PromniCAT - Collecting and Analyzing Heterogenic Business Process Models

Cindy Fähnrich[1], Tobias Hoppe[1], Andrina Mascher[1]

Hasso Plattner Institute, University of Potsdam
[1] firstname.lastname@student.hpi.uni-potsdam.de

Abstract. Analyzing business process models is of high value for organizations, since it enhances performance improvements to their internal processes, bringing competitive advantages to organiziations. Thus, they create large collections of process models, which often are of heterogeneous nature in format and notation. A beneficial analysis of these large collections involves dealing with both an efficient processing of the data and its heterogeneity.
To address these needs, we introduce PromniCAT. Our framework collects process models from different notation and origin, harmonizing the heterogenic data to match a uniform format. Additionally, PromniCAT provides functionality to conduct performant analyses on collections of arbitrary process models.
In this work, we present the details of the overall framework in short, whilst concentrating on three main aspects: The implementation of common process metrics, the clustering of process models by both numeric and nominal criteria, and the index creation for efficiently storing and retrieving arbitrary analysis results.

1 Analyzing Process Model Collections

Business process modeling represents an important part for modern organizations. The resulting business process models (hereafter referenced as *process models*) depict the organizations' internal processes and reveal potentials for performance optimization, which can bring enormous competitive advantages for organizations. According to that, they build up large collections of process models, which are often created with proprietary modeling tools in arbitrary notations (BPMN [21], EPC [12], etc.) and formats (XML, JSON, SVG, etc.). Nevertheless, all of these heterogenic process models need to be analyzed, searched, and compared by certain criteria in an efficient manner to detect performance issues.

In this paper, we present *PromniCAT*, a framework for the collection and analysis of heterogenic process models that addresses the introduced needs. By importing process models from different origins such as the BPM Academic Initiative (BPMAI[1]) [13] or National Process Library (NPB[2]) [5], PromniCAT

[1] http://bpt.hpi.uni-potsdam.de/BPMAcademicInitiative
[2] http://www.prozessbibliothek.de

converts them into a uniform format and offers different possibilities for analysis. Our framework provides the usage of various process metrics relating to size and complexity of process models as well as a hierarchical clustering of process models by both numeric and nominal criteria. Additionally, our framework enables an efficient analysis by storing and retrieving arbitrary analysis results for later reuse.

The remainder of this paper is structured as follows: Section 2 gives a short introduction to the PromniCAT framework and its architectural details. Section 3 concentrates on the realization of calculating various process metrics. Section 4 focuses on creating indices for an efficient search and storage of analysis results. Section 5 elucidates the details of clustering and labeling process models by both numeric and nominal criteria. The results of our work are summarized in Section 6.

2 PromniCAT

PromniCAT aims at importing and analyzing a variety of process models provided in different notations into one generic database. This enables research on the entire collection across the boundaries of business process modeling notations. In this Section, we highlight the main architectural components of PromniCAT, elucidate the underlying database schema, and reveal the details of our generic process model.

2.1 Architecture

The framework consists of different layers presented in Fig. 1 and is implemented in Java. In the following, we describe the components of our architecture.

The *importer* component is used to fetch process models from a given process model collection. For every origin, a specific importer converts the collection into the PromniCAT database schema and saves it in the database. At this point, importers for the BPMAI and NPB are implemented. Later on, they can be used to update the framework's database with the latest changes of the external process model collection. According to their notation and format, some process models can be parsed into our process model for further analysis. possibly The *persistence API* captures and queries the underlying process model database OrientDB[3], which is a mixture of a document store and graph database. Further, the persistence API is able to store arbitrary Java objects such as analysis results or process metrics. All database objects need to inherit from *AbstractPojo* (pojo: plain old java object) to assure a database id, as shown in Fig. 2. Objects can be either synchronously or asynchronously loaded and PromniCAT provides a user layer on top of the database to map user functions to OrientDB-specific commands.

The *utility units* are executed within a *unit chain* to form a sequence to filter and parse process models as well as extracting further information. Each utility

[3] `http://www.orientechnologies.com`

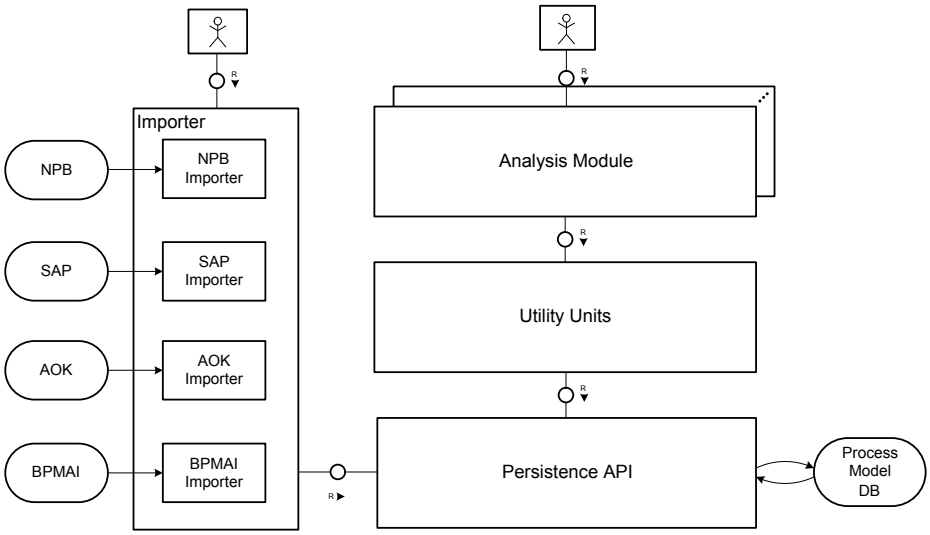


Fig. 1: Components and user endpoints of PromniCAT

unit encapsulates one single step towards process model analysis and can be used in multiple unit chains. The utility units of a unit chain are executed in a pipe to improve computation performance. The tasks for the different utility units break down into three aspects: Filtering, transforming, and extracting. Filtering utility units reduce the set of process models or other results, for instance regarding their database schema attributes, element types, label substrings, or connectedness value. Transforming utility units convert data, usually from JSON provided by BPMAI to our generic process model format. Extracting utility units calculate additional information, such as the element types or labels.

The *analysis modules* represent use cases by executing unit chains, saving and retrieving their analysis results. An example use case would be to calculate the model's connectivity value and clustering all imported models from different origins by these results.

2.2 Database Schema

The database schema consists of the classes *Model*, *Revision*, and *Representation* as depicted in Fig. 2 and is inspired by BPMAI. Each instance of Model groups several objects of the type Revision to account for changes. Revision objects again group instances of the class Representation to allow different notations or formats in the same context. Each Representation instance stores the detailed process description as a byte series and the path to the original file of this description. Further, each Revision instance has a revision number and a flag set if it is the last version of this group, as well as an author and human language of the labels and comments. The metadata offers storage of arbitrary key/values

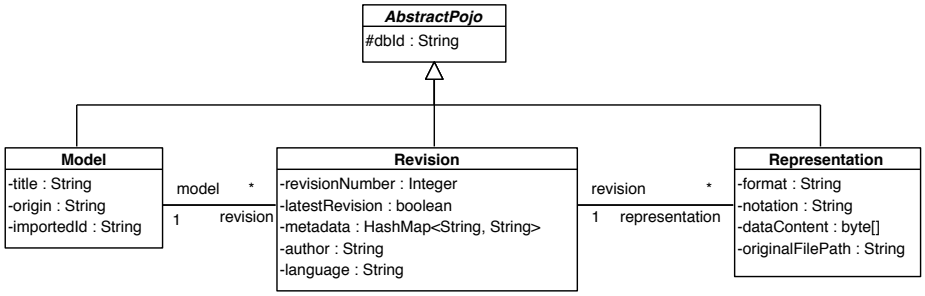


Fig. 2: The database schema: a model groups revisions and representations

but internally these are stored as key/value items with separator signs within the value. The model object has a title, an origin such as BPMAI or NPB and an importedId that is taken from the original data collection. The importedId is useful when the process model collection should be imported again to extend the Model objects with new revisions.

All these objects need to be stored in the database, hence they inherit from *AbstractPojo*. All importers have to conform to this database schema and the utility units need to be able to read it and convert the underlying data. Per default, the OrientDB loads all connected objects as well, but within the BPMAI data set an instance of Model might contain up to 80 Revision objects. Therefore, the persistance API also provides a *lightweight mechanism* to load a Representation object merely with its connected Revision and Model instances, but does not load all further connections.merely merely reliesh cua dat

2.3 Generic Process Model

A unique handling of process models from several collections and modelled in different notations requires an abstract mapping for these models. Amongst others, generic process models have been proposed by APROMORE [14] and jBPT [4]. The latter is a framework providing generic graph algorithms with focus on analyzing petri nets. In jBPT, a graph consists of a set of vertices and edges connecting them.

jBPT serves as basis for PromniCAT's generic process model, which is shown in Fig. 3. The vertices of a process model are separated into two types: FlowNodes and NonFlowNodes. *FlowNodes* are all objects that take part in the control flow of a process - this involves *Activities*, *Events*, and *Gateways*. Gateways are split up into three classes for and-, or-, and xor-semantic as well as a generic one, covering all other types. Following the argumentation given in [26], these are the basic types of process model elements being part of the control flow. *NonFlowNodes* thereby are those vertices that do not actively take part in the control flow such as data nodes. Further process model elements apart from

[4] `http://jbpt.googlecode.com/`

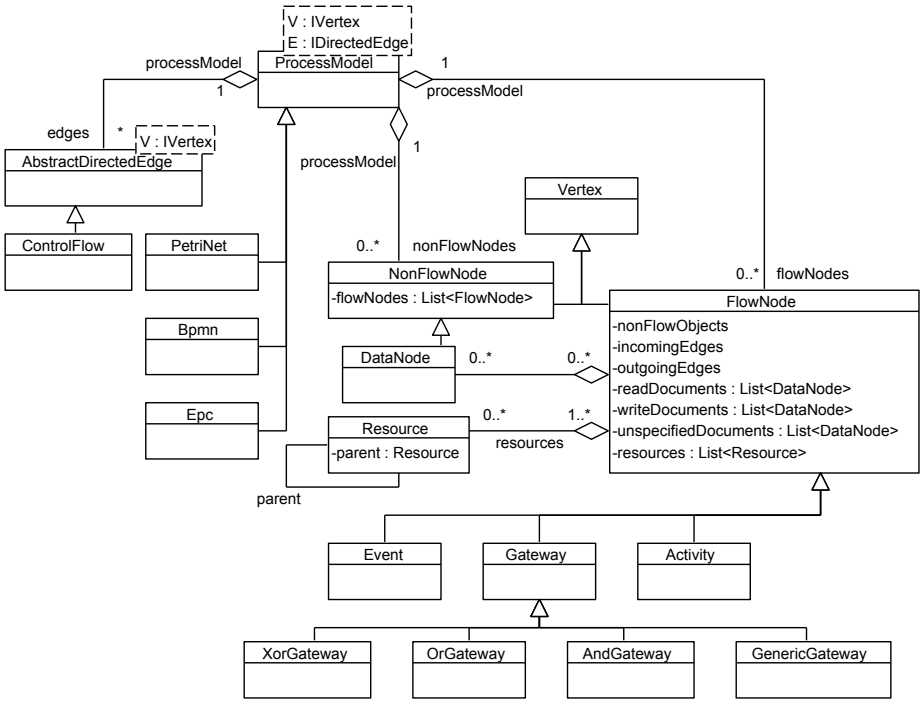


Fig. 3: Generic process model building on the jBPT class structure

Flow- and NonFlowNodes can be included as *Resources*. Currently, all kinds of roles in a process model are mapped onto this class. The control flow edges are stored explicitly within the process model whereas other kinds of edges, like data flow edges, are not modelled. These connections could be obtained from the *DataNodes* as well as from the FlowNodes, because these classes contain internal lists of the vertices they access or they are accessed by.

The aforementioned generic process model from jBPT can be easily extended to fit the constraints of a certain process model notation. Within jBPT, specific process models for EPC and BPMN are provided. These models extend the generic process model by special kinds of activities for EPCs, explicit storage of message flow edges, and start and end events for BPMN. Furthermore, BPMN requires a specialization of its control flow to handle attached events.

Parsing Process Models Those EPC and BPMN models given in the BPMAI JSON file format can be parsed into the specific jBPT representation with the help of a model parser integrated in PromniCAT. Depending on the concrete notation (EPC, BPMN 1.1, or BPMN 2.0), the corresponding model parser transforms the JSON format into an intermediate format with the help of the BPMAI Parser [5]. Afterwards, the jBPT representation can be created.

The model parser has the ability to detect control flow edges without a start or end node. It can be parametrized to either skip process models with such control flow edges or to ignore these edges and parse the rest of the process model.

[5] `http://code.google.com/p/bpmai/`

3 Process Metrics

This Section introduces the process metrics calculation approach realized within the PromniCAT framework. Firstly, we describe the integration of the process metric calculation algorithms into the existing framework. Secondly, an overview about the implemented process metrics is given. Thirdly, we present some specifics of the process metric calculation. Fourthly, an evaluation according to the execution time needed for process metrics calculation of the process models from the BPMAI is given. Finally, we propose process metrics gathering the aspects of data node and role usages.

3.1 Introduction

Metrics [1] are widely used in the field of software engineering to identify possible error sources of a software at the earliest possible point in time. Furthermore, they act as quality indicators of software systems, thus they make those systems comparable according to their quality. Software metrics are essential, because a simple code walkthrough to identify error-prone system parts is not realizable for software systems consisting of thousands or even millions of lines of code [20]. In addition to that, software systems must be extended over and over again. Therefore, often people are involved that have not written the original code which makes it even more complex to keep an eye on the system's architecture and existing design patterns [6]. These facts can be mapped onto the field of process modelling.

Today, companies have huge collections of process models reflecting a vast number of business processes. Take for instance the SAP reference model which consists of almost 10,000 process models [18]. With the help of process metrics it would be much more easier to quantify the quality of process models. Therefore, different aspects like size, modelling error probability and complexity of process models are reflected by process metrics. Furthermore, process model specific concepts like sequentially, decision points, concurrency and repetition are taken into concern. Besides, process metrics analysing the development of process models build the foundation of identifying process model creation patterns.

3.2 Integration into PromniCAT

The integration of the process metric calculation into PromniCAT is shown in Fig. 4. The existing framework has been extended by an analysis module called ProcessMetrics, an utility unit named ProcessModelMetricsCalculatorUnit, the process metric calculator class ProcessMetricsCalculator, and the unit data container UnitDataProcessMetrics.

The *ProcessMetrics* analysis module has been implemented to calculate process metrics for the EPC and the BPMN models of the BPMAI. Therefore, it uses a unit chain builder to create the following unit chain: First, a database filter is added. It ensures, that only EPC and BPMN process models from the

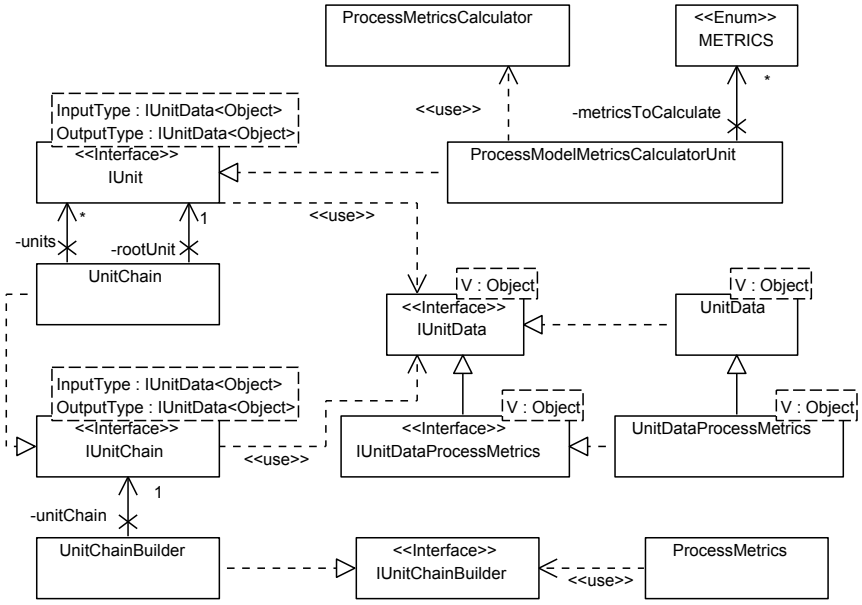


Fig. 4: Integration of the process metric calculation into PromniCAT.

BPMAI are used for analysis. Second, a *JsonToDiagramUnit* and a *DiagramToJbptUnit* are added to transform the process models given as JSON files into the jBPT representation explained in Section 2.3. Third, the newly implemented utility unit ProcessModelMetricsCalculatorUnit is appended to realize the process metrics calculation. Finally, a *SimpleCollectorUnit* is added, which collects all results. Thereafter, the created chain is executed and the process metric values for all available process models are calculated. Afterwards, the collected results are written into a CSV file that can be used for further analysis.

The *UnitDataProcessMetrics* class implements its corresponding interface *IUnitDataProcessMetrics* which extends the *IUnitData* interface by a getter and a setter for each process metric. Moreover, UnitDataProcessMetrics extends the *UnitData* class to get the stored process model. Hence, this unit data is a container for the analysed process model and its corresponding process metric values.

The *ProcessMetricsCalculator* class provides methods to calculate each process metric listed in Table 6 or only a subset of them given as list of *METRICS* constants. The *UnitDataProcessMetrics* instance holding the process model being used for process metric calculation as well as its corresponding process metrics has to be provided as parameter. Furthermore, the handling of subprocesses can be switched on or off by a parameter. The ProcessMetricsCalculator is stateless, thus the same instance can be used for process metrics calculation of different process models. In addition to that, each process metric value can be obtained separately by calling the corresponding method and passing the process model that should be analysed as well as a boolean that indicates, whether to include available subprocesses or not.

The *ProcessModelMetricsCalculatorUnit* class implements the *IUnit* interface, thus it can be used like any other utility unit. During the execution phase, process metrics are calculated using the ProcessMetricsCalculator. Subprocess handling as described in Section 3.3 can be controlled with the help of a parameter. Furthermore, this unit offers the possibility to define a set of process metrics being calculated during execution. These process metrics are given as constants of the *METRICS* enumeration which realizes the mapping to the method name used in the ProcessMetricsCalculator. If an empty set is given, all available process metrics are calculated during execution of the ProcessModelMetricsCalculatorUnit.

3.3 Implemented Metrics

The current implementation of the ProcessMetricCalculator provides algorithms to calculate 29 process metrics. We implemented 16 process metrics referring to the size of a process model by counting a well-diversified number of process model elements. The other process metrics capture various aspects concerning the structure of process models. An overview about the implemented process metrics and a textual description of them is given in Table 6. The implemented process metrics represent a subset of the process metrics introduced in [18], [17], [25] and [16]. We have chosen these process metrics for our implementation

regarding the aspect, that the process metric calculator is a prototype showing the feasibility of process metric calculation with PromniCAT. As it acts as proof of concept, we focused on process metrics analysing different aspects of process models. The analysis of the control flow behaviour played an important role. Amongst others, the counting of splits and joins belongs to this kind of process metric. Further process metrics, like diameter or separability, have been adapted due to the process model specifics described in the following paragraphs.

Handling of Subprocesses Subprocesses are only present in BPMN process models [21]. Nevertheless, they must be taken into consideration for process metric calculation. The handling of subprocesses can be switched on or off with a parameter in the constructor of the ProcessMetricCalculator. This offers the user the possibility to decide whether subprocesses should be considered or not. If subprocess handling is enabled, each subprocess is regarded as process model and the result value is added to the metric value of the originally given process model. This mechanism also allows the handling of subprocesses which contain further subprocesses. We call them stacked subprocesses. As a result, the process metric value of a process model with subprocesses consists of the process model's metric value and the sum of the process metric values of all of its subprocesses, even stacked ones. If subprocess handling is disabled, the underlying subprocesses of the given process model are not taken into consideration for process metric calculation.

Handling of Multiple Parts A process model can be split into multiple parts that are not connected by control flow edges. Hence, there exists two options to handle these kind of models. First, each part is handled separately and multiple results — one for each part — are calculated. Second, the given process model is handled as it is and only one result is calculated. The latter approach is realised within the current implementation of the ProcessMetricCalculator, because the PromniCAT framework assumes, that each JSON file represents a single process model instance. This assumption holds for process models being in an early state and that will be completed in further revisions. Besides, a process model can contain multiple modelling attempts reflecting the same business process which should be compared with each other. Moreover, modeller wants to have multiple views on the same process and put them side by side to achieve a better overview. Within the last two cases, the aforementioned assumption is violated, but these scenarios can be easily avoided by using independent process models for different modelling approaches which can be compared with the help of a process model editor or by putting them side by side manually. This can be done with few additional effort. A further benefit of splitting different views on the same process model into multiple files is an advanced support for collaborative process model editing, because multiple files can be edited by different people at the same time, whereas a single file can be changed only by one person at the same time.

As a result, we assume that each JSON file contains only a single process model. Nevertheless, all process metrics referring to the control flow of a process model had been adapted to handle incomplete process models consisting of multiple parts.

Handling of Missing Entries and Exits Further adaptations of the algorithms proposed in [18] have been made in the field of process models without an entry — a node without an incoming control flow edge — or an exit — a node without an outgoing control flow edge. The author of [18] assumes, that each process model have one entry and at least one exit. This assumption does not hold for all process models of the BPMAI. Consequently, some of the algorithms have been adapted to handle all kinds of process models, even those without entries or exits.

The separability metric has been changed to the number of nodes splitting a model into several parts divided by the number of nodes minus the number of entries and exits. Originally, the formula of the process metric has set the number of exits and entries to two, which does not hold for all process models.

The diameter metric has been adjusted too. If at least one entry and exit is given, it is calculated as defined in [18]. If multiple entries or exits or both is present, the longest path without visiting a node twice for each combination of entry and exit is calculated. In case of a missing entry, the diameter is calculated like mentioned before, except that the algorithm starts from each node of the process model without the exits themselves. For missing exits the diameter is calculated analogously starting from each entry. If either an entry nor an exit is given, the diameter represents the largest number of visited edges starting from any node of the process model and finishing at the moment a node has been visited twice.

Handling of Gateways The process metric algorithms proposed in [18] and [25] assume, that a gateway is either split or join. This holds for well formed EPC models, but in BPMN the same gateway can be both split and join. Furthermore, BPMN allows the modelling of activities with multiple incoming and outgoing edges [21]. Theses activities must be handled as gateways too. As a result, the algorithms for identifying splits and joins of a process model have been adapted to handle the aforementioned process model specifics.

Activities with multiple incoming or outgoing control flow edges have to be taken into consideration for calculating the number of splits and joins of a process model as well. Each activity with multiple incoming control flow edges represents a XOR-join and those with at least two outgoing control flow edges have to be handled as AND-splits, if the outgoing control flow edges do not have any conditions. Otherwise, they have to be considered as OR-splits.

As a result, the number of gateways is the sum of all splits and joins and does not equal the number of originally modelled gateways, if at least one gateway being split and join or an activity with multiple incoming or outgoing control flow edges is present.

3.4 Evaluation

The ProcessMetricsCalculator offers several process metric algorithms to analyse different aspects of process models. The process metrics values build the foundation for further analyses on process model collections. On the one hand, they are used as search criteria for selecting process models with the help of indices or as input for clustering algorithms, grouping process models with similar process metrics. On the other hand, process metrics are used for studies about the structure of process models within a certain process model collection. Take for instance the analysis results of the BPMAI published in [13].

In the following the process metric values presented in [13] are compared with those calculated with PromniCAT. The process metrics available in [13] as well as in PromniCAT are: number of nodes (*NN*), diameter (*Diam*), density (*Dens*), coefficient of connectivity (*CNC*), average connector degree (*Avg DR*), maximum connector degree (*Max DR*), and cycling (*CYC*).

	NN	Diam	Dens	CNC	Avg DR	Max DR	CYC
Results for BPMN models:							
Avg	14.46	7.30	0.09	0.76	2.01	2.23	0.05
Max	131.0	54.0	0.5	1.6	14.0	14.0	1.0
Results for EPC models:							
Avg	23.36	15.06	0.07	0.91	2.61	2.79	0.10
Max	123.0	67.0	0.5	1.1	5.0	5.0	0.9

Table 1: Subset of process metric values obtained with PromniCAT.

The process metric values calculated with PromniCAT are given in Table 1. They differ from those given in [13] due to different definitions of correct process models. In [13] all process models has been analysed. Only control flow edges without a source or a target node has been removed. In contrast, we decided to completely skip process models containing such an control flow edge, because the removal of those control flow edges leads to process models that do not reflect their underlying business processes any longer. As explained in Section 2.3, the filtering of erroneous process models can be done by PromniCAT's integrated model parser.

Apart from the aforementioned differences in chosing the process models to analyse, the average process metric values for the number of nodes, density, coefficient of connectivity, and cycling are approximately similar. The erroneous process models intensively influence the process metrics reflecting the number of control flow edges being connected with a gateway. As a result, the extensive usage of gateways seems to increase the probability of creating error-prone process models.

Execution Times The processing time to calculate all available process metrics for the EPC and the BPMN models of the BPMAI is about 12 minutes. We used a notebook with two 2.53 GHz processors and four gigabyte RAM. The used operating system was a 64 bit version of Microsoft's Windows 7. The test runs have been executed with Java 6 Update 29 and a 64 bit Java virtual machine with a maximum of one gigabyte heap space. Moreover, PromniCAT has been configured to use 12 threads for process metric value calculation.

	EPC	BPMN with subprocesses	BPMN without subprocesses
Loading, parsing, and process metrics calculation for all process models:			
execution time	76	833	681
number of process models	426	7035	7035
avg. execution time per model	0.18	0.12	0.10
Loading, parsing, and process metrics calculation for flawless parsed process models:			
execution time	72	215	175
number of process models	281	3540	3540
avg. execution time per model	0.26	0.06	0.05
Process metrics calculation for all process models:			
execution time	49	602	602
number of process models	426	7035	7035
avg. execution time per model	0.11	0.09	0.09
Process metrics calculation for flawless parsed process models:			
execution time	9	80	61
number of process models	281	3540	3540
avg. execution time per model	0.03	0.02	0.02

Table 2: Execution times for process metrics calculation in seconds.

An overview about the execution times measured with the notebook specified above is given in Table 2. On the one hand, we distinguish between the time needed to calculate the process metrics and the time needed for loading a process model from database, parsing it into its jBPT representation, and calculating the process metrics. On the other hand, we have drawn a distinction between the execution time needed for process metrics calculation of all EPC and BPMN models and those process models being correctly parsed regarding the criteria defined in Section 2.3.

Resulting from the measured values, we can draw the conclusion, that the loading of process models from database and their parsing into the jBPT representation takes only up to one third of the time needed for process metrics calculation. Furthermore, the handling of subprocesses has only a slight impact on the execution time, because subprocesses are only rarely used in process models of the BPMAI. Due to the fact, that EPC models consists of more nodes than

BPMN models (see Table 1), the process metrics calculation for a EPC model takes a higher effort than the processing of a BPMN model.

As a result, more than 1,200 process models per minute can be processed with the ProcessMetrics analysis module. The ProcessMetricsCalculator itself can handle more than 3,000 process models within the same time.

3.5 Future Work

The currently implemented prototype outputs the calculated process metric values as a CSV file which can be analysed either manually or with the help of other programs. Further integration with the PromniCAT framework can be achieved by persisting the calculated process metric values in the database used for process model storing. As a result, the process metric values can be easily used by other analysis modules and must not be recalculated each time they are needed. Due to the fact, that process models are not edited by PromniCAT, the process metric values will not change. Hence, they can be calculated and persisted during process model import.

The currently implemented process metrics depict various facts to make process models comparable according to the number of used model elements as well as their complexity. In contrast, the process model elements not being part of the control flow are not considered yet. Therefore, we would like to propose some process metrics concerning data nodes and roles.

Data Nodes To reflect the number of data nodes being used in a process model, process metrics for counting the data nodes being read, written, or both as well as those with unspecified access rights would be useful. In addition to that, the maximum and average number of accesses of data nodes could be calculated to support the quantification of the complexity of a process model. Moreover, the number of different states of a data node can be reflected by a process metric which would be helpful to improve the comparability of process models according to their complexity, because a process model with a stateless data node that is used several times is easier to understand than a process model with a data node in many different states. The most challenging part is the identification of different states due to the missing support in common notations like BPMN or EPC.

Roles As mentioned before, roles has not been considered in any process metrics of [18], [17], [25] and [16]. A first step into this direction can be achieved by counting the number of roles used in a process model. Therefore, a separation of the given roles into those representing organizations, like pools in BPMN, and those describing different positions within an organization, like lanes in BPMN, could be useful. In addition to that, the number of handovers between roles can be put into a process metric. As a result, this process metric offers the possibility to categorize process models according to the number of people, systems, and organizations being involved during execution. Moreover, the process models can

be classified according to the number of interactions between different organizations or systems. In addition to that, the aforementioned process metrics act as error predictors, because the process metric value could be easily compared with the number of roles involved in the business process reflected by a certain process model.

4 Indices

Search is an essential part of all collections. Convenient and fast search mechanisms make managing large data collections more convenient. Indices are usually used to speed up searches. In PromniCAT, indices are not only used to decrease wait time but also to store and load arbitrary analysis results.

One use case is the storage of various process model metrics, previously described in Section 3, for a given process model. One opportunity to store metrics is to extend the *Representation* class introduced in Fig. 2 to hold additional results as new class members. But extending the database schema to arbitrary class members is a difficult task, especially in a database which is possibly used by multiple users. So we consider the database schema with Model, Revision, and Representation fixed. We can use indices instead to store these results as described in Fig. 5. It visualizes the use case of storing the number of data nodes, the connectivity, and the labels of all nodes for some Representation objects within three indices.

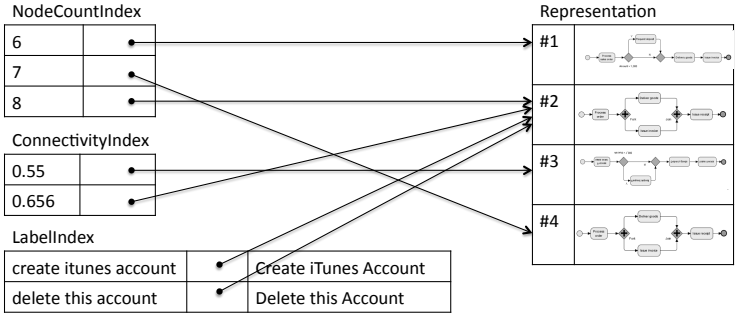


Fig. 5: Indices can be used to store analysis results for Representation instances such as the number of nodes, the connectivity, and the labels within the nodes

Indices are basically stored as key/value pairs, such as *no. nodes* $\rightarrow$ *pointer to process model*. The pointers in all indices are realized as database ids. The exemplary LabelIndex shows, that we needed to extend this key/value structure to allow case-sensitive and case-insensitive string information. It further shows that some indices can have multiple entries referencing the same object. Common drawbacks of using indices is the additional space needed to store the index as well as a little longer insert or update time to extend the index.

This use case could easily be extended by a wrapper class `MyResults` that would combine all results for one instance of Representation with four members:

- `noOfNodes` as Integer,
- `connectivity` as Double,
- `labels` as String collection or array, and
- `dbId` as String containing the referenced database id.

With OrientDB being the underlying database, it is possible to store this class in the database directly by the built-in object mapper. The three exemplary indices inroduced above would point to the instance of MyResults instead which in turn would point to the Representation instance. To finally load specific results from the database, OrientDB requires special commands such as:

```
SELECT myResults FROM index:LabelIndex
WHERE label LIKE '%account%' or label LIKE '%Account%'
```

This is called NoSQL (Not Only SQL [23]) as it adopted most of the commands used in SQL (Structured Query Language) and offers additional commands that fit to the underlying structure instead of a relational database that SQL was designed for. PromniCAT builds on these NoSQL commands and allows custom case-insensitive search like `myLabelIndex.selectContains("account")` with predefined search mechanisms like numbers within a range or a regular expression for strings. Though the number of implemented methods is limited, the user does not need to learn the specific NoSQL dialect of OrientDB.

This chapter will describe the underlying database with some of its features first. Afterwards, the difference in the implementations of using numerical keys, string keys and providing arbitrary values are described. Using these indices, multiple search criteria can be combined. We close with some ideas for future work.

4.1 Preliminaries in OrientDB

OrientDB was chosen as the underlying database which is accessed by the persistance API.

OrientDB is a combination of document store and graph database. Document stores use semi-structures to materialize their data, this means they consist of recursive key/value pairs [15]. In the case of OrientDB, these are JSON files, which are flexible enough to be mapped to a large number of Java objects as each class member has a name (which can be used as key) and a value connected to it. Graph databases are designed for fast traversal between database objects without possibly costly joins as used in relational database management systems [3]. They further provide query mechanisms such as the following in OrientDB:

```
SELECT FROM representation
WHERE revision.model.origin like 'BPMAI'
     and notation like 'EPC'
```

This query retrieves all Representations that are connected to a model with the origin 'BPMAI' and that are written in the notation 'EPC'. Here, the reference from one Revision to its Model is stored explicitly within the Revision as `model: modelDbId` and as a list of database ids `revisions: [revisionDbIds]` inside the Model object for all available Revisions.

In addition to the direct storage of connected database ids, OrientDB is required to load the referenced objects fast as well. First, they partition the

database entries by their classes into *clusters* which are stored as separate files on the disk. Having multiple smaller files instead of reading from one large file might be an advantage when reading files sequentially. The database ids already contain the number of the cluster explicitly so that the database management system can easily access the correct file. Second, they speed up the access within each file by their so called *MVRB-Tree*. A quote from their website says:

> *OrientDB uses a new indexing algorithm called MVRB-Tree, derived from the Red-Black Tree and from the B+Tree with benefits of both: fast insertion and ultra fast lookup.*

The Red-Black Tree as well as the B+Tree are both binary search trees. The *Red-Black Tree* is a balanced tree, this means that look up time from the root is nearly constant to all available leaves [22]. But this approach requires some computation time during insert. In the *B+Tree* entries are stored within the leaves only instead of nodes. Nodes can point to multiple nodes, and nodes and leaves within one level form a connected list by pointing to the following node or leave, respectively [11] which makes look up extremely fast.

OrientDB uses this MVRB-Tree mechanism to load objects fast and to index them. In the remainder of this chapter, we show how we build on OrientDB and their implementation to provide an index API to the user.

4.2 Implementation as Key/Value Pairs

In PromniCAT, keys can be a numerals or strings, values must be any kind of object that has a database id. The `NumberIndex` and `StringIndex` are used for the numerical and string keys respectively. As described in Fig. 6, both inherit from `AbstractIndex`, both allow only `AbstractPojos` as values, and both return `IndexElements` when `load()` is called. Their implementation, however, differs greatly.

The overall indexing API is provided by the AbstractIndex. The index name is of high importance, it serves as the unique identifier for the index. Moreover, an index can only be created if the name is not yet reserved. All subclasses provide different selection mechanisms based on their nature. Whenever a selection criterion is set, they update the `noSql` command accordingly. This way, only the last selection criterion will be evaluated. If no criterion is set, all entries in the index are loaded per default. When the entries are loaded, they are converted into a set of IndexElements. Each IndexElement provides access to the key, the database id of the value and the value as fully loaded object. Java generics allow access to various types for key and value without additional class casting.

Arbitrary Values: While restricting the key to numericals and strings, the index values can be nearly arbitrary. As mentioned above, PromniCAT allows to store arbitrary Java objects that inherit from AbstractPojo to include a database id. The types of attributes are limited though to Java primitives and their object representations, as well references to other database classes, Collections of them

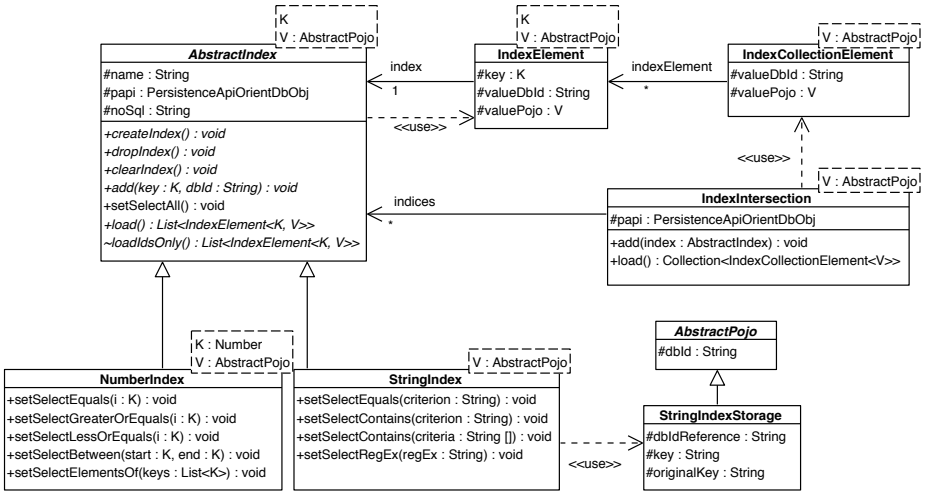


Fig. 6: The NumberIndex and StringIndex return IndexElements on load(). Indices can be intersected to return IndexCollectionElements

or specific Java classes like Date. These pojos can be saved and retrieved by their database id like this:

```
String dbId = papi.save(pojo);
AbstractPojo loadedPojo = papi.load(dbId);
```

This database id itself can be saved in an index with a specific key or in any other pojo that serves as a result collector.

Numerical Keys: The NumberIndex can be used with any kind of Java Numeric, such as Integer, Double, Long and Float and a number of typical math comparators such as $=, >=, <=$. An example of storing and loading the connectivity value of some given Representation objects is shown below:

```
NumberIndex<Double,Representation> connectivityIdx =
    new NumberIndex<Double,Representation>("myConnectivityIdx", papi);
connectivityIdx.create();
connectivityIdx.add(connectivityValue, representation.getDbId());
connectivityIdx.setSelectBetween(min,max);
List<IndexElement<Double,Representation>> results =
    connectivityIdx.load();
```

For each IndexElement in the `results`, it is possible to access the key as the specified Double and the value as Representation without casting. Java generics assure that all parameters are provided using the correct type, since OrientDB is not able to deal with a mixture of number types. When using numbers, the manual indexing mechanism from OrientDB are sufficient, this means most methods

are merely mapped to specific OrientDB commands in NoSQL notation. In this example, the selection would be mapped to the internal NoSQL query:

```
SELECT FROM index:myConnectivityIdx WHERE key BETWEEN <min> AND <max>
```

The method `selectElementsIn(List<Number> l)` finds all values contained in the given list l of length n with the internal mapping to

`SELECT FROM index:myConnectivityIdx WHERE key IN [<`l_o`>,<`l_1`>,...,<`l_n`>]}.`

String Keys: The StringIndex can be used to store string keys such as all labels of the model's elements. It allows selection criteria like:

- `contains("create account")` for one substring,
- `contains(["create", "account"])` for multiple substrings,
- `equals("create itunes account")` for strict equality, and
- `regex(".*(user|account).*")` for flexible regular expressions.

A StringIndex with these features could not be implemented as a manual index like the NumberIndex, because only `key LIKE 'x'` is supported. This means, it could only provide equality search and substring search with only one substring.

To support more search features on texts, OrientDB allows where clauses of queries to be in the form of `key CONTAINSTEXT 'account'` and `key MATCHES '.*(user|account).*'`. This eanbles search for equality, a substring and a regular expression, although multiple substrings are not accomplished without complex regular expressions and the search is not case-insensitive automatically. The most important drawback with this technique, however, is the requirement to use automatic indices instead of manual indices. Automatic indices in OrientDB are created for a specific class with a defined key, such as `StringIndexStorage` with its member `key`. These indices are automatically extended or updated whenever an instance is saved as composed to manual indices that need to be updates with a `INSERT INTO index:<name>` command. This might be helpful in most indexing cases but it is not convenient whenever many different indices are made up of different instances of the same class, such as multiple instances of StringIndexStorage. This would imply that two different indices would both be using StringIndexStorage and therefore would both be retrieving all entries of the other index as well.

Finally, the implemented solution for all presented needs is to use OrientDB `clusters` instead of indices. Per default, each Java class is mapped to its own cluster, which is one file on the disk. It is further possible, to define additional clusters for a class, so that StringIndexStorage can be manually stored in different clusters and loaded via `SELECT FROM cluster:<clusterName>`. This does not allow to use the `CONTAINSTEXT` command, though. But `MATCHES` is still possible as well as `key LIKE 'x' AND key LIKE 'y'` or the `=` operator. Since all clusters are stored as a separate file on disk, access time is only influenced by the cluster, this means the current index, itself. The string operators `MATCHES` and `LIKE`

are only possible, when the class member `key` is defined as a string property beforehand.

Overall, search should usually be case-insensitive but the user might rather use the original case values for further analysis. To account for this need, StringIndexStorage contains both versions. The case-insensitive version is used as the key, whereas the case-sensitive version is called `originalKey`. The originalKey is used to create the resulting IndexElement given to the user. Every search criterion itself needs to be converted to lower case as the internal matching of OrientDB is case-sensitive again.

Searching for symbols reserved by NoSQL such as (,), or `SELECT` are possible as well. Stop words are also taken into account and possible to search for.

4.3 Multiple Search Criteria

Another use case of retrieving analysis results can be based on multiple criteria. This needs to incorporate multiple indexes. In case, these criteria are combined with an `and` connector, the overall result set is an intersections of both individual indices' result sets. Per default, both individual indices return a set of IndexElements with fully loaded value objects. To avoid that all objects are loaded, we intersect the resulting database ids of both indices before loading the left over set of ids. Note that one index can contain multiple keys for one value. This is the case with multiple labels for one process model. Again the goal was an easy-to-use interface and the exact usage is presented below. Each resulting `IndexCollectionElement` contains one referenced database id with a fully loaded pojo together with multiple IndexElements for the various individual results pointing to the same object. With this IndexCollectionElement it is possible to group all selected analysis results for one pojo.

```
labelIndex.setSelectContains("account");
nodeCountIndex.setSelectGreaterOrEquals(min);

IndexIntersection<Representation> intersection =
    new IndexIntersection<Representation>(papi);
intersection.add(labelIndex);
intersection.add(nodeCountIndex);
Collection<IndexCollectionElement<Representation>> result =
    intersection.load();
```

4.4 Future Work

Both NumberIndex and StringIndex could well incorporate more methods for selection such as mathematical operators and further string comparators. The StringIndex only uses clusters at the moment, although some selections can be accomplished with manual or automatic indices provided by OrientDB. It might be possible, that a combination of both cluster and internal indices perform even better. Instead of an intersection, a union or difference could be implemented similarly by loading the resulting objects only after reducing the amount of database ids to load. It would be usefull, to update a selected set of indices with some metrics and other common search criteria such as the labels of nodes already during the import of process models, because each model needs to be loaded into memory anyway.

5 Clustering Process Models in PromniCAT

Analyzing process models does not only include examining each process model on its own. Rather *comparing* and *characterizing* process models and their features puts the different results in relation to each other. Process models can further be classified by certain criteria to give an indication about the actual quality of their features. Clustering process models is a commonly used strategy to achieve such a classification.

The clustering process breaks down into three main parts: Preprocessing the data by retrieving the features required for clustering from each process model, clustering the process models by formerly defined criteria, and labeling the resulting clusters with expressive titles. This Section deals with the realization of these three processing steps to facilitate process model clustering in PromniCAT.

5.1 Retrieving Feature Vectors from Process Models

To determine the similarity of two process models during the clustering process, they are compared regarding the values of their formerly selected features. For each process model, these feature values are represented in a feature vector $\boldsymbol{f_i}$ that is assigned to a process model i: $\boldsymbol{f_i} = (a_1, \ldots, a_j, \ldots, a_n)$, for $n = |\text{selected features}|$ and $a_j =$ value of feature j. The feature a_j can be of numerical (e.g, a process metric value) or string (e.g., a process' activity's label) nature.

This preprocessing step has to be executed before the actual clustering takes place. For the integration into PromniCAT, retrieving a feature vector for each process model involves the creation of a new unit chain, providing a possibility to select appropriate features as well as creating a representation of the actual feature vector.

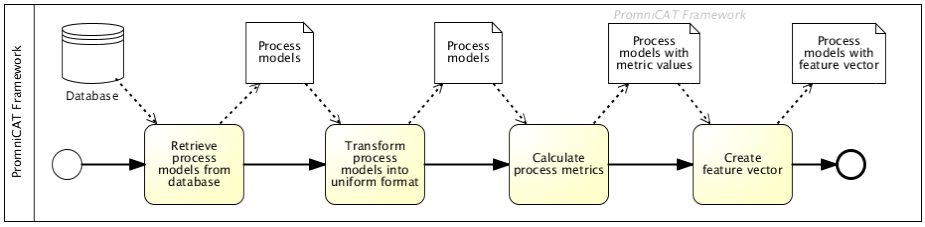


Fig. 7: Workflow for retrieving the process models' feature vectors

At the current state of the implementation, the features represent structural characteristics of a process model. This includes the different process metrics as presented in Section 3 as well as the labels of different elements of a process model, for instance a process' name. All these values are retrieved by executing a specific workflow, which is modeled in BPMN 2.0 in Fig. 7. At first the actual process models to cluster have to be selected by their origin, format, or notation

from the database. Afterwards, these process models have to be transformed into a uniform format to be comparable. In the next step, the desired process metrics have to be preprocessed for each process model. The last step creates the actual feature vectors from each process model for the selected features. The resulting output has to contain the process models and their feature vectors to continue with the clustering.

Setting up the Unit Chain For the realization in PromniCAT, the different workflow steps shown in Fig. 7 can be directly mapped onto utility units. The first two — retrieving process models from the database and transforming them into a uniform format — have been implemented in previous work, since they are essential for most of all created unit chains. The third utility unit calculating all process metrics has been implemented within the scope of the work presented in Section 3. Up to this point, a computation of the string feature values beforehand is not necessary. The string features currently taken into account do not require a complex calculation as for the process metrics and can be directly retrieved from the process models. Nevertheless, this aspect should be kept in mind for the implementation of further and more sophisticated label retrievals. The utility unit for the last execution step has been implemented whithin this work and its details are covered by the next Section.

Before the unit chain is executed, it has to be customized according to the specific user demands. To this belongs defining selection criteria for the process models as well as the configuration of the feature vector for the later clustering process. Since the former functionality is already provided in PromniCAT, the implementation of the latter belongs to the scope of this work.

For that, two sets of constants are introduced: One covering all indicators for the numeric features — by now, merely the process metrics —, and another set containing all indicators for the string features. Both sets are captured in the class *ProcessFeatureConstants* shown in Fig. 8 and can be easily extended by new features. At this point, no weights are assigned to the features. This is done later at the setup of the clusterer.

Creating a New Utility Unit The new utility unit takes the output of the former unit as input, i.e., the process models and the calculated values for the selected process metrics. The task now is to retrieve the string feature values directly from the process models and combining them with the already calculated feature values within a feature vector. Thus, the output of the utility unit contains the current process model and its feature vector.

Fig. 8 depicts the implementation's underlying class structure in an UML [4] class diagram. Highlighted classes indicate the additions made to the already existing class structure. The new utility unit is called *ModelToFeatureVectorUnit*, and owns a *config* attribute of the type *FeatureConfig*, containing the names of all selected features. These values are set in advance at unit chain creation.

As all utility units, the ModelToFeatureVectorUnit implements the interface *IUnit*. During the execution of the new utility unit, it evaluates the given

configuration and collects the corresponding feature values. The actual feature value retrieval is encapsulated in the constants class ProcessFeatureConstants, since it maintains the selectable features in enumerations which also return the corresponding values.

The feature values are represented by instances of the class *Feature*, whereas the feature vector is modeled by the class *FeatureVector*. It has an attribute *features* holding a list of instances of Feature. The resulting instance of FeatureVector is then annotated to the output of the utility unit: the *UnitDataFeatureVector*, which inherits from *UnitData*.

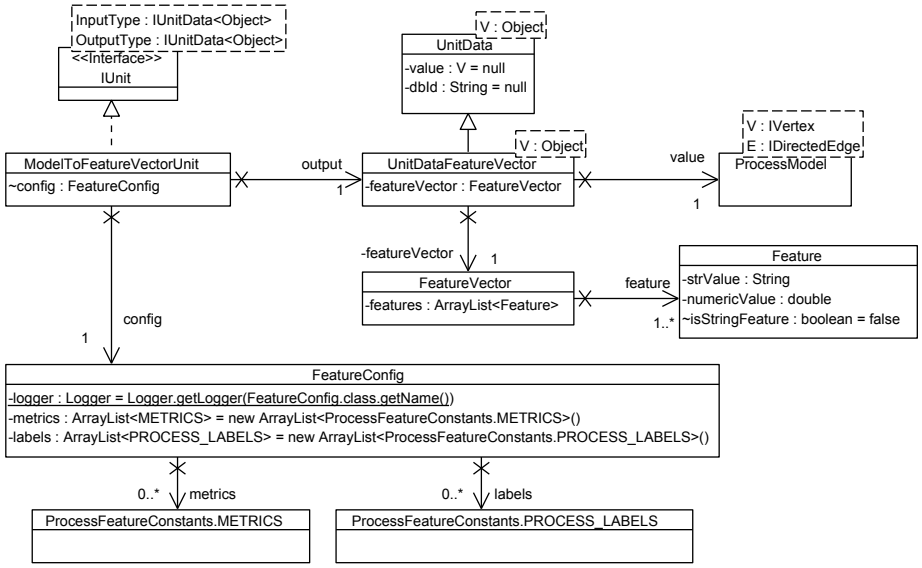


Fig. 8: Class structure for the implementation of the new utility unit

5.2 Extending WEKA's Hierarchical Clusterer

After the unit chain has been executed, each process model owns a feature vector and is thus prepared for the actual clustering process. Two clustering methods often applied in practice are *partitive* and *hierarchical* clustering [27,10]. The former results in a number of distinct clusters, whereas hierarchical clustering returns a clustering tree (*dendrogram*) where each node represents a cluster composed of its sub-nodes. k-means is the most popular partitive clustering algorithm, but has an indeterministic behavior. I.e., for two clustering executions k-means would result in two different clusterings for the same execution parameters. Hierarchical clustering is divided into a top-down (*divisive*) and bottom-up (*agglomerative*) strategy. The latter enables users to also navigate through the

clusters' different hierarchy levels, which is why we chose to implement hierarchical agglomerative clustering in PromniCAT. Due to space limitations, we spare a pseudocode description of the algorithm and merely reduce its description the essential execution steps:

1. Start with each process model i in an own cluster c_i.
2. Compute the distance $d = dist(c_i, c_j)$ between all clusters c_i and c_j.
3. Merge the two clusters with smallest d.
4. Repeat steps 2 and 3 until there exists a single cluster containing all process models, which represents the root of the resulting dendrogram.

The distance measure $dist(c_i, c_j)$ encapsulates the comparison between two clusters c_i and c_j. For that, at first the actual feature vectors $\boldsymbol{f_i}$ and $\boldsymbol{f_j}$ have to be selected for comparison. This can be achieved by choosing a representative from the different feature vectors within in a cluster (*single*, *average*, *complete linkage*) or creating a new representative from all of them (*centroid*, *ward's method*) [8]. The actual distance of the feature vectors $\boldsymbol{f_i}$ and $\boldsymbol{f_j}$ is then computed via a distance function $df(\boldsymbol{f_i}, \boldsymbol{f_j})$, such as the *euclidean* or *manhattan distance* [8].

There already exist lots of implementations of the hierarchical clustering algorithm in different data mining libraries such as HAC[6] [2], ClusterLib[7] or WEKA[8] [9]. In terms of using learning or classification approaches for future work in cluster labeling, we decided to use the open-source WEKA library for our purposes. WEKA includes a number of machine learning algorithms for data mining tasks; containing tools for classification, clustering, association rules, and visualiziation.

WEKA comes along with an implementation of an agglomerative clustering algorithm with a complexity of $O(n^2 log(n))$, and implements common distance measures introduced before. Some distance functions such as euclidean or manhattan distance are also provided. Nevertheless, these implementation do not fully suffice our demands. WEKA's hierarchical clusterer does not support a normalization of the numeric feature values. Additionally, WEKA considers no weights during the distance calculation in the clustering process, which implies all attributes to be treated equally. Moreover, WEKA can only cluster numeric values, whilst we also want to incorporate label analyses such as the process name for clustering. Additionally, WEKA's output format of the resulting dendrogram has to be adapted for our needs. Since WEKA uses customized data structures in their framework, we do not only have to map the output of our unit chain onto their structures. Rather, we have to enhance these structures by the following aspects: Incorporating normalization of numeric feature values, supporting weight assignment to the features, clustering with features of both numeric and string nature, and presenting the clusters in an appropriate output format.

[6] `http://sape.inf.usi.ch/hac`
[7] `http://niels.drni.de/s9y/pages/clusterlib.html`
[8] `http://www.cs.waikato.ac.nz/ml/weka/`

Normalizing Feature Values The normalization of feature values is essential for their comparability. When comparing two values, the significance of their distance can not be determined without knowledge of the overall values. For instance, two process models a and b are compared by the amount of their activities. a has 5 activities, whereas b owns 35. Comparing only these two values, the difference between them seems to be significant. But in the data set, there exist other process models c and d, which include 1000 and 400 activities. Compared to their distance, suddenly the difference of 30 activities between a and b seems to be much less significant than the difference of 600 activites between c and d. By determining the values' actual boundaries and putting them in relation to each other, the significance of a current distance value is clarified at once.

The normalization process takes only place for all numeric feature values, and is executed in two phases: Determining the maximum and minimum value for each feature, and normalizing each feature value to the interval $[0 \ldots 1]$ when computing the distance. The first step is directly executed after retrieving the results from the unit chain by iterating over all feature vectors. The result is a collection containing for each feature the maximum and minimum values of the result set. To avoid another iteration over all feature vectors, the feature values retain their original values whilst performing the normalization in the distance calculation. To achieve this, the distance function has to be adapted by the possibility to commit the features' boundary values as well as considering them in the distance calculation. For the usage in PromniCAT, we implemented these changes for the euclidean distance function by introducing the new class *WeightedEuclideanDistance*, which inherits from WEKAs *EuclideanDistance* class.

Weighting Selected Features Assigning weights to the different features of a process model enables users of PromniCAT to customize the clustering process according to their own demands. WEKA already supports weighting in their classifying functionalities via so-called *Attribute* classes, which can be reused for our purposes. In the context of PromniCAT, an attribute describes the meta level of a feature, owning the feature name and its weight. Attributes are assigned to each instance of the class ProcessInstance to be accessed for interpretation of the feature vector. In the clustering process, the weights are incorporated in the calculation of the distance of two feature vectors. This involves adaptations to all distance functions used in the clustering, thus implementing the new distance classes *WeightedEuclideanDistance* and *WeightedEditDistance*, which extend WEKAs *EuclideanDistance* and *EditDistance* classes.

Clustering with Strings As WEKA supports only clustering with numerical, i.e., *double* values. In order to enable a clustering with string values, we extend WEKA by the corresponding functionality to match our requirements. For an object that is to be clustered, the numerical features are assigned to a variable containing the feature values in an array of doubles. Since this data structure has been specified by WEKA and thus can not be altered, the string feature values

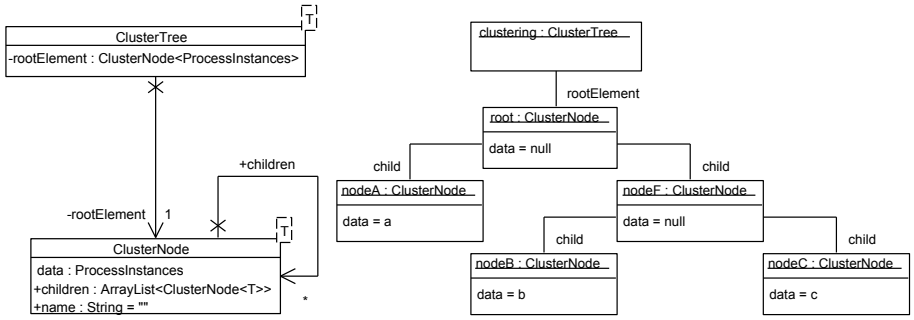
have to be hosted separately in another variable. Thus, we have to extend the existing classes to cope with a new attribute containing the string features in an array of strings. This strict division into string and numerical features includes an overall separate handling of these two different feature types: Creating, setting, and retrieving the feature values and calculating the distance between two feature vectors has to be implemented separately. Whilst the realization of the former aspect is straightforward, the latter requires more efforts. This involves the usage of a further distance function in addition to the numeric one. By applying two different distance functions for the comparison of two feature vectors, their results again have to be combined to achieve a final distance result. That is, the string distance function computes the overall distance of all string, the numeric distance function computes the distance for all numeric features. When using both numerical and string features, the final distance result results from the average of both values.

In addition to the simple distance calculation of two feature vectors, some of the measures for the calculation of the inter-cluster distance need to be adapted to also consider the string features. That is the case for centroid and Ward's method, since the changes necessary to the other strategies are already covered by the adaptions made to the distance functions. The centroid method creates a new representative from the feature vectors contained in a cluster by retrieving the average — the *centroid* — from the feature values. Ward's method calculates the distance of the change that would be caused by merging the current two clusters by comparting the two clusters' centroids. Both strategies need to find an average value for each feature, which is trivial for numerical values. For string attributes, this strategy does not apply.

Instead, we have to choose an appropriate representative of all the values for each string feature in particular to create the centroid. For that, all values of a specific feature within the cluster are compared to each other. The feature value with the lowest overall distance — since a high distance value indicates less similarity — is selected for the representative. The distance between the feature values is calculated by the same distance function that is used for the actual feature vector comparison.

Returning the Clustering Results The output of a clustering execution is a dendrogram of the clustered process models. Thus, clusters have to be presented in a format that allows data exploration and further analyses. The only output of clustering results provided by WEKA is a string representation in Newick tree format [7], which is not applicable for our purposes since we would have to find adequate printings for each process model. The internal structure used by WEKA for clustering is also not sufficient, since it is a binary tree consisting of indices for the clustered objects. Nevertheless, this can be used as basis for a structure containing the actually clustered process models and their feature vectors. For that, we introduce a tree structure as despicted in Fig. 9a. The clusterer returns an instance of the class *ClusterTree*. It holds a reference to the root node of the tree and encapsulates further functionality for accessing

and transforming the tree and its elements. The tree's nodes are instances of the class *ClusterNode*. Each node represents a cluster, again composed of the clusters their child nodes represent. Since a clustering can also end up in several instances of ProcessInstances lying in their own clusters, the root node of the dendrogram can potentially have more than two child nodes. Additionally, the actual ProcessInstances objects with their process models are only contained in leaf nodes, but the ClusterNode class offers functionalities to retrieve the actual cluster elements for each node. Fig. 9b models an example cluster tree in an UML object diagram with three instances of the class *ProcessInstances* a, b, and c containing the process models and their feature vectors. a is situated in its own cluster, whilst b and c are contained in the same cluster, again consisting of two sub clusters.



(a) Dendrogram class structure (b) Object diagram of an example dendrogram

Fig. 9: Class structure and example instantiation for dendrograms

5.3 Labeling the Resulting Clusters

Labeling clusters with suitable descriptors is important for an efficient navigation through a dendrogram. By having a look at the cluster's label first, the user gets a notion of the process model contained in the cluster without examining the actual data. Thus, he can immediately decide whether it is worth to conduct a further exploration of the cluster or not. In this work, we primarily focused on the data's preprocessing and clustering process, which is why we now present an elementar approach for cluster labeling.

Since the distance of two process models is determined by comparing feature vectors, it is obvious to create a cluster label that is composed from representative values of each features. For that, we have to realize two strategies for retrieving the representatives for the numerical and string features.

The numeric feature values allow an easy labeling by generating closed intervals from them. This means that the values of a specific feature from all the process models contained in a cluster are situated within the range of the interval

depicted by the feature's representative value. For instance, a cluster contains three process models a, b, and c that are clustered by their amount of activities. a owns 5, b has 2, and c holds 7 activities. The resulting interval of these values would be in [2-7]; the cluster's label would thus be "*#activities:* [2-7]".

Obviously, the interval strategy does not apply for the string features. Therefore, we remove the stop words with WEKA's stop word list from each string value of a specific feature and count the word frequencies. The two most frequent words are then selected to be the representative for the current feature. We chose to select two instead of only one word, since one single word lacks a sufficient expressiveness. Having a second word makes the context of the feature's label clearer. To give an example, the aforementioned process models are again clustered, this time by their amount of activities and their process titles. The title of a is "Account Managing 1", b is called "Account Validation", and c has "Account Managing 2" as title. With 3 occurances, "Account" is the most frequent word, followed by "Managing" with the count of 2. The representative value for the title feature would thus be "Account Managing". Toghether with the other feature, the final cluster label would be "*title:* Account Managing; *#activities:* [2-7]".

In both strategies, instead of only considering the child clusters' labels for cluster name creation, the feature values of the actual process models contained in the cluster are considered for selecting a representative value. Nevertheless, a descriptive labeling requires more advanced strategies such as learning approaches or classifiers, especially for the creation of suitable representatives for the string features.

5.4 Future Work

The work constituted in this Section is certainly far from being complete; it rather establishes the basis for further improvements of this analysis module. Consequently, the different components presented here offer several possibilities for future work.

For instance, this could include the usage of indices for improving the performance of the clustering algorithm. Furtermore, other distance or similarity measures should be implemented to incorporate behavioral characteristics of process models in addition to their structural aspects. [24] already presented an approach to measure behavioral similarity between process models, which could be applied for usage in PromniCAT. Next to that, the cluster labeling presented in this work is only a simple approach. A more sophisticated algorithm for labeling clusters would be more sufficient for a satisfactory result. For instance, the work from [19] also takes into account the hierarchical relations between clusters for a suitable labeling and could serve as starting point for further improvements.

6 Conclusion

In this paper, we introduced PromniCAT as a framework to integrate process models from various sources into one knowledge base. PromniCAT owns a generic process model which provides an abstraction of the specific formats and notations for process models. Via the unit chains and their utility units, our framework offers various functionalities to efficiently extract informations from process model collections according to customized criteria. In addition, PromniCAT allows extending the framework by further analysis modules.

Process metrics calculation has been integrated into PromniCAT via a new utility unit. In the current implementation, a large set of process metrics is provided covering several aspects of process models like size, control flow complexity, and error probability. Moreover, these process metrics can be extended by metrics reflecting the usage of those modeling elements that are not part of the control flow, such as data nodes or roles. Nevertheless, the provided set of process metrics builds the foundation for further analyses of process model collections and can be easily used for the classification of process models.

The persistance API has been extended to accomplish indices as key/value pairs to store nearly arbitrary analysis results. Numerical keys can be used with $=, <=, >=, between$ and $elementOf$ operators to select items from the index. Selecting indices with string keys can be based on equality, one or multiple substrings, or regular expressions. It is further possible to intersect multiple indices before the database objects are fully loaded.

For the clustering of process models, a new utility unit has been introduced to preprocess these process models for the actual clustering execution. By extending WEKA's hierarchical agglomerative clusterer to deal with feature normalization, weighting, and the clustering according to both numeric and string features, we enable users to classify and navigate through process models collections that are set up by customized criteria. Additionally, by establishing an elementary cluster labeling approach, we provide the basis for further sophisticated cluster labeling algorithms.

References

1. *IEEE Standard 1061-1998, Standard for Software Quality Metrics Methodology.* New York: Institute of Electrical and Electronics Engineers, 1998.
2. Andrea Adamoli and Matthias Hauswirth. Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports. In *SOFTVIS '10: ACM Symposium on Software Visualization*, pages 73–82, October 2010.
3. R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.
4. G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, 2005.
5. R.H. Eid-Sabbagh, M. Kunze, and M. Weske. An open process model library. In *Business Process Management Workshops*, pages 26–38. Springer, 2012.

6. Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
7. J. Felsenstein, J. James Archie, WHE Day, W. Maddison, C. Meacham, FJ Rohlf, and D. Swofford. The newick tree format, 1986.
8. G. Gan, C. Ma, and J. Wu. *Data clustering*. SIAM, Society for Industrial and Applied Mathematics, 2007.
9. Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.
10. A.K. Jain and R.C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.
11. C.S. Jensen, D. Lin, and B.C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 768–779. VLDB Endowment, 2004.
12. G. Keller, M. Nüttgens, and A.W. Scheer. Semantische prozessmodellierung auf der grundlage "ereignisgesteuerter prozessketten (epk)". *Veröffentlichungen des Instituts für Wirtschaftsinformatik*, 89, 1992.
13. M. Kunze, A. Luebbe, M. Weidlich, and M. Weske. Towards understanding process modeling–the case of the bpm academic initiative. *Business Process Model and Notation*, pages 44–58, 2011.
14. M. La Rosa, H.A. Reijers, W.M.P. Van der Aalst, R.M. Dijkman, J. Mendling, M. Dumas, and L. García-Bañuelos. APROMORE: An advanced process model repository. *Expert Systems with Applications*, 38(6):7029–7040, 2011.
15. J. McHugh and J. Widom. Query optimization for semistructured data. Technical Report 1997-19, Stanford InfoLab, 1997.
16. J. Melcher and D. Seese. Visualization and clustering of business process collections based on process metric values. In *Symbolic and Numeric Algorithms for Scientific Computing, 2008. SYNASC'08. 10th International Symposium on*, pages 572–575. IEEE, 2008.
17. J. Mendling. Testing density as a complexity metric for epcs. In *German EPC workshop on density of process models*, 2006.
18. J. Mendling. Event-driven process chains (epc). *Metrics for Process Models*, pages 17–57, 2009.
19. M. Muhr, R. Kern, and M. Granitzer. Analysis of structural relationships for hierarchical cluster labeling. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 178–185. ACM, 2010.
20. Hausi A. Müller, Kenny Wong, and Scott R. Tilley. Understanding software systems using reverse engineering technology. pages 41–48, 1994.
21. Object Management Group. Business process model and notation (bpmn) specification, version 2.0.
22. C. Okasaki. Functional pearl: Red-black trees in a functional setting. *J. functional programming*, 9(4):471–477, 1999.
23. M. Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.
24. B. van Dongen, R. Dijkman, and J. Mendling. Measuring similarity between business process models. In *Advanced Information Systems Engineering*, pages 450–464. Springer, 2008.
25. I. Vanderfeesten, H. Reijers, J. Mendling, W. van der Aalst, and J. Cardoso. On a quest for good process models: the cross-connectivity metric. In *Advanced Information Systems Engineering*, pages 480–494. Springer, 2008.

26. M. Weske. *Business process management: concepts, languages, architectures.* Springer Publishing Company, Incorporated, 2010.
27. R. Xu, D. Wunsch, et al. Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678, 2005.

Implemented Process Model Metrics

Metric Name	Description
Number of Nodes	The total number of events, activities, functions, and gateways.
Number of Edges	The total number of edges being part of the control flow.
Number of Events	The total number of start events, internal events, and end events.
Number of Start Events	The total number of events without an incoming control flow edge.
Number of Internal Events	The total number of events with at least one incoming and one outgoing control flow edge.
Number of End Events	The total number of events without an outgoing control flow edge.
Number of Tasks	The total number of functions and activities.
Number of AND-Splits	The total number of AND-Gateways with multiple outgoing control flow edges and BPMN activities with multiple outgoing, unconditional control flow edges.
Number of AND-Joins	The total number of AND-gateways with multiple incoming control flow edges.
Number of XOR-Splits	The total number of XOR-gateways with multiple outgoing control flow edges.
Number of XOR-Joins	The total number of XOR-gateways and BPMN activities with multiple incoming control flow edges.
Number of OR-Splits	The total number of OR-gateways with multiple outgoing control flow edges and BPMN activities with multiple outgoing, conditional control flow edges.

Number of OR-Joins	The total number of OR-gateways with multiple incoming control flow edges.
Number of Alternative-Splits	The total number of gateways with multiple outgoing control flow edges, that have whether OR, XOR, nor AND symantic.
Number of Alternative-Joins	The total number of gateways with multiple incoming control flow edges, that have whether OR, XOR, nor AND symantic.
Number of Gateways	The total number of splits and joins.
Avg. Connector Degree	Average number of control flow edges being connected with a gateway.
Coefficient of Connectivity	Total number of control flow edges divided by the total number of nodes.
Coefficient of Network Complexity	Square of the total number of control flow edges divided by the total number of nodes.
Control Flow Complexity	Sum of all split gateways weighted by their number of possible states after the split.
Cross-Connectivity Metric	Average strength of connection between all pairs of process nodes.
Cycling	Number of nodes on a loop divided by the total number of nodes.
Cyclomatic Number	Total number of paths through a process model where loops are visited not more than once.
Density	Total number of control flow edges divided by the maximum number of control flow edges needed to connect the same number of nodes.
Density Related To Gateways	The density is calculated from the ratio of the number of control flow edges to the number of events, activities and the weighted number of gateways.
Depth	The maximum nesting level.

Diameter	The length of the longest path (number of visited control flow edges) without visiting a node twice. If a start node or end node is present, only paths from start to end are considered.
Max. Connector Degree	The maximum number of control flow edges being connected with the same gateway.
Separability	Number of nodes whose deletion divides a process model into at least two separate parts, divided by the total number of nodes minus the number of nodes without incoming or outgoing control flow edges.

Table 3: Description of implemented process model metrics.