Model-Driven Verification of Data Science Pipelines Execution

***Abstract*—The number of tools and approaches to define pipelines for data science projects has significantly increased in the last few years. These tools enable not only pipeline definition but also code generation for pipeline execution, making it simple for non-expert users to develop and deploy these projects. However, there are still some issues that these tools do not fully address, which may compromise data science projects results like verification of data consistency through data pipelines execution. Our approach proposes a model-driven framework for data science projects that defines pipelines at two different abstraction levels: conceptual (platform-agnostic) and operational (platform-specific). This paper addresses the conceptual level with: i) the proposal of a metamodel containing the core concepts needed for the pipelines specification; ii) the definition of a library of reusable data operations used in the pipelines specification; and iii) the use of reusable contracts for verification rules that provide a systematic way of checking data consistency using preconditions, postconditions, and invariants on the datasets. Using a real publicly available data science pipeline for validation purposes, the proposed approach was able to identify the contracts that were not fulfilled considering the specified verification rules.**

***Index Terms*—verification, data science pipeline, model-driven engineering, OCL, replicability**

I. Introduction

The availability of huge amounts of data generated in our daily lives challenges data science platforms that are now repeatedly used for extracting knowledge from that data. Although the considerable evolution of these tools, there are still challenges to be addressed by the research community. One of these challenges is associated with the difficulty in reproducing and replicating the experiments developed in data science projects [1]. Among the reasons for this difficulty are: i) the used infrastructures [2]; ii) the complexity of the data science pipelines [3]; and, iii) the lack of standards for defining these pipelines [4]. As consequence, data pipelines are manually defined on top of a plethora of tools and technologies. These pipelines are usually error-prone and difficult to trace [5]. Despite significant contributions from Predictive Model Markup Language (PMML[1]), a a standard to improve platform interoperability, and thus enhancing reproducibility and replicability (R&R), several limitations remain as it addresses the creation of the machine learning models and the representation of the input/output data of the model but does not allow the representation of a complete data science pipeline (such as the data cleaning and feature engineering processes). Also, this standard is not supported by all tools and does not provide a way to validate the consistency of the results obtained when running the same pipeline in multiple platforms.

In that context, the work in [6] highlights the challenge of verifying the consistency of the results obtained throughout experiments as a key action to ensure (R&R) of data science projects. Based on this verification, the particular operations where a failure occurs could be identified and, thus, the correction actions carried out. However, data consistency verification can easily become complex. Data pipelines usually define complicated sequences of data derivations on input datasets. Tens of data operations may transform input data to make it suitable for machine learning (ML) algorithms. Most data platforms provide the functionality to graphically design and deploy data pipelines, but verification and validation features are less common and usually incomplete. Moreover, data pipelines execution cannot be easily verified when designing and executing platforms are not the same. Different tools may provide slightly different implementations of data operations, even those data divergences could appear when using different versions of the same tool/platform or they could be manually introduced in a migration. Therefore, different executions of the same pipeline could imply divergent data derivations making data inconsistent among execution tools. As a result, the (R&R) may be compromised.

Based on this assumption, this paper presents a model-driven approach for the systematic verification of data consistency through data pipelines execution. This approach is defined upon a model-driven framework for the specification of platform-agnostic data science pipelines (conceptual level) and the specification of their deployment and execution (operational level). In this work, we focus on the conceptual level, where we defined a metamodel (that builds and extends PMML) which provides data scientists with a language for specifying platform-agnostic data science pipelines. Based on the conceptual model, we propose the definition of a library of common data operations to be reused in the definition of data science pipelines. For verification purposes, we define a reusable contract for every data operation by means of preconditions, postconditions and invariants. Therefore, given a particular instance of a data operation, we can check that: (1) the input dataset is complete and correct (satisfying preconditions); (2) the output dataset presents the expected structural and data derivations (satisfying postconditions); and (3) output data is properly derived from input data (invariants). As a result, we can systematically check data consistency in any point of a data science pipeline execution checking the contracts on the operations involved in that execution. Furthermore, our

[1]https://dmg.org/pmml/pmml-v4-4-1.html

approach also remains technology independent, so contract verification can be reused in different executions of the same pipeline, eventually improving R&R.

The rest of paper is organized as follows. Section 2 describes related works are presented. Section 3 introduces our conceptual model for the definition of platform-agnostic data science pipelines. In order to illustrate the approach, Section 4 presents an illustrative example whilst Section 5 presents the core of our model-driven verification approach based on our conceptual model and illustrated by using the running example. Section 6 presents an evaluation of the approach in a publicly available and well-known data science pipeline. Moreover, main limitations found are discussed. Finally, Sections 7 concludes the paper and presents future research lines.

II. Related Work

Several works that try to contribute to the (R&R) in the development of data science projects, such as [7]–[12]. As example of some of them, the work in [7] reviews eleven applications focused on assisting researchers in publishing reproducible research, including executable code and data. REANA platform [8] aims at facilitating reproducible science practices using a cluster composed of a set of micro-services that allows instantiating, launching and monitoring container-based computational workflows on cloud-based infrastructure. ReproZip [9] provides reproducibility and basic replicability by allowing the encapsulation and execution of data, code and computational environment on a Reprozip server without the need for tracking down or installing dependencies. [10] provides a Python API for predictive modeling or production rule analysis in education that uses Docker to enhance reproducibility. None of these works considers the separation into two abstraction levels, defining data science pipelines in a platform-agnostic level.

This separation is driven by the MLOps methodology [13], which attempts to separate the data scientist from the infrastructure used to deploy the developed pipelines. Although not explicitly focused on the development of reproducible and replicable pipelines, more and more papers try to propose frameworks to develop pipelines that are agnostic to the deployment platform [14] [15] [16].

Model Driven Engineering (MDE) has been almost neglected in the field of error detection and quality control in data science pipelines [17]. In [17], the Pitfalls Analyzer automates the detection of pitfalls in data science projects following MDE practices. [18] defines a model-driven approach for the validation of data science pipelines. The pipelines are represented as Directed Acyclic Graphs and go through them looking for the defined anti-patterns. Unlike these works, our approach does not only uses static analysis but also a dynamic-type for checking invariants.

For Model-Driven correctness verification, *correctness needs to be checked against a contract or specification* [19]. As we do in this work, OCL has been used as constraint specification language to test correctness in model transformations. In [20], the authors introduced an approach to apply correctness verification to ATL model transformations based on models that comply OCL-constrained metamodels. Like the work presented here, contract-based verification is used to check pre and post-conditions in order to ensure that the model transformation is valid. [19] also uses OCL contracts to test model transformations. In this case, input test models are generated and later automatically transformed into the output models whose properties are checked by using USE tool.

In our previous work[2], the (R&R) of data science projects were addressed considering the separation between the conceptual and the operational layer, but mainly presenting the contributions for the operational layer and the environment where the pipelines will be deployed. Next, the conceptual layer, contracts and library of reusable components are formalized.

III. Model-driven Data Science Pipelines

The model-driven verification approach presented herein is part of our research line on systematic (R&R) of data science projects (MD4DSPRR)[3]. Therefore this verification process is defined upon our model-driven approach organized into two fundamental layers: (1) the conceptual layer for data scientists to specify data pipelines (data and operations); (2) and an operational layer for data engineers to provide and configure the required technological infrastructures. Such separation allows the definition of platform-agnostic (conceptual) pipelines that can later be deployed in multiple execution environments. Moreover, given a pipeline specification and multiple execution environments for it, multiple properties could be verified by means of model-driven methods and tools.

In this section we briefly introduce the new evolution of our conceptual metamodel (MM) for the specification of data and data operation sequences. Then, our library of reusable data operations used in the pipelines specification is presented.

A. Conceptual Metamodel

For the definition of our conceptual MM we thoroughly studied the PMML standard and collected some extensions from KNIME[4], one of the most widely used data science tools in the literature [21], [22]. Concretely, we used the PMML terminology to define: i) the data entities processed by pipeline operations (i.e. data dictionary entities); and ii) the main concepts for the definition of data transformation.

Additionally, we consider two main extensions from KNIME: i) the inclusion of parameters in data transformations, an ii) the addition of concepts for data transformation sequencing and modularization (e.g. using jobs to integrate several operations or input and output ports).

In order to make the paper self-contained, next we just describe the main entities included in our MM (DataProcessingElement, DataField, and Transformation), shown in Fig. 1, as these provide the fundamental concepts for data and data operation definition and sequencing.

[2]removed reference for double-bind peer-review
[3]removed reference for double-bind peer-review
[4]https://www.knime.com/

A key component of our MM is the DataProcessingElement abstract class, which defines two different concrete subclasses: i) Transformation, which are atomic operations applied to data, and (ii) Job, which is a collection of related data transformations or nested jobs for the specification of projects of various sizes and granularities. Each Transformation or Job contains one or more ports (input or output) which are identified by their index attribute. Ports are mainly DataDictionary elements (i.e. datasets), which can be identified by their name and described in terms of their rows number and columns (DataField entity). In the future, other data formats, such as graph data, could also be ports.

DataField entities represent data fields from a data dictionary. They are identified by their name. Their dataType attribute specifies the type of the actual data (String, Time, Integer,...). Their id attribute indicates whether it belongs to the DataDictionary identifier (e.g. primary key in relational databases). Their target attribute is set to true when a specific DataField will be used as a target variable for a machine learning (ML) model.

DataField may also collect valid, invalid or missing (i.e. null) values by means of the named associations with ValueField entities, which specify a real data value and its appearance frequency in the dataset. In the case of Continuous DataFields, a set of intervals may also be specified. Note that some PMML constraints have been modeled as OCL constraints in our MM (not presented in this work). For example, Continuous instances must have its dataType attribute set as Integer, Double or Float.

A Transformation represents a data operation applied on a data field of the data dictionary. For each Transformation a Field entity indicates which DataField entity is transformed.

Usually a transformation involves the generation of a new data field in the dictionary, represented by the DerivedField class. Moreover, MapSpecialValues instances allow specifying how to treat missing values and values by default.

A Transformation may also include a set of parameters that define its behaviour. The Parameter class is identified by its name; the required attribute specifies whether it is mandatory or not. Parameters may be of different types: PrimitiveValue represents any basic configuration data that is needed for the transformation; FilterType and MatchingType, with the associated classes MissingValues, Range, FieldValue and RowNumber, allow the definition of filters to be applied to a DataDictionary; Map is used to define the configuration of mapping operations (e.g. Label Encoding); ImputeType, using DerivedValue and NumOperation, allows defining the different imputations of noisy values that may be applied to the DataDictionary; DiscretizeBin defines the different intervals to be used when continuous values are discretized (i.e. data binning in Data Science).

The abstract syntaxes of the conceptual and operational metamodels have been defined by means of the USE Modeling Framework[5]. These metamodels are available in a public repository[6] for further use and referencing.

[5]https://www.db.informatik.uni-bremen.de/projects/USE-2.3.1/

B. Library of Transformations

Conceptual models for data pipelines could be created from scratch, thus requiring a huge modeling effort. To alleviate such effort, this work proposes the definition of a library of transformations, which are specified as incomplete model snippets conforming to our conceptual metamodel. Those templates can be directly copied and fully instantiated in a model specifying a particular data pipeline.

As a first step, several known data mining tools [21], [22] (RapidMiner[7], KNIME[8], Orange Data Mining[9]) were thoroughly analyzed to collect their main data operations and map them. Additionally, their definition in PMML standard version 4.4.1 was also examined. A small excerpt of the results obtained is shown in table I. The interested reader may find the complete list of transformations and their mapping to several well-known data mining tools in our previous work[10].

TABLE I
DATA OPERATIONS MAPPING.

Transformation	KNIME	Python	RapidMiner	PMML
Impute	Missing Value	Pandas fillna	Impute Missing Values	Yes
RowFilter	Row Filter	Pandas filter	Filter Examples	Yes
RemoveDataField	Column Filter	Pandas drop (columns)	Remove Attribute Range	No
Mapping	RuleEngine	Pandas map	Map	Yes
ParseInt	String to Number	Pandas astype (int)	Parse Numbers	Yes
ReplaceOutliers	Numeric Outliers	Pandas apply with condition	Detect Outlier	Yes
Discretize	Numeric Binner	Pandas cut	Discritize by Binning	Yes

Following, we provide a brief description of those transformations. IMPUTE performs an imputation of missing or invalid values. ROWFILTER performs filters by values and by range of rows. REMOVEDATAFIELD removes a DataField from the DataDictionary. MAPPING changes the value defined in "inValue" to the value defined in "outValue". PARSEINT changes the dataType of a dataField to "Integer", which contains numeric data but its dataType is "String" for various reasons (invalid values, corrupted...). REPLACEOUTLIERS replaces outliers with through the indicated operation. DISCRETIZE divides a numeric DataField into intervals.

The library is available in our repository cited previously. However Table II shows a summary of the Library in tabular form, with the most important aspects of the different transformations.

First of all, note that every Transformation is an instance of Transformation, all of them have an input and output DataDictionary, and are applied on a DataField (this part is not included in the Table to simplify its content and because it is common to all Transformations). The column Configuration refers to the entities related to the Transformations beyond the parameters (MapSpecialValues, DerivedField and Field entities).

[6]removed for double-bind peer-review
[7]https://rapidminer.com/
[8]https://www.knime.com/
[9]https://orangedatamining.com/
[10]removed reference for double-bind peer-review

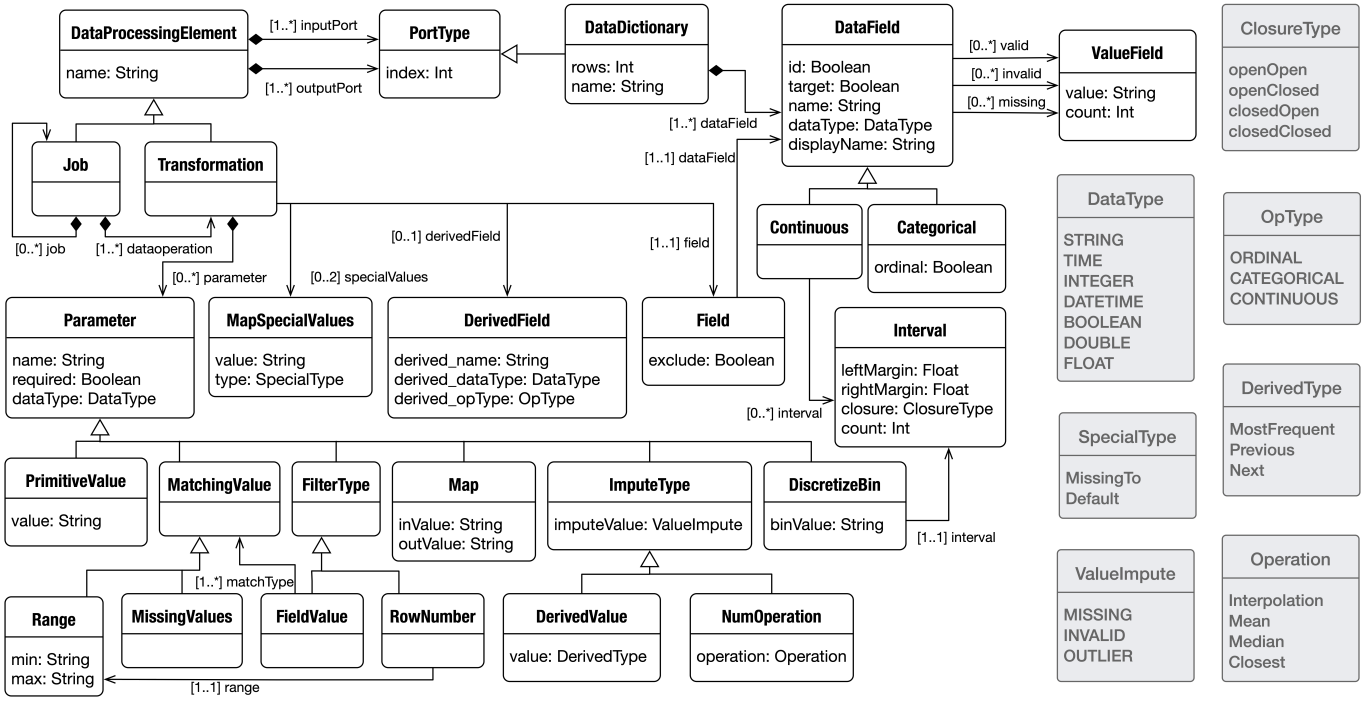

Fig. 1. Conceptual Metamodel

TABLE II
DATA OPERATIONS DEFINITIONS.

Transformation	Configuration	Parameters
Impute	DerivedField(0..1)	impute:ImputeType(1..1)
RowFilter	DerivedField(0..1); MapSpecialValues(0..1)	filterType:FilterType(1..1); value:MatchingValue(1..1)
RemoveDataField	–	–
Mapping	DerivedField(0..1); MapSpecialValues(0..1)	map:Map(1..*)
ParseInt	DerivedField(0..1); MapSpecialValues(0..1)	–
ReplaceOutliers	DerivedField(0..1)	operation:ImputType(1..1); q1:PrimitiveValue(1..1); q3:PrimitiveValue(1..1); multIqr:PrimitiveValue(1..1)
Discretize	DerivedField(0..1); MapSpecialValues(0..2)	bin: DiscretizeBin (1..*)

Each transformation has its configuration and parameters defined according to its definition. For example, all transformations can have a DerivedField that stores the result of the transformation, except REMOVEDATAFIELD, since, according to its definition, it removes the DataField from the data dictionary, so it does not make sense to store the result in a resulting DataField (DerivedField). On the other hand, those operations for which it makes sense to specify what to do in case of a null value, have as configuration MapSpecialValues(0..1). As an example, in IMPUTE transformation, the configuration for defining what to do in case the operation hits a null value does not make sense, since it is implicit in the operation action (observe that this operation is applied to replace null values). In the case of the DISCRETIZE transformation, it has as configuration MapSpecialValues is (0..2) because besides specifying what to do in case of a null value, we need to be able to establish the action to be performed for those values that are out of the defined intervals, i.e. what value it returns by default.

For the sake of illustration, we next describe the DISCRETIZE operation and model it according to our MM (Fig. 2). This operation contains a complete configuration using most of the MM classes required to define an operation. DISCRETIZE, as previously mentioned, divides a continuous variable (DataField) into different range intervals. In order to create these intervals, DiscretizeBin parameters are used. Continuous variable is specified using Field, which references the corresponding Continuous Datafield of the input Datadictionary. Each parameter contains a binValue attribute, used to name that interval, and the corresponding interval by means of the Interval class. In Fig. 2, we show only one DiscretizeBin parameter, but a model could contain as many as needed. The input DataField could contain values not included in any of the defined intervals, in that case, MapSpecialValues may be used to define a default value for them (type attribute set as "Default"). Moreover, the transformation may generate a derived DataField (DerivedField) of Categorical type (derived_opType) with String data type (derived_dataType). The result of this Transformation will be reflected in new DataField included in output DataDictionary. Notice that Fig. 2 contains some attributes with "Undefined" value because we are just defining the generic template for the operation and not showing how to apply it on any specific real DataField.

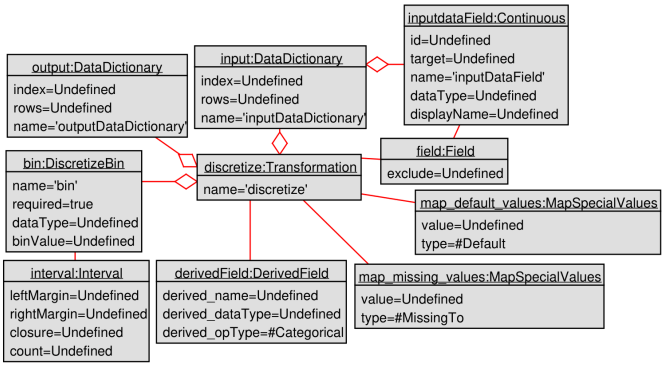

Fig. 2. USE Modeling Framework view of Discretize Transformation model

IV. Illustrative Example

Next, an illustrative example is presented to better convey how the previously defined conceptual model and operations library can be used to specify data science pipelines. This example was excerpted from a previous project on academic dropout prediction developed by our team[11]. The primary objective of this project was to predict whether an engineering student is going to drop out of his/her degree at the end of each semester, so correction actions may be triggered to reduce the existing high-rate dropout in technical careers. For this purpose, a ML model was trained from academic and personal data. Concretely, an ensemble model was applied integrating the results of three different ML algorithms: Gradient Boosting, Random Forest and Support Vector Machines.

For the sake of clarity, here we present the pipeline fragment for the creation of the Gradient Boosting learning model as an illustrative example, because of its reduced size and representative operations flow. This flow contains (i) operations involved in the learning model generation (training and testing), (ii) common data processing transformations, and (iii) feature engineering transformations. More information and materials are available at the project repository.

The input dataset structure is presented in Table III. Each row shows the data needed to define each column of the dataset with our conceptual DSML. Two additional columns present a brief description and some example values for each column (feature) respectively.

TABLE III
Dataset Definition

Columns	Data Types	ID	Target	Description	Examples
degree_name	categorical	0	0	Name of degree	Computer Science
call_year	categorical	0	0	Year of the call for Access	2007-08, 2008-09
call	categorical	0	0	Call for Access	June, September
access_description	categorical	0	0	Access Type Description	Exam, Transferred
gender	categorical	0	0	Gender of student	M, F
interval_year_birth	categorical	0	0	Year of Birth divided into 5-year intervals	(1990-1995], (2000-2005]
scholarship	categorical	0	0	Indicate if the student has a scholarship	N, Y
cum_pass_ratio	float	0	0	Ratio of subjects passed per taken	0,5; 0,7
marks	float	0	0	Mark obtained in University Entrance Exam (0 - 14)	10, 11, 9
cum_absent_ratio	float	0	0	Ratio of subjects not presented to the exam per taken	0,8; 0,9
record_id	integer	1	0	Student ID by degree	1, 5, 105
degree_id	categorical	1	0	Unique Degree Identifier	233, 1623, 1627
dropout	categorical	0	1	Indicates if the student has dropped out	0, 1

[11]removed reference for double-bind peer-review

Having explained the dataset we will use in the illustrative example, we proceed to define how to describe our pipeline, presented in Fig. 3, in our DSML (conceptual model).

Note that an ad-hoc concrete syntax (described in the legend) is used to better illustrate this example. Here we just explain the actual parameters of the most relevant operations previously described in section III-B.

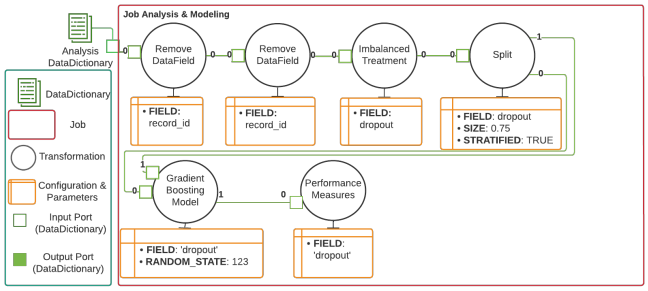

Fig. 3. Conceptual Model of the Illustrative Example

In this work, we are only interested in the transformations defining the pre-processing stage of the data pipeline so that next we will explain Remove DataField, Imbalanced Treatment, and Split transformations. The rest of the pipeline are, thus, not detailed since they are part of the Machine Learning (ML) training and testing stages. Note that the numbers next to input and output ports of each operation allow distinguishing between datasets when they are more than one and refers to attribute index of PortType entity (according to our metamodel).

First, a Remove DataField transformation is instantiated to remove the DataFields record_id and degree_id for dataDictionary anonymization. Then, an Imbalanced Treatment transformation is applied to correct the data imbalance present in the dropout column (i.e. true cases of dropout were clearly greater than false cases). Finally, once a series of transformations have been applied to our dataset to clean it prior to the creation of the ML model (Gradient Boosting), we proceed to split dataset in Test and Train sets, and for this we apply the Split operation. In this case, we want to allocate 75% of the data for training the ML model (output port 0) and the rest (25%) for testing it (output port 1). In addition, we want that the target dataField (dropout) has the same distribution (50%/50%) for its categories (True and False), for this reason, transformation reference to dropout dataField.

V. Model-driven Verification of Data Pipelines

As claimed in [6], the validity of R&R in data science projects requires the verification of obtaining consistent results for each pipeline operation (this is what we called data consistency).

To this purpose, and based on our conceptual model and library of operations, we define a reusable and systematic method for the verification of data consistency through data

pipelines execution by means of model-driven techniques. This verification approach is presented in subsequent subsections.

A. Contract Definition

One of the main contributions of our conceptual model is the possibility of verifying pipeline quality properties in models based on it. In that sense, data pipelines typically define complex data derivation sequences from input datasets and it verification may be complex. We propose explicitly defining reusable technology-independent contracts for data operations so that data consistency can be verified at any point in the execution of data pipelines.

Our contracts for data operations follow the structure proposed by the software correctness methodology Design by Contract in [23], so three different types of conditions can be specified: **preconditions**, **postconditions** and **invariants**.

Preconditions. Define the conditions that must be met by the DataField to which the transformation will be applied, as well as the parameters for the transformation. As an example, in a Mapping transformation, the input DataField must at least contains a value that is equal to the value specified in the inValue attribute of the Map parameter.

Postconditions. Specify the conditions that the output DataDictionary and the target DataField must fulfill. Following the aforementioned Mapping example, the output DataField may not contain any ValueField with a value equal to the specified in the inValue attribute of the Map parameter.

Invariants. Specify relationships between input and output DataDictionary that must be held by the execution of the transformation. Using the same example, the DataField values of the input DataDictionary equal to the inValue attribute must be converted to the value of the outValue attribute of the Map parameter in the output DataField.

Those conditions can be defined in terms of a DataDictionary, particular DataField elements, or Parameters. Table IV presents the types of conditions possible per element in a contract. In preconditions, the conditions may evaluate Parameter values, DataField properties (id, target and dataType attributes), and their potential values (ValueField) or ranges (Interval). In postconditions, the output DataDictionary is evaluated in terms of its derived DataFields, the resulting rows number, and the properties and values or their ranges. Finally, in invariants, the output DataField values are evaluated according to the input ones.

TABLE IV
CONTRACT ELEMENTS

Element	Preconditions	Postconditions	Invariant
DataDictionary	-	DataField existence Value count	-
DataField	Properties ValueField Interval	Properties ValueField	Input and Output values
Parameter	Value	-	-

B. Tooling

In order to systematically verify data consistency in a data pipeline execution, it is necessary (1) to represent data pipelines and datasets (dataDictionaries) by means of models, (2) to specify the contracts in a model-driven language able to query and evaluate conditions on models, and (3) a toolkit to transform datasets into models and to evaluate the contracts on them. An USE-based metamodel has been already presented in section III-A to represent data pipelines and datasets. All the transformation contracts have been specified in OCL in the metamodel available in our public repository[12], which has been previously applied for contract definition and evaluation in other works [19], [20], [24]. As OCL evaluation tool, the UML Based Specification Environment (USE) has been used. Finally, a simple T2M transformation from datasets (CSV) to models (USE soil files) has been defined by means of a Python script. Regarding the data of the data fields, the script generates only the data of the data field on which the Transformation is applied, to reduce the load of data that is not useful to the process of verification.

All the material listed above is available for use in our repository.

C. Library Verification

A contract for data consistency assessment was defined for every transformation included in the library, as shown in Table V, so they can be easily reused in the execution of data pipelines to improve its replicability.

In Table V, we show the contracts for each transformation in natural language, however, they have been also described by using our metamodel through OCL conditions.

Continuing with the example of the Discretize transformation of section III-B, the definition of Postcondition **bins** of Discretize transformation in OCL is:

```
context Transformation::validateDiscretize(bins : Set(
    DiscretizeBin))

post bins: bins ->forAll( dB:DiscretizeBin | (self.
    getOutputDataField().validValues->select( v:ValueField
    | (v.value = dB.binValue) )->size() = 1) )
```

As Table V indicates, this condition establishes that: for each interval (DiscretizeBin) created by this transformation, there must be in the output dataField (DerivedField) at least one value (ValueField), equal to the one defined for that interval (binValue).

Note that our conditions are based on the definition of the transformations and the input and output data of the transformations through their execution, allowing them to be used independently of the selected development or execution tool.

D. Verification Process

In order to verify the contracts defined for the transformations appearing on a data pipeline, previously we need to derive a definition of such pipeline using our metamodel (at

[12]removed for double-bind peer-review

TABLE V
TRANSFORMATIONS CONTRACTS

Transf.	Preconditions	Postconditions	Invariants
Impute	**missing**: If valuesToImpute is Missing, there must be missing values in the input DataField. **invalid**: If valuesToImpute is Invalid, there must be invalid values in the input DataField.	**noMissing**: If valuesToImpute is Missing, there cannot be missing values in the derived DataField. **noInvalid**: If valuesToImpute is Invalid, there cannot be invalid values in the derived DataField.	**imputedMissing**: If valuesToImpute is Missing, missing data must be mapped to a specific non-null value. **imputedInvalid**: If valuesToImpute is Invalid, invalid data must be mapped to a specific non-null value.
RowFilter	**values**: If FilterType is FieldValue, the values to filter are defined as ValueField; **rows**: If FilterType is RowNumber, the range to filter must be valid for the number of rows of the DataDictionary.	**noValues**: If FilterType is FieldValue, the values must not be present in the derived DataField; **noRows**:If FilterType is RowNumber, the final number of rows of the DataDictionary must be decreased by the rows in the range.	–
Remove DataField	–	**removed**: The removed DataField must not be part of the DataDictionary.	–
Mapping	**inValue**: inValue must belong to the input DataField as ValueField.	**noInValue**: inValue must not belong to the derived DataField as ValueField.	**mapValue**: inValue values of the input DataField must be mapped to the outValue value in the derived DataField.
ParseInt	–	–	**parsed**: each value in the input DataField must have the same value in the derived DataField
Replace Outliers	**outliers**: outliers in input DataField	**noOutliers**: no outliers in derived DataField	**replacement**: input outliers mapped to the result of the selected NumOperation::operation in the output DataField.
Discretize	**intervals**: The Intervals associated with the DiscretizeBin attr. must be equal to the Intervals associated with the DataField.	**bins**: the ValueFields of the derived DataField must be those specified by the binValue attribute of the DiscretizeBin param. **size**: the count attr. of each ValueField of the derived DataField must correspond to the count attr. of the corresponding interval of the input DataField.	**values**: each value of the input DataField must have in the derived DataField the value defined in the binValue attribute associated to its corresponding range.

the conceptual level). Once the conceptual model is defined, concrete platform instances may be derived by means of the operational level, which allows specifying different versions of the pipeline into alternative execution environments. For example, it may be deployed by using two different platforms: KNIME and Python[13] technologies. Eventually, our approach will provide import/export operations to/from the conceptual model (i.e. T2M and M2T transformations) for the most common data science tools.

In that sense, the data operation library presented in section III-B is of upmost importance for: (i) the deployment of the pipeline in different execution environments; and (ii) the systematic verification of data consistency in each data operation. In both aforementioned deployment platforms we can automatically and systematically verify the data consistency throughout the execution of the pipeline by means of contract evaluation. Basically, we instrumentalize the execution process to save a copy of the temporal datasets generated by every

[13]https://www.python.org/

transformation executed in the pipeline. Therefore, we can later evaluate all the conditions of the contracts corresponding to each transformation. As final product, we can report on which conditions of a transformation contract (i.e. by their names) have not been satisfied for a specific instance of such transformation in the pipeline. Finally, the data scientist may get a detailed view of data consistency issues found in the pipeline execution.

VI. Validation

As a validation scenario, we have selected to apply our approach in the verification of data consistency when a pipeline is migrated from its design platform, where the pipeline is defined and tested with a small data sample, to an specific deployment platform, where the pipeline is actually executed to properly train and test the model with the full dataset. In this work, KNIME was selected as design platform, meanwhile the pipeline was migrated (deployed) to a Python environment with the utilization of Pandas Library (Python) among others. These platforms have been selected as they are among the most widely used in the Data Science literature [22]

To begin with, a publicly available pipeline has been obtained from the KNIME Community Hub, which contains more than 12K pipelines developed by users from different backgrounds, i.e. industry and academia. The criteria followed to select a significant pipeline is two-fold: (1) it must contain the most common data pre-processing transformations according to [25]; and (2) the application of those transformations has to be relevant enough to become a likely source of data divergence in different platforms. As main examples of those operations we can find those dealing with imbalanced datasets , or dealing with null or invalid values imputation, or dealing with data discretization (i.e. data binding); or encoding techniques (such as label encoding, among others).

The KNIME Hub pipeline selected is *Model data set*[14]. Its last update was in October 2021 and it has been downloaded or used as a reference by more than 250 users in the community. This pipeline contains 137 data operations (nodes in KNIME terminology): 58 of these operations related to pre-processing tasks, 10 to training and evaluating three different machine learning models and the rest of them used for data visualization. The main goal of this pipeline is to predict student enrollment into an educational program. This prediction is based on a set of 26 distinct features (sex, ethnicity, average degree in previous programs, etc.) and an additional *enroll* feature as the target one.

Since the main purpose of our approach is to improve the replicability of Data Science projects, in this case by verifying data consistency across different executions of the pre-processing data operations, we replicated the pipeline in Python, our deployment platform. This second version of the pipeline has been developed by an external member of our team specialised in the development of Data Science projects in such a platform. The final implementation is available in our public repository previously referenced.

[14]https://kni.me/w/SFKjghagXCJNpEN_

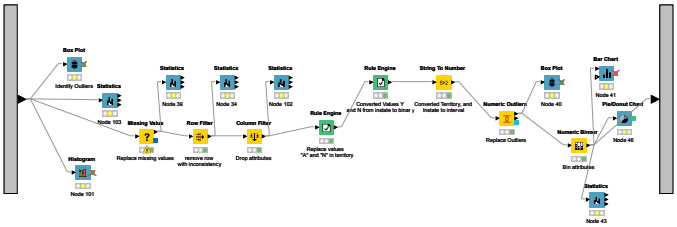

Fig. 4. Data Understanding and Data Preprocessing node

For the sake of brevity, we just present the *Data Understanding and Data Preprocessing* node, which contains the pre-processing operations included in the first part of the pipeline. This node gathers most of the data pre-processing operations of the project because it is executed three times along the pipeline. The data pre-processing operations included in this node are represented in Fig. 4. Since in this work we focus on data pre-processing operations, we will skip any other operation, i.e. plotting operations, shown in blue in Fig. 4.

The results obtained for the performance metrics by the execution in KNIME of the two machine learning models included in the first part of the project are summarized in Table VI. These results may be compared to the results obtained after replicating the project in the selected deployment platform (Python environment), shown in the same table. As presented, there are significant differences in the results obtained by both platforms. The most remarkable, due to the importance of the metric, is the variation of the accuracy obtained by the different models, which is almost 5% in the Random Forest algorithm. However, we may also mention the variation observed in other metrics, such as Kappa, whose variation is almost 10% for Random Forest.

TABLE VI
Performance metrics of ML models in KNIME and Python

Tool	ML Algorithm	Accuracy	Kappa	Error	ROC Score
KNIME	Random Forest	93.928%	87.9%	6.072%	0.975
KNIME	Decision Tree	91.860%	83.7%	8.14%	0.961
Python	Random Forest	89.860%	78%	10.13%	0.890
Python	Decision Tree	89.470%	77.2%	10.520%	0.889

The identification of the errors or inconsistencies in the replica would entail reviewing the results obtained by the different data operations of the pipeline one by one, which would easily become a cumbersome and error-prone task when the number of operations to review is high. This is where our approach comes to the scene, systematically validating the operations by means of the defined contracts and allowing us to verify that they have been applied as expected, according to their definition. After applying our verification process for the replica (Python environment), we get a report with the results of the verification of each condition of the contract associated to each operation of the pipeline. Table VII illustrates such

report. In this case, we verify the proper behavior of 49 operations in the pipeline. In the selected node, 6 of those operations are verified and their contract results are detailed by condition. As seen, most of the operations behave properly with the exception of Impute and Discretize ("Missing Values" and "Numeric Binner" in KNIME).

TABLE VII
VALIDATION PIPELINE DATA OPERATIONS.

KNIME	Frequency	Transf.	Precond. Sat.	Postcond. Sat.	Inv. Sat.
Missing Value	6	Impute	2/2	**1/2** (noMissing)	**1/2** (imputedMissing)
Row Filter	6	RowFilter	2/2	2/2	-
Column Filter	7	Remove DataField	-	1/1	-
Rule Engine	12	Mapping	1/1	1/1	1/1
String To Number (Integer)	6	ParseInt	-	-	1/1
Numeric Outliers	6	Replace Outliers	1/1	1/1	1/1
Numeric Binner	6	Discretize	1/1	**1/2** (size)	**0/1** (values)
Total	**49**				

Regarding Impute operation, the following conditions are not satisfied: (1) **noMissing** post-condition, which states that there cannot be null values in the derived DataField; (2) **imputedMissing** invariant, which states that null values must be mapped to the non-null value in the derived DataField. After a careful code review, including a thorough examination of Pandas documentation, we concluded that the Pandas operation selected, interpolate[15], was causing an error because it performs the linear interpolation considering just the values preceding the null values. Therefore, if null values appear at the top of a data field, this default interpolation cannot be performed but it need to be explicitly configured to use preceding and subsequent values. Note that KNIME missing value operation considers by default preceding and subsequent values.

Likewise, the Discretize operation contains the following conditions not satisfied: (1) **size** post-condition, which states that the number of values included in each interval in the input dataset must be equal to the number of values with a reference to that interval in the output dataset (e.g. if there are 10 observations in the input data with values between 0 and 5, there must be 10 observations in the output data with the value associated to that interval); (2) **values** invariant, which establishes that each input data must be represented in the output dataset by the value assigned to the interval that it corresponds to. Again, after a careful code and documentation review, we concluded that the Pandas operation selected, cut operation[16], was causing an error because, by default, its first interval is considered open-closed, so the leftmost value is left out of the interval. As a consequence, all the input values equal to the leftmost value of the first interval are directly mapped to null in the output dataset, since they do not belong to any interval specified. Note that KNIME numeric binner operation considers by default closed-closed intervals.

After solving the identified errors, the results obtained in the replica are shown in Table VIII. Note that result divergence between platforms has been clearly reduced. They are now lower than the 2% for all the metrics.

TABLE VIII
PERFORMANCE METRICS OF ML MODELS IN PYTHON WITHOUT BUGS

ML Algorithm	Accuracy	Kappa	Error	ROC Score	Δ Accuracy
Random Forest	91.948%	83.8%	8.051%	0.918	-1.980%
Decission Tree	91.610%	83.1%	8.389%	0.915	-0.25%

Finally, to estimate the impact of this kind of errors, we have performed a search in the KNIME Hub to get how many pipelines are using the problematic transformations: Impute (Missing values) and Discretize (Numeric binner). We found there are 165 pipelines using Discretize and more than 1500 using Impute. Therefore, a considerable amount of pipelines can generate data consistency errors when migrated to a Python platform. As a result, it seems automatic and systematic data consistency verification approaches could improve the replicability of a relevant number of data science projects.

In conclusion, small differences in the default behavior of similar operations in different platforms may easily become a source of data consistency errors, which are really complex to locate, track and debug.

A. Discussion

This section provides a discussion about different limitations that could jeopardize the approach application.

Scalabilty. We are aware that the modelling of the column sdata on which the transformation is applied, could jeopardise the results of the approach in terms of scalability (at loading and validating the models). Therefore, as mentioned in section VII, in the future the verification will be carried out externally to the defined model. Also, the evaluation of some conditions defined as Invariants require re-executing some transformations. We also plan to use probabilistic methods that allow us to perform these verifications on a sufficiently significant sample of the data, and not on all of them.

Contracts specification language. We decided to use OCL as contract specification language because it is widely use for restriction specifications in USE metamodels. However, we have observed a couple of limitations of the language to efficiently define contracts for data operations. First, the definition of common dataset operations (metadata or data queries) generates large OCL expressions, hindering its understanding and maintenance, specially when several conditions need to be specified in the same expression. In order to alleviate that issue, we have defined specific OCL query operations for common dataset operations already appearing in our contracts, e.g. getting the values of a particular column by its name, fetching the column of a dataset from a column parameter of a data operation, etc. Second, the mathematical function library is not complete enough to provide the required functionality for contract definition. Although some extensions have been defined in the past [26], they seem insufficient for dataset handling.

[15]https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html

[16]https://pandas.pydata.org/docs/reference/api/pandas.cut.html

VII. Conclusions and Future Work

In this work, we presented an approach to systematize the verification of data consistency through data pipelines execution by means of model-driven techniques. Our approach is defined upon a model-driven framework for the specification of platform-agnostic data science pipelines (conceptual level) and the specification of their deployment and execution (operational level). We focused here on the conceptual level and proposed the definition of a library of common data operations to be reused in the definition of data science pipelines. For verification purposes, we defined a reusable contract for every data operation by means of preconditions, postconditions and invariants. Therefore, given a particular instance of a data operation in an execution point of the pipeline, we can check that: (1) the input dataset is complete and correct (satisfying preconditions); (2) the output dataset presents the expected structural and data derivations (satisfying postconditions); and (3) output data is properly derived from input data (invariants). Finally, we validated our approach by applying it in a real publicly available data science pipeline.

As future work, we plan: (1) to use probabilistic methods to avoid verifying all the data but only a significant sample of them for the invariant conditions, which may require re-executing some transformations, (2) to be able to define additional contracts for each of the transformations through our metamodel, which allows contracts to be tailored to the characteristics of each project, and (3) to be able to generate these conditions in the desired language depending on the development language used (Java, Python, R ...) by the development tool, in order to incorporate them as part of the pipeline, just as tests are incorporated into a software development code, and overcome the above limitations of using OCL as a contract definition language.

References

[1] M. Hutson, "Artificial intelligence faces reproducibility crisis," *Science*, vol. 359, no. 6377, pp. 725–726, 2018.

[2] C. Willis and V. Stodden, "Trust but verify: How to leverage policies, workflows, and infrastructure to ensure computational reproducibility in publication," *Harvard Data Science Review*, vol. 2, no. 4, 12 2020.

[3] L. Rupprecht, J. C. Davis, C. Arnold, Y. Gur, and D. Bhagwat, "Improving reproducibility of data science pipelines through transparent provenance capture," *Proceedings of the VLDB Endowment*, vol. 13, pp. 3354–3368, 8 2020.

[4] C. Byrne, *Development Workflows for Data Scientists*. O'Reilly Media, Inc., 2017.

[5] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: Pitfalls and challenges." IEEE Computer Society, 5 2018, pp. 286–295. [Online]. Available: https://doi.org/10.1145/3183519.3183547

[6] X.-L. Meng, "Reproducibility, replicability, and reliability," *Harvard Data Science Review*, vol. 2, no. 4, 2020.

[7] M. Konkol, D. Nüst, and L. Goulier, "Publishing computational research - a review of infrastructures for reproducible and transparent scholarly communication," *Research Integrity and Peer Review*, vol. 5, pp. 1–8, 12 2020.

[8] T. Šimko, L. Heinrich, H. Hirvonsalo, D. Kousidis, and D. Rodríguez, "Reana: A system for reusable research data analyses," *EPJ Web of Conferences*, vol. 214, p. 06034, 9 2019.

[9] V. Steeves, R. Rampin, and F. Chirigati, "Using reprozip for reproducibility and library services," *IASSIST Quarterly*, vol. 42, pp. 14–14, 12 2018.

[10] J. Gardner, C. Brooks, J. M. Andres, and R. S. Baker, "Morf: A framework for predictive modeling and replication at scale with privacy-restricted mooc data," *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*, pp. 3235–3244, 1 2019.

[11] L. White, R. Togneri, W. Liu, and M. Bennamoun, "DataDeps.jl: Repeatable Data Setup for Reproducible Data Science," *Journal of Open Research Software*, vol. 7, no. 1, p. 33, oct 2019.

[12] A. M. Domenech and A. Guillén, "ml-experiment: A Python framework for reproducible data science," *Journal of Physics: Conference Series*, vol. 1603, no. 1, p. 012025, sep 2020.

[13] M. Treveil, N. Omont, C. Stenac, K. Lefevre, D. Phan, J. Zentici, A. Lavoillotte, M. Miyazaki, and L. Heidmann, *Introducing MLOps*. O'Reilly Media, 2020.

[14] S. Mäkinen, "Designing an open-source cloud-native MLOps pipeline," p. 66, 2021. [Online]. Available: https://helda.helsinki.fi/handle/10138/328526

[15] I. Pölöskei, "MLOps approach in the cloud-native data pipeline design," *Acta Technica Jaurinensis*, vol. 15, no. 1, pp. 1–6, apr 2021. [Online]. Available: https://acta.sze.hu/index.php/acta/article/view/581

[16] R. Miñón, J. Diaz-de Arcaya, A. I. Torre-Bastida, and P. Hartlieb, "Pangea: An MLOps Tool for Automatically Generating Infrastructure and Deploying Analytic Pipelines in Edge, Fog and Cloud Layers," *Sensors*, vol. 22, no. 12, 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/12/4425

[17] G. K. Rajbahadur, G. A. Oliva, A. E. Hassan, and J. Dingel, "Pitfalls analyzer: Quality control for model-driven data science pipelines," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2019, pp. 12–22.

[18] ——, "Pitfalls Analyzer: Quality Control for Model-Driven Data Science Pipelines," in *Proceedings - 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems, MODELS 2019*. IEEE, sep 2019, pp. 12–22. [Online]. Available: https://ieeexplore.ieee.org/document/8906968/

[19] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal Specification and Testing of Model Transformations," in *Formal Methods for Model-Driven Engineering*, ser. Lecture Notes in Computer Science, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer Berlin Heidelberg, 2012, no. 7320, pp. 399–437.

[20] F. Büttner, M. Egea, and J. Cabot, "On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds. Springer Berlin Heidelberg, 2012, no. 7590, pp. 432–448.

[21] S. Dwivedi, P. Kasliwal, and S. Soni, "Comprehensive study of data analytics tools (rapidminer, weka, r tool, knime)," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. IEEE, 2016, pp. 1–8.

[22] S. Bonthu and K. H. Bindu, "Review of leading data analytics tools," *International Journal of Engineering & Technology*, vol. 7, no. 3.31, pp. 10–15, 2017.

[23] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[24] J.-M. Mottu, B. Baudry, and Y. Le Traon, "Model transformation testing: oracle issue," in *IEEE International Conference on Software Testing Verification and Validation Workshop, 2008. ICSTW '08*, Apr. 2008, pp. 105–112.

[25] A. Chapman, P. Missier, G. Simonelli, and R. Torlone, "Capturing and querying fine-grained provenance of preprocessing pipelines in data science," *Proceedings of the VLDB Endowment*, vol. 14, pp. 507–520, 12 2020. [Online]. Available: https://dl.acm.org/doi/10.14778/3436905.3436911

[26] J. Cabot, J. N. Mazón, J. Pardillo, and J. Trujillo, "Specifying aggregation functions in multidimensional models with OCL," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6412 LNCS, 2010, pp. 419–432.