntactic sugar can be used. In line 2, the *properties* generator selects some properties from the processed type to be included as columns. Property names must be indicated between square brackets, separated by commas. For each property, a new column with the same name is created, which holds the value of that property for each processed element. These properties must hold values of a primitive type, not being possible to apply this generator over references to other model types. In our example, the *name* and *isAbstract* properties of a class are selected to become columns of the target dataset. It is also possible to modify the final column names with an alias and the *as* keyword. In the example, the *name* property is renamed to *class_name*.

To include some information about types related to the processed one, we can use the *Reference* generator. This construct receives the name of a reference of the processed type, and a set of properties from that reference. The generator creates a new column for each specified property of the reference, and the values of these columns will be simply obtained from the corresponding properties, as before. If no alias is provided, the name of the columns is obtained by combining the reference and property names with an underscore. In our example, this construct is used to include the name of the package that contains the class (Listing 3, line 3). As a result, a new column denoted *package_name* is created, which stores the name of the package to which each class belongs.

The presented generators automatically manage any presence of *null* values in the model. If a property is not present, a blank value is inserted instead. In the same way, if the reference of an element points to null, blanks are inserted for all included properties of that reference.

C. Row Filtering Options

In the previous examples, datasets contain one row for each instance of the type being processed. However, it could be the

case that we are not interested in processing all these instances, but a subset of them. Pinset offers two alternatives to perform instances filtering: (1) specifying a *guard* condition; or (2) declaring a *from* expression. One alternative might be more suitable than the other, depending on the characteristics of the filtering process. Listings 4 and 5 illustrate both alternatives, respectively. For the sake of simplicity, column definitions have been omitted from the listings.

Listing 4. Selection over the type elements of a dataset with a guard.

```
1  dataset classSelectionGuard over class : Class {
2    guard : class.isAbstract
3    ...
4  }
```

A *guard* is a condition declared with the *guard* keyword followed by a boolean expression (Listing 4, line 2). This expression is evaluated over each instance of the corresponding type. Those instances that do not match the condition are discarded, and therefore no row is generated for them in the dataset. As an example, in Listing 4, the guard specifies that only abstract classes must be processed.

Listing 5. Selection over the type elements of a dataset with from.

```
1  dataset classSelectionFrom over class : Class
2      from : Class.all.select(c | c.isAbstract) { ... }
```

Another way of performing the same selection is shown in Listing 5, which employs a *from* clause. This clause explicitly indicates the collection of elements to be used in the creation of the dataset. The clause is declared with the *from* keyword, followed by an expression that returns the mentioned collection of instances. Listing 5 provides an example of this use case. The from expression calculates the set of all abstract classes in the model (line 2).

It should be noted that, when the *from* clause is used, the dataset rule may not need to access the input model to search for instances of the selected type if, for instance, the collection of instances has been previously calculated in a *pre* block. Therefore, in those cases where the same subset of instances is used as input for several dataset rules, the *from* option might be preferred over a *guard* for performance reasons, since this subset would be calculated just once for all rules.

Finally, it is worth pointing out that both mechanisms are not mutually exclusive, and can be applied in combination. When combined, the guard condition is evaluated over the collection of elements provided by the from clause, so the guard is used in this case to filter that collection. This feature might be used to obtain refined datasets after performing some preliminary analysis over a broader dataset. For instance, a first analysis over all classes in a package might indicate that there could be a problem affecting only the abstract classes of that package. A second analysis could focus just in these abstract classes, in order to investigate the roots of this problem more accurately.

D. Multiple Columns Definition: Grid

In some cases, we detected that sets of columns were defined with an almost identical expression. A typical example of this situation happens when we calculate the same metric for different values of a concrete property. For instance, we might be interested in knowing the number of attributes of a class for each visibility modifier, i.e., number of public, private, protected or package attributes. Table II shows the header of a dataset containing this information. The columns of this dataset are the name of the class and one column per each possible visibility modifier. These last columns register the number of attributes that each class has with that visibility.

TABLE II
CLASS ATTRIBUTES COUNT, GROUPED BY VISIBILITY.

name	#public_attrs	#protected_attrs	#private_attrs	...
User	0	5	2	...
Seller	1	0	4	...
...	...	...	...	...

The one-by-one definition of these columns, for instance by using the *Column* generator, becomes redundant, as the expression that counts the attributes is identical except for the visibility modifier that is considered for each case. Therefore, these expressions might be abstracted by converting this variable element into a parameter.

This can be achieved using the *Grid* generator, which allows defining multiple columns over the same expression. This generator creates a set of columns based on a collection of elements, denoted as *keys*, which is specified by means of an EOL expression. Each key is then processed to generate a column, based on two extra components: (1) a *header*, which determines the name of each column being generated: and, (2) a *body*, which contains the piece of code that calculates the value for each generated column. Both header and body expressions can access to the key being processed through the *key* variable name.

Listing 6. Grid example that generates the dataset of Table II.

```
1  dataset attributesByVisibility over class : Class {
2    properties [name]
3    grid {
4      keys : Sequence{UML!VisibilityKind#public,
5        UML!VisibilityKind#protected,
6        UML!VisibilityKind#private,
7        UML!VisibilityKind#package}
8      header : "#" + key + "_attrs"
9      body : class.attributes
10                .select(a | a.visibility = key).size()
11   }
12 }
```

Listing 6 shows a dataset rule that uses a grid generator (lines 3-11) to create the dataset of Table II. In this case, the keys collection is specified as a sequence of literals (lines 4-7). These literals represent all visibility modifiers available in the UML metamodel. Next, the *header* specifies that the name of each column will be generated with the name of the visibility modifier being processed, accessed by the *key* parameter, prefixed with "#" and ended with "_attrs" (line 8). Finally, for each instance of the type being processed, i.e., *Class* in our case, the *body* is evaluated for each *key* to calculate the number of attributes each class has of each kind of visibility (lines 9-10).

It should be noticed that a grid can be used to generate datasets with a variable number of columns, which depends

on the contents of the input model. For instance, we might want to calculate the number of dependencies from each class in a model to classes in that model that are placed in other packages. In this case, we would calculate a collection containing all existing packages in the input model. This collection would be used as the keys of a grid, so that dependencies to classes from each package can be easily calculated in the column corresponding to each key.

E. Typeless Dataset Rules

In the examples we have shown in previous sections, the datasets were created over a type from the model, precisely, UML classes. However, it is possible that, instead of placing the data from each class in a row of the final dataset, we may want to aggregate the data of certain classes according to some grouping criteria. For instance, we might be interested in knowing the number of classes that define less than one, five, or ten attributes; or the ones with more than one, five or ten methods. The natural way of defining these conditions would be to declare a set of thresholds, e.g., $\{1, 5, 10\}$, and then calculate the aggregations for each threshold.

To perform aggregations in Pinset, we can use the *from* expression in a typeless dataset rule. In section IV-C, this construct was presented as a row filtering mechanism, where it provided the list of elements that were to be transformed into rows. The process now is the same: the *from* expression provides the list of elements used to generate rows, but these elements are not restricted to a type from the model.

Listing 7. Typeless rule that counts the number of classes that fulfil a set of conditions. The conditions are defined over the threshold parameter.

```
1  dataset thresholdMetrics over threshold
2      from : Sequence{0,1,2,5,10} {
3    column threshold : threshold
4    column classes_w_NOA_leq_th : allClasses.select(
5        c | c.attributes.size() <= threshold).size()
6    column classes_w_NOM_leq_th : allClasses.select(
7        c | c.operations.size() <= threshold).size()
8    column classes_w_FanIn_geq_th : allClasses.select(
9        c | c.fanIn().size() >= threshold).size()
10   column classes_w_FanOut_geq_th : allClasses.select(
11       c | c.fanOut().size() >= threshold).size()
12 }
```

Listing 7 shows the thresholds example in a typeless dataset rule. The rule counts the number of classes that fulfil different properties. The parameter of the rule, which does not have a type in this case, holds the *threshold* value (line 1). This parameter will iterate over the values of the collection provided by the *from* expression, which in this case is a sequence of integers (line 2). The dataset contains four metrics that seek to estimate the size of the classes in the diagram, both in number of features and in the amount of relationships with other classes. The first two calculate how many classes have at most as many attributes (*NOA*) and methods (*NOM*) as the threshold, respectively (lines 4-7). The last two count the number of classes whose *FanIn* and *FanOut* metrics (see Table I for details) are greater than or equal to the threshold (lines 8-11).

The result of Listing 7 rule is a dataset containing five rows (one for each threshold value) and five columns, which contain the threshold value of the row and the results of the aggregation expressions for that value.

F. Column Post-Processing

Pinset allows performing transformations to the columns of a dataset after they have been calculated. For instance, some data mining algorithms do not allow null values in a column. A typical transformation involves filling these null values with something, e.g., a default value, the mean of the column, or the mode. Another example is the normalization of a numeric column to, for instance, comprise its values in the 0 to 1 range. This is necessary when comparing numeric columns that may have a different scale, e.g., *age* and *numberOfChildren* columns of a person dataset should be normalized. These transformations are done in Pinset through column annotations. Listing 8 shows some of these annotations.

Listing 8. Post-processing nulls filling and normalization examples.

```
1  dataset postProcessing over class : Model!Class {
2    properties [name]
3    @fillNulls none
4    column parentName { ... }
5    @normalize
6    column CK_NOC : class.children().size()
7  }
```

First, nulls are treated in column *parentName* (line 3). This column calculation, which appears in Listing 2, returns *null* if the class has no parent. The *fillNulls* annotation indicates that nulls will be filled by the declared value, *none* in this case. Other supported ways of filling nulls can be applied: using *mode* or *mean* as value would fill null cells with the mode or the mean of the column, respectively. Obviously, the *mean* option can only be used in those columns that are numerical.

Second, the *CK_NOC* column, which is the number of children of the class, is normalized (line 5). In this case, we use the *normalize* annotation, which by default divides column cells by the maximum value of the column, i.e., the largest children count. We can use other normalization value by indicating it as with the *fillNulls* annotation.

The next section gives implementation details about Pinset.

V. Implementation

Pinset has been made available as open-source software[3]. In addition, all dataset extraction examples shown in this paper can be found in a separate repository[4]. Right now, the implementation consists of two Eclipse plugins. The first one contains Pinset's parser and execution engine, while the second one offers an Eclipse editor with support for Pinset syntax and configurable execution wizards.

The following describes the internal components of Pinset, and the steps that take place in the execution of a Pinset file.

1) Epsilon Platform Usage: Epsilon [11] is a software suite composed of interoperable languages, each supporting a different model management task. These tasks include, among others, validation (EVL), comparison (ECL), model-to-model (ETL) or model-to-text (EGL) transformations. All

[3]https://github.com/alfonsodelavega/pinset

[4]https://github.com/alfonsodelavega/pinset-examples

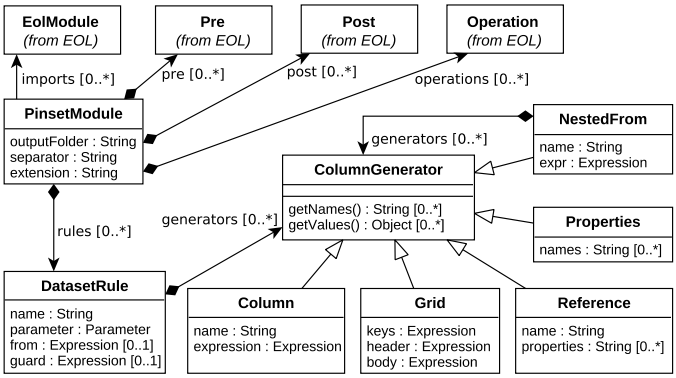

Fig. 3. Abstract syntax of Pinset.

these languages share a common *core*: the *Epsilon Object Language (EOL)* [20]. This language provides OCL-like expressions for model management, and supports imperative language structures such as conditional and loop statements, user-defined operations and import declarations. The other languages provided by the Epsilon suite are developed atop EOL's syntax and execution engine. Following the same approach, we implemented Pinset using EOL as base.

Other benefit provided by Epsilon is support for many model types. Epsilon provides a wider definition of what can be treated as a model with the *Epsilon Model Connectivity (EMC)* layer. This layer allows supporting new model types through the implementation of a driver. At the moment, languages from the Epsilon Platform (thus including Pinset) can operate with a broad number of information representations, including, among others, EMF and UML models, XML files, spreadsheets and relational databases.

2) Structure of Pinset: Figure 3 shows the abstract syntax of Pinset. As the language is defined over EOL, some elements are inherited, such as *Expressions* or *Operations*.

Pinset programs are organized in modules. A module (*PinsetModule*) can import external modules from the Epsilon platform, such as an EOL library file with operation definitions. Each module also contains optional *Pre* and *Post* statement blocks, which are executed before and after the datasets are generated, respectively. It is also possible to declare *Operations* for the encapsulation and reuse of common functionality during the dataset creation process.

Additionally, the module contains information about where and how to store the generated datasets. It requires an *outputFolder*, an *extension* for the dataset files, and the *separator* to be placed between the columns. By default, CSV files are generated, but these output settings are configurable.

The main component of a module are its *DatasetRule* definitions. These rules have a *name*, a *parameter* that stores the name and type of the transformed elements, and a set of *ColumnGenerators*, that provide the columns of the dataset.

The *ColumnGenerator* interface defines two methods: *getNames*, which returns the names of the columns it defines; and *getValues(Object)*, which calculates the column values for the object that is passed as parameter. Depending on the column generator, one or more columns will be generated. For instance, a *Column* construct always returns one column, while in the case of the other generators this number is variable.

3) Execution Process: First, *pre* blocks are executed, in the same order they were declared. Secondly, each defined *DatasetRule* is processed individually. The elements that will be iterated to generate rows are gathered (see section IV-C). Column names are obtained once from the *getNames* method of the generators. Then, the selected elements are processed one by one. Cell values of a row are calculated by feeding the *getValues* method of the declared generators with the respective element of that row. For those generators that employ expressions, the element is made accessible through the name of the rule's *parameter*. Post-processing operations are performed as the last calculation step. The obtained datasets are stored following the output details of the *PinsetModule*, regarding destination folder, column separator and file extension. Finally, *post* blocks are executed, in the order that they were declared.

TABLE III
SIZE IN LINES OF CODE (LOC) OF PINSET AND ETL SCRIPTS.

Extraction Script	ETL/LOC	Pinset/LOC	% Reduction
Listings 1-2 (Overview)	36	16	55,6%
Listing 3 (Accessors)	18	4	77,8%
Listing 3 (Extended)	28	4	85,7%
Listings 4-5 (Filtering)	13	4	69,2%
Listing 6 (Grid)	30	12	60,0%
Listing 7 (Typeless)	27	11	59,3%
All metrics	61	36	41,0%
Sum of all scripts	231	98	57,6%

VI. DISCUSSION

Here we analyse whether Pinset satisfies our initial goal. The goal consisted in the creation of a DSL for specifying data acquisition tasks from models. The use of this DSL should lead to more compact and less verbose specifications, which should be easier to understand and maintain.

To analyse compactness, we have compared the size of the data acquisition scripts shown throughout this paper with their corresponding ETL [14] counterparts. The selection of ETL for the comparison is irrelevant, as other M2M languages such as ATL would offer similar results. The ETL scripts can be found in an external file[5]. In addition to the scripts present in this paper, we added two other examples: an extended version of Listing 3, where more properties and references are extracted; and a script where all metrics from Table I are calculated. Table III summarises the results of this comparison.

To measure script size, we counted lines of code (LOC). For ETL scripts, some artefacts that are reused across scripts were not considered. Specifically, these artefacts are: (1) the helper functions for dataset management (Listing 1); (2) the dataset metamodel definition (Figure 2); and, (3) the model-to-text transformations that would generate the final CSV files.

[5]https://github.com/alfonsodelavega/pinset-examples/blob/master/es.unican.istr.pinset.examples.etlComparison/etl/01-examples.etl

Table III shows that Pinset is able to reduce scripts size by half on average ($\backsim$57%), when compared with ETL scripts. This reduction is due to the use of high-level column generators specifically designed for certain data acquisition tasks. These primitives avoid the need to explicitly manage column creation. Moreover, some columns generators, such as feature accessors (see section IV-B), greatly help reduce script size. These benefits are not present in general-purpose model transformation languages, like ETL, where including this domain-specific syntactic sugar would not make sense.

Moreover, Pinset provides some specific features that help simplify code, such as the management of null values. For some generators, when a reference is accessed, we do not need to check if this reference points to null. If it does, instead of raising an exception, Pinset provides an appropriate default behaviour, which most of the time prevents developers from having to take care of this issue.

The high-level syntax of Pinset also contributes to improve maintainability. Thanks to this syntax, we do not need to manage explicitly the dataset structure, which makes dataset definitions easier to maintain. As an example, if we wanted to include or remove a column from a dataset in Pinset, we would only need to update one section of the script, that is, the generator where the column is defined and calculated. In the ETL scripts, such as the one shown in Listing 1, there are three different places that would require modifications: (1) the section where columns are created (lines 5-11); (2) the statement where columns are assigned to the dataset (lines 12-16); and (3) the piece of code that calculates the values for that column (lines 23-38). This redundancy makes maintenance more complex, and it might lead to inconsistencies and errors.

These improvements in conciseness and maintainability seem to indicate that Pinset scripts might be easier to understand when compared to equivalent versions written in a generic model-transformation language. However, to be rigorous, this assessment needs to be confirmed with empirical research, where possible end-users, i.e., modeling experts, evaluate Pinset against the tools they employ daily. However, executing these empirical experiments gets outside of the scope of this paper, and will be part of our future work.

With respect to performance, in our experience, Pinset scripts take a similar execution time to those of ETL. As with conciseness and maintainability, we will carry out more detailed tests to achieve a rigorous performance comparison.

VII. Conclusions and Future Work

This work has presented Pinset, a language for the extraction of tabular-based datasets from models. This kind of extractions allow to enrich data mining processes by including models as new data sources, which enables the assessment of these models through advanced quality analysis techniques.

When compared with existing model management or model transformation tools, Pinset offers an equally powerful but more concise way of declaring datasets. This mainly happens because existing tools require preparing and managing the structure of the datasets explicitly while, in the case of Pinset, this structure is managed internally by the language. Therefore, it allows forgetting about boilerplate code and focusing on the features we wish to extract from the models. In addition, Pinset offers high-level constructs that facilitate the definition of dataset columns and the execution of typical dataset-related tasks that would need to be manually performed instead.

As future work, we will carry out detailed performance and end-user tests, to empirically assess the language and to see what kind of new features would be well-received.

Acknowledgment

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

References

[1] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[2] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, 2015.

[3] F. Palomba et al., "Investigating code smell co-occurrences using association rule learning: A replicated study," in *IEEE Workshop on ML Techniques for Soft. Quality Evaluation (MaLTeSQuE)*, 2017, pp. 8–13.

[4] M. Ochodek et al., "Using machine learning to design a flexible LOC counter," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Feb 2017, pp. 14–20.

[5] J. Zhang et al., "Predictive mutation testing," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2018.

[6] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010, pp. 31 – 41.

[7] Beller, Moritz et al., "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration," in *14th IEEE Working Conference on Mining Software Repositories*, 2017.

[8] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Mining Metrics for Understanding Metamodel Characteristics," in *International Workshop on Modeling in Software Engineering, MiSE 2014*, pp. 55–60.

[9] Object Management Group, "The Object Constraint Language Specification," https://www.omg.org/spec/OCL/.

[10] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A Model Transformation Tool," *Sci. Comput. Program.*, vol. 72, pp. 31–39, 2008.

[11] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. , "The design of a conceptual framework and technical infrastructure for model management language engineering," in *IEEE International Conference on Engineering of Complex Computer Systems*, 2009.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Soft. Eng.*, vol. 20, pp. 476–493, 1994.

[13] R. Hebig et al., "The Quest for Open Source Projects That Use UML: Mining GitHub," in *ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, 2016, pp. 173–183.

[14] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon Transformation Language," in *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, 2008, pp. 46–60.

[15] U. Fayyad et al., "The KDD Process for Extracting Useful Knowledge from Volumes of Data," *Commun. ACM*, vol. 39, pp. 27–34, Nov. 1996.

[16] H. Osman et al., "Automatic feature selection by regularization to improve bug prediction accuracy," in *IEEE Workshop on ML Techniques for Soft. Quality Evaluation (MaLTeSQuE)*, 2017, pp. 27–32.

[17] C. Couto et al., "Predicting software defects with causality tests," *Journal of Systems and Software*, vol. 93, pp. 24 – 41, 2014.

[18] Tantithamthavorn, Chakkrit et al., "Automated parameter optimization of classification techniques for defect prediction models," in *38th International Conference on Software Engineering (ICSE)*, 2016, pp. 321–332.

[19] J. J. López-Fernández, E. Guerra, and J. de Lara, "Combining unit and specification-based testing for meta-model validation and verification," *Information Systems*, vol. 62, pp. 104–135, 2016.

[20] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The Epsilon Object Language (EOL)," in *Model Driven Architecture – Foundations and Applications*, 2006, pp. 128–142.