KDDML: a middleware language and system for knowledge discovery in databases

Andrea Romei, Salvatore Ruggieri, Franco Turini

Dipartimento di Informatica, Università di Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa ITALY

Abstract. KDDML (KDD Markup Language) is a middleware language and system designed to support the development of final applications or higher level systems which deploy a mixture of data access, data preprocessing, extraction and deployment of data mining models. A KDDML language query is an XML-document where XML tags corresponds to operations on data/models, XML attributes correspond to parameters of those operations and XML sub-elements define arguments passed to the operators. The core of the KDDML system is a KDDML language interpreter with modularity and extensibility requirements as the main goals.

1 Introduction

The KDD (Knowledge Discovery in Databases) process, i.e. the process of finding "nuggets" of knowledge in data, is a complex task, heavily dependent on the problem and on the data at hand. It may consists of several repeated phases including business problem understanding, data understanding, data preparation, modelling (or data mining), evaluation and deployment. The development of KDD solutions requires then to specify the tasks at each phase and the interactions/dependencies among them. While KDD technology has reached a maturity state as far as the design of efficient knowledge extraction algorithms is concerned, the design of final applications is still an "art", aimed at smoothly composing algorithm libraries, proprietary API's, SQL queries and stored procedure calls to RDBMS, and *much much* code.

There is a fervent activity of standardization in the area of mining model representation and access, and of mining algorithms API's [2–4, 9]. In our view, a middleware language and system is needed to support the development of final applications or higher level systems which need a mixture of data access, data preprocessing, mining extraction and deployment. In this context, XML appears as a bridge between database technology and data mining tools. However, its use in existing tools appears to be limited to the exchange of mining models between applications. We would like to go further and conceive a language (and system) where XML is used for model and data representation and exchange, as well as for defining data and model processing tasks.

In this paper, we present our three-year experience in the design and development of KDDML, an XML-based middleware language and system in support of the KDD process.

The syntax of the KDDML language is XML-based, so to favor machine-processability, with each tag modelling an operator of the language. However, the semantics is purely "functional", which ensures compositionality of operators. The semantics of a KDDML language expression is either a model or a data table. Therefore, we call a KDDML language expression a *KDDML query*, in order to emphasize that a result is expected. We will present operators on data access and preprocessing, model extraction and deployment, and control flow operators. Concerning data and model representation, an XML-based approach is adopted here as well. In particular, models are represented using an extension of the PMML (Predictive Markup Modelling Language standard) [9].

Also, we outline the general architecture of the KDDML system, structured in layers for data/model repository, operator implementations, query interpretation, graphical user interface. Modularity and extensibility are a must in the design of the system.

2 KDDML: KDD Markup Language

The KDDML language assumes a *data repository*, containing relational tables, a *model repository*, containing mining models, and a *query repository*, containing queries. Tables, models and queries can be referenced by an identifier. KDDML queries are XML-documents, where XML tags correspond to operations on data and/or models, XML attributes corresponds to parameters of those operations and XML sub-elements define arguments passed to the operators. As an example, the query below specifies the construction and the application of a decision tree.

```
<KDD_QUERY name="sample">
   <TREE_CLASSIFY xml_dest="results.xml">
      <TREE_MINER xml_dest="tree.xml" target_attribute="class">
         <TABLE_LOADER xml_source="trainingSet.xml"/>
         <ALGORITHM algorithm_name="YADT">
            <PARAM name="confidence_for_pruning" value="0.4"/>
            <PARAM name="num_instances_for_leaf" value="3"/>
         </ALGORITHM>
      </TREE_MINER>
      <TABLE_LOADER xml_source="testSet.xml"/>
   <TREE_CLASSIFY>
</KDD_QUERY>
```

The root tag is `<KDD_QUERY>`, with the query identifier as an attribute.

`<TREE_CLASSIFY>` is the operator that applies a decision tree to predict the class of tuples in a test set. The attribute `xml_dest="results.xml"` states that the results of the classification are stored in the data repository for further processing or analysis. The decision tree to be applied is provided by the first sub-element (with tag `<TREE_MINER>`) which specifies the construction of a decision tree. The test set is provided by the second element (with tag `<TABLE_LOADER>`), which specifies a table in the data repository. In turn, the construction of a decision tree (tag `<TREE_MINER>`) takes place on a training set `trainingSet.xml` from the data repository by applying a decision tree induction algorithm (here, YADT from [8]) with parameters concerning the pruning strategy of the algorithm. The name of the class attribute is provided as attribute of the `<TREE_MINER>` element.

As one could expect, arguments to an operator must be of an appropriate type and sequence, i.e. an operator *signature* must be specified. We denote the signature of an operator $f : t_1 \times \ldots \times t_n \to t$ returning type t by defining a DTD for KDDML queries that constraints sub-elements to be of type $t_1, \ldots, t_n$. Thus, KDDML queries corresponds to terms in the algebra of operators, though syntactically represented as XML documents.

The set of types of KDDML operators includes: `table`, `tree`, `rda`, `sequences`, `clusters`, `hierarchy`, `scalar`, `algs` and `xml`. Intuitively, there is one type for data sources, one type for each mining model (`tree`, `rda`, `sequences`, `clusters`), one type for hierachies, one type for algorithms (`algs`) and one for operators that return a scalar (i.e., a number or a string). Finally, the `xml` type denotes arguments that are generic XML elements to be evaluated directly by the operator.

Under this interpretation, the semantics of a KDDML query amounts to a strict functional execution of the corresponding term. The evaluation of an XML-fragment:

```
<OPERATOR_NAME xml_dest="results.xml" att1="v1" ... attM="vM">
   <ARG1_NAME> .... </ARG1_NAME>
   ...
   <ARGn_NAME> .... </ARGn_NAME>
</OPERATOR_NAME>
```

consists of:

1. recursive evaluation of fragments from `<ARG1_NAME> ... </ARG1_NAME>` to `<ARGn_NAME> .... </ARGn_NAME>`; in case the ith argument of `<OPERATOR-_NAME>` is expected of type `xml`, the element `<ARGi_NAME> ... </ARGi_NAME>` is itself the result of its evaluation;
2. evaluation of attributes `att1 ... attM` returning a set of scalar values;
3. a call to an operator $f_{\texttt{OPERATOR\_NAME}}$, accepting results from (1) and (2) and yielding the final result of the fragment.

Moreover, a copy of the final result (which may be an intermediate result of a possibly larger query) is stored in the (model or data) repository if the attribute `xml_dest` is specified. Notice that repositories are persistent, so to favor the reuse of extracted knowledge and preprocessed data.

As a by-product, the language satisfies a *closure principle*, namely that any operator returning type t can be used wherever an argument of type t is required. Also, validation of queries as XML documents against the DTD corresponds to static type-checking of operators in the query.

2.1 Data access and preprocessing

Data format The KDDML language assumes a *data repository*, where tables are represented as an XML file containing a *schema* and a reference to the actual data, which is stored in CSV (Comma Separated Values) format. Here it is a fragment of a XML document `census.xml` describing a data set of people with their age, education and purchase data (day of week, product brand and amount).

```
<KDDML_TABLE data_file="census.csv">
   <SCHEMA logical_name="census" number_of_attributes="6"
           number_of_instances="16">
      <ATTRIBUTE name="age" number_of_missed_values="0"
                 type="numeric">
         <NUMERIC_DESCRIPTION mean="40.75" variance="237.8"
                              min="18.0" max="70.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="education" number_of_missed_values="3"
                 type="nominal">
         <NOMINAL_DESCRIPTION number_of_values="4">
            <VALUE value="doctorate" cardinality="1"/>
            <VALUE value="bachelors" cardinality="7"/>
            <VALUE value="HS-grad" cardinality="3"/>
            <VALUE value="masters" cardinality="2"/>
         </NOMINAL_DESCRIPTION>
      </ATTRIBUTE>

      ...

   </SCHEMA>
</KDDML_TABLE>
```

The `data_file` attribute of the `<KDDML_TABLE>` tag refers the location of physical data. The XML document specifies metadata information, including attribute type (nominal, numeric or string) and some simple statistics on attribute values.

Data access The data repository is populated by KDDML queries that yield tables as output. As one could expect, data access operators are basically available to access RDBMS (using SQL SELECT queries) and text files in the ARFF format adopted by the Weka [10] suite.

```
<ARFF_LOADER xml_dest="BasketData.xml"
             arff_file_path="D:/Repository/BasketData.arff"/>

<DATABASE_LOADER xml_dest="BasketData.xml"
                 database_name="jdbc::odbc::basketDB"
                 sql_query="SELECT * FROM BasketData"/>

<TABLE_LOADER  xml_source="BasketData.xml"/>
```

In the first (resp. second) query fragment, an ARFF table (resp. database table) is accessed, transformed into the internal representation, saved into the repository (recall that, however, the `xml_dest` attribute is optional), and finally passed up to the father node of the query fragment. Mapping from ARFF (resp., SQL) data types to the logical data types of the XML representation is automatic. However, preprocessing operators allows for specifying different logical types of attributes of loaded tables.

Export of tables to database and ARFF formats is achieved by the `<DATABASE_WRITER>` and `<ARFF_WRITER>` operators.

```
<DATABASE_WRITER database_name="jdbc:odbc:basketDB",
                 table_name="BasketData">
   <TABLE_LOADER xml_source="BasketData.xml"/>
</DATABASE_WRITER>
```

Data preprocessing Data preprocessing [7] is a time-consuming phase of the KDD process, including tasks such as data selection, filtering, merging, cleaning, discretization, sorting, aggregating and many others. KDDML offers some operators for data preprocessing, yet they are the latest addition to the language and system. Their typing is quite intuitive, typically requiring a table as an argument. As a simple example, the following fragment removes the `age` attribute from the input table. Its typing is $f_{\texttt{PP\_FILTER\_ATTRIBUTES}} : \texttt{table} \rightarrow \texttt{table}$.

```
<PP_FILTER_ATTRIBUTES xml_dest="census_removed.xml"
                      attributes_list="age"
                      take_or_remove="remove">
  <TABLE_LOADER xml_source="census.xml"/>
</PP_FILTER_ATTRIBUTES>
```

On the contrary, new attributes can be added by the operator `PP_NEW_ATTRIBUTE` In this case, the attribute is derived from existing ones by means of a simple expression language. Also, the type of the derived attribute can be set. Its typing is $f_{\texttt{PP\_NEW\_ATTRIBUTE}} : \texttt{table} \times \texttt{xml} \rightarrow \texttt{table}$. The second argument is an `EXPRESSION` tag, which is directly interpreted by the operator in order to compute calculated values.

```
<PP_NEW_ATTRIBUTE attribute_name="born_year"
                  attribute_type="numeric"
                  position="1">
   <TABLE_LOADER xml_source="census.xml"/>
   <EXPRESSION>
      <SEQ_TERM op_type="subtract">
         <BASE_TERM value="2004"/>
         <BASE_TERM value="@age"/>
      </SEQ_TERM>
   </EXPRESSION>
</PP_NEW_ATTRIBUTE>
```

In this example, a new numeric attribute `born_year` is added in the first position of the `census.xml` dataset. The values of the new attribute are calculated as the difference between the current year and the year of birth. With the special symbol "@" in the second term of the expression, we denote the input table attribute `age`.

Sampling is a largely used task: for an input table, a subset of rows is selected accordingly to a sampling method. Therefore, the typing of a sampling operator is $f_{\texttt{PP\_SAMPLING}} : \texttt{table} \times \texttt{algs} \rightarrow \texttt{table}$. Below is an example query selecting 66% of input rows without replacement sampling policy.

```
<PP_SAMPLING xml_dest= "sampling.xml">
    <TABLE_LOADER xml_source= "census.xml"/>
    <ALGORITHM algorithm_name="simple_sampling">
        <PARAM name="percentage" value="0.66"/>
        <PARAM name="with_replacement" value="false"/>
    </ALGORITHM>
</PP_SAMPLING>
```

There are a few other preprocessing operators in KDDML, including: `PP_NUMERIC_DISCRETIZATION` to discretize numeric attributes, `PP_RENAME_ATTRIBUTES` to rename attributes, `PP_CHANGE_TYPE` to change the logical type of attributes, `PP_REMOVE_ROWS` to delete rows under specified conditions, `PP_REWRITING` to apply pattern matching rewriting of attribute values, `PP_SORTING_ATTRIBUTE` to sort rows according to the values of an attribute or according to their frequencies.

While the list of KDDML preprocessing operators is not comparable to the huge number of preprocessing tasks described in the literature, we point out that adding a new operator in KDDML is not much of a problem, since it is enough to specify the operator signature and the DTD of the operator tags. The overall (functional & compositional) semantics of KDDML allows for a smooth integration of the new operator in the language.

2.2 Mining models

Model format As for data, the KDDML language assumes a *model repository*, containing extracted data mining models, which can be referenced by an identifier (in a different namespace from data).

KDDML represents models as an extension of PMML documents. Since PMML in its present version is not sufficient to capture all details of mining models, we deploy the PMML extension mechanism in two cases both regarding classification models. In the first one, the notion of confusion matrix is added to decision tree models. In the second one, we allow for classification models that exploit predictions of two or more decision trees, e.g. a voting classifier. A classic example concerns meta-classifiers, which are intended to overcome the bias due to the random selection of the training set or due to the choice of specific algorithms and parameters.

Model access Direct access to models in the *model repository* is achieved by the `TREE_LOADER`, `SEQUENCE_LOADER`, `RDA_LOADER`, `CLUSTER_LOADER`, `HIERARCHY_LOADER` operators. As the name suggests, the forms of knowledge currently addressed include decision trees, sequential patterns, association rules (`RDA`), clustering and item hierarchies. Also, PMML compliant models provided from external tools can be accessed and imported in the repository.

```
<TREE_LOADER  xml_source="DecisionTree.xml"/>

<PMML_RDA_LOADER  xml_dest="ExternRdA.xml"
                  pmml_source="ftp://www.foo.edu/models/RdA.xml"/>
```

Model extraction Mining models are extracted from a data source using a data mining algorithm. In the next example, the top 20 association rules are extracted from market basket data with minimum support of 40% and confidence of 60%.

```
<RDA_MINER xml_dest="MineBasket.xml">
   <ARFF_LOADER arff_file_path="D:/Repository/BasketData.arff"/>
   <ALGORITHM algorithm_name="DCI">
      <PARAM name="min_support" value="0.4"/>
      <PARAM name="min_confidence" value="0.6"/>
      <PARAM name="max_number_of_rules" value="20"/>
   </ALGORITHM>
</RDA_MINER>
```

The `<RDA_MINER>` tag expects a sub-element with input data and a second sub-element with the algorithm name and parameters (name and value).

Input data can be in the *transactional format*, i.e. with an attribute `transaction` and an attribute `event`. This format allows for deriving intra-attribute association rules, such as `spaghetti AND tomato juice` → `parmesan`.

Also, the *relational format* is recognized, i.e. with an attribute for every item (or item category). This format allows for deriving inter-attribute association rules, such as `carType=racing AND homeInsurance=false` → `married=false`.

The algorithm used here is DCI (Direct Count & Intersect) from [5]. DCI is an efficient procedure that takes into account density or sparsity of input transactions.

Analogously, tags `<TREE_MINER>`, `<SEQUENCE_MINER>` and `<CLUSTER_MINER>` exists for extracting decision trees, sequence patterns and clusters. The algorithms used are respectively the already mentioned YADT [8] for decision tree induction, a main-memory implementation of the PrefixSpan [6] for sequential patterns, and the EM and KMeans clustering algorithms from the Weka library [10]. In the next example, three clusters are extracted from `census.xml` using the EM algorithm.

```
<CLUSTER_MINER xml_dest="MineClusters.xml">
   <TABLE_LOADER xml_source="census.xml"/>
   <ALGORITHM algorithm_name="EM">
      <PARAM name="number_of_clusters" value="3"/>
   </ALGORITHM>
</CLUSTER_MINER>
```

Model application and evaluation Extracted models can be applied on (new) data to predict features or to select data accordingly to the knowledge stored in the model.

We have seen earlier how a decision tree extracted from a training set can be applied to predict the class of tuples in a test set. More in detail, `<TREE_CLASSIFY>` yields a `table` with an additional column (whose name is the one of the class column followed by *_predicted*) consisting of the class predicted by the decision tree. The procedure used to determine the class predicted is the one adopted in the C4.5 algorithm [8]. In addition to `<TREE_CLASSIFY>`, operators for model application include:

– `<MISCLASSIFIED>`: given a table returned by the `<TREE_CLASSIFY>` operator, selects the rows where the predicted class value differs from the actual class one;
– `<RULE_SATISFY>` (resp., `<RULE_EXCEPTION>`): given a set of association rules and a table, extracts those transactions in the table that satisfy (resp., are exceptions to) one or more of the association rules; `<SEQUENCE_SATISFY>` and `<SEQUENCE_EXCEPTION>` are the equivalent operators for sequence models;
– `CLUSTER_NUMBER`: given a cluster model and a dataset, this operator returns the tuples of the dataset belonging to a specified cluster number (`max` can also be specified to get the tuple in the cluster of maximal cardinality);
– `CLUSTER_CENTROID`: given a cluster model, it returns tuples describing the cluster centroids.

Model (meta-)reasoning Models extracted by data mining algorithms very often need to be further processed, e.g., combined with other models. The example below returns a voting classifier among three decision trees: `tree1.xml` and `tree2.xml` are already present in the model repository, and `tree3.xml` is mined from `trainingSet.xml`.

```
<TREE_COMMITTEE xml_dest="treeCommittee.xml">
   <TREE_LOADER xml_source="tree1.xml"/>
   <TREE_LOADER xml_source="tree2.xml"/>
   <TREE_MINER xml_dest="tree3.xml" target_attribute="class_name">
      <TABLE_LOADER xml_source="trainingSet.xml"/>
      <ALGORITHM algorithm_name="YADT">
         <PARAM name="confidence_for_pruning" value="0.4"/>
         <PARAM name="num_instances_for_leaf" value="3"/>
      </ALGORITHM>
   </TREE_MINER>
</TREE_COMMITTEE>
```

The operator performs a run-time checking that the three classifiers share the same meta-data. If this is not the case, the evaluation of the query terminates with a run-time error.

Other operators on model (meta-)reasoning available in the system include filtering of association rules and sequential patterns, and the selection of rules that are preserved over a hierarchy of items.

2.3 Control flow and external programs

In this section, we describe the operators that allow for better control of flows of data and models in queries, and for calling external programs.

Calls to external programs / RDBMS Specialized procedures can sometimes be useful to preprocess or analyse data. The `<EXT_CALL>` operator allows for calling external programs, including e.g., calls to RDBMS stored procedures.

```
<EXT_CALL path="/usr/bin/mysql">
   <PARAM value="localhost">
   <PARAM value="UPDATE mytable SET cost = cost * 1.10"/>
</EXT_CALL>
```

The `<PARAM>` operator returns a scalar (the one of the `value` attribute), which is then used as a command line argument of the called program. `<EXT_CALL>` also returns a scalar, e.g., the number of updated rows in the example above.

Calls of queries In order to modularize (long) queries, an operator that retrieves and evaluates queries in the query repository is provided. Queries admit parameters, whose list is specified at the `<KDD_QUERY>` tag. Actual parameters are substituted to formal parameters at the time the query is loaded from the repository.

```
<KDD_QUERY name="generic_tree" par_list="perc,source,dest">
    <TREE_MINER xml_dest="#dest#" target_attribute="class_name">
      <PP_SAMPLING xml_dest= "sampling.xml">
          <TABLE_LOADER xml_source="#source#"/>
          <ALGORITHM algorithm_name="simple_sampling">
              <PARAM name="percentage" value="#perc#"/>
              <PARAM name="with_replacement" value="false"/>
          </ALGORITHM>
      </PP_SAMPLING>
      <ALGORITHM algorithm_name="YADT">
        <PARAM name="num_instances_for_leaf" value="3"/>
      </ALGORITHM>
    </TREE_MINER>
</KDD_QUERY>
```

The above query builds a decision tree on a subset of a data source and saves the tree in a specified destination. Notice that the syntax for using a formal parameter requires to write it between # signs. The sample query can be called from within other queries as follows:

```
<CALL_QUERY name="generic_tree">
   <PARAM name="perc" value="0.6"/>
   <PARAM name="source" value="training.xml"/>
   <PARAM name="dest" value="tree.xml"/>
</CALL_QUERY>
```

The operator <CALL_QUERY> returns the same result of the called query. Since the type may not be known at compile time (e.g., when the query name itself is provided by a parameter), the type of the result is checked at run-time.

Sequences and parallelism of queries It is sometimes useful to evaluate queries in strict sequence or to mark potential parallelism. As an example, consider building two distinct models on a same training set. The preprocessing of the training set is preliminary to the (independent, hence potentially parallel) evaluation of the tree building queries.

```
<KDD_QUERY>
  <SEQ_QUERY>
    <EXT_CALL path="mypreprocessing">
      <PARAM value="inputdata.arff"/>
      <PARAM value="training.arff"/>
    </EXT_CALL>
    <PAR_QUERY>
       <TREE_MINER xml_dest="tree1.xml>
          ...
       </TREE_MINER>
       <TREE_MINER xml_dest="tree2.xml>
          ...
       </TREE_MINER>
    <PAR_QUERY>
  </SEQ_QUERY>
</KDD_QUERY>
```

The <SEQ_QUERY> operator (resp., <PAR_QUERY>) models sequentialization (resp., potential parallelism). The returned value of both operators is assumed to be the one of the last operator in the sequence of their arguments.

3 KDDML: system architecture

KDDML is implemented in Java, in order to be portable. The overall architecture of KDDML is structured in layers, as reported in Fig. 1. Each layer implements a specific functionality and supplies an interface to the layer above:

– The *repository layer* manages the read/write access to data and models repositories and the read access to data and models from external sources providing programmatic functionality to the higher layers.
– The *operators and algorithms layer* includes the implementations of language operators
– The *interpreter layer* accepts a validated KDDML query, evaluates it, save the final result in the repository and returns it to caller.
– The *user interface layer* is a GUI for user friendly input of queries and for browsing of extracted knowledge. Strictly speaking, the GUI is not part of the core of the system. In fact, KDDML queries can be generated by other

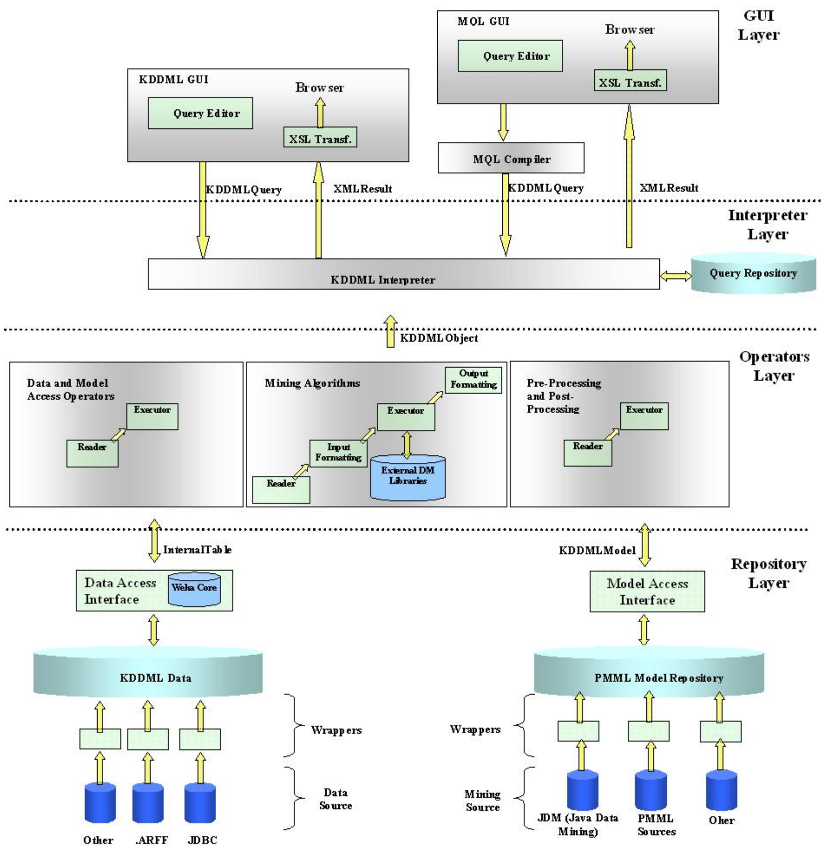

Fig. 1. KDDML system architecture.

programs, such as a vertical applications that need performing some KDD steps. The result of the invoked interpreter is returned as a DOM object or an XML document, which can be further processed with standard tools. The picture shows that queries can also be expressed in a logic and algebraic query language MQL [1] and then compiled into KDDML.

The design of the KDDML system took into special account the requirements of extensibility of the KDDML language, which can be distinguished in:

– **data sources extensibility:** adding a new data source type in the KDDML language consists of simply adding a new tag (such as `<NEW_DATA_SOURCE_-LOADER>`) with appropriate attributes and sub-elements specifying how to locate the table. As a consequence, the system should allow for transparently adding a wrapper to/from the new data source type, which encapsulates the details of transforming data and meta-data into the ones the internal table representation;

– and **algorithms extensibility:** adding a new preprocessing, tree induction, clustering, association rules or sequential pattern mining algorithm should be as simple as possible. As for data sources, the idea is that the new algorithm should be *pluggable in* the language and system. Notice that, as far as the language part is concerned, this is not much of a problem, since the algorithm name and parameters are not part of the language syntax.

4 Conclusions

KDDML is a middleware language and system for KDD that supports the development of higher level applications and systems. The language is XML-based, with a functional semantics of language operators, which are represented as XML tags. The system has been designed to be easily extensible with new data formats/sources, model format/sources, and preprocessing/mining algorithms. KDDML is distributed under the GNU GPL licence at the web site: `http://kdd.di.unipi.it/kddml`.

Acknowledgments. This work has been partially supported by Italian national FIRB project no. RBNE01KNFP *GRID.it* and by Italian national strategic project *legge 449/97* No. 02.00640.ST97.

References

1. M. Baglioni and F. Turini. MQL: An Algebraic Query Language for Knowledge Discovery. In A. Cappelli and F. Turini, editors, *Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence*, volume 2829 of *Lecture Notes in Computer Science*, pages 225–236. Springer-Verlag, 2003.
2. International Organization for Standardization (ISO). Information Technology – Database Language – SQL Multimedia and Application Packages – Part 6: Data Mining, 2003. Draft Standard No. ISO/IEC 13249-6:2003.
3. JSR-73 Expert Group. Java Data Mining API, 2004. Java Specification Request No. 73, `http://www.jcp.org/en/jsr/detail?id=73`.
4. Object Management Group (OMG). Common Warehouse Meta-model (CWM), 2002. Version 1.1, `http://www.omg.org/cwm`.
5. S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resources-aware mining of frequent sets. In *IEEE ICDM Int. Conf. on Data Mining*. IEEE Computer Society, 2002. `http://hpc.isti.cnr.it/~palmeri/datam/DCI`.
6. J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Trans. on Knowledge and Data Eng.*, 16(11):1424–1440, 2004.
7. D. Pyle. *Data Preparation for Data Mining.* Morgan Kaufmann Publishers, San Francisco, 1999.
8. S. Ruggieri. YaDT: Yet another Decision Tree builder. In *Proc. of the 16th Int. Conf. on Tools with Artif. Intellig.*, pages 260–265. IEEE Computer Society, 2004.
9. The Data Mining Group. Predictive Model Markup Language (PMML). Version 2.1, 2003. `http://www.dmg.org`.
10. I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan & Kaufmann, 2000. Version 3.4.3 from `http://www.cs.waikato.ac.nz/ml/weka`.