BeMoRe: a Repository for Handling Models Behaviors

Youness Bazhar
LIAS/ISAE-ENSMA
Futuroscope, FRANCE
Email: bazhary@ensma.fr

Yamine Aït-Ameur
IRIT/INP-ENSEEIHT
Toulouse, FRANCE
Email: yamine@enseeiht.fr

Stéphane Jean
LIAS/University of Poitiers
Futuroscope, FRANCE
Email: jean@ensma.fr

***Abstract*—With the increasing size of models and their instances, the management of models in databases becomes a necessity. Persistent Model Management Systems (PMMS) aim at providing a persistent environment for the management of instances, models and metamodels. They consist of (1) a database that stores metamodels, models and their instances, and (2) an associated exploitation language for manipulating these different abstraction layers. Several PMMS have been proposed in the literature but they currently mostly focus on the structural definition of models and metamodels in terms of (meta-)classes and (meta-)attributes. The behavioral semantics that consists of associating operations to models and metamodels elements is currently mostly not supported or only partially supported (by a set of predefined hard coded operations or by imposing a single programming language). In this paper, we propose an extension of PMMS to support the definition of behavioral semantics of models and metamodels using a wide range of programming possibilities. Our approach consists of introducing dynamically user-defined operations that can have multiple and heterogeneous implementations (e.g., external programs or web services). As a consequence, this extension enhances PMMS giving them more coverage and further flexibility. Our proposal has been implemented in a PMMS called *BeMoRe* and several experiments have been run to analyze the scalability of this PMMS.**

***Keywords*—model management; meta-modeling; database**

I. Introduction

Models are widely used in software engineering to design software components such as database schemes or user interfaces. This involves operations on models such as code generation, transformation, archiving, versioning, etc. Following the vision of Bernstein [1], several *model management systems* (*MMS*) have been set up during the last decade to manage instances, models and metamodels and support operations on them (e.g., [2], [3], [4], [5]).

With the increasing size of data instances and models in several domains (e.g., in genomics, the Uniprot dataset, www.uniprot.org, gathers more than 200GB of protein sequence resources), the possibility of managing large scale models and instances in MMS has raised a lot of interest. Two main approaches have been followed to increase the scalability of MMS. The first approach consists of connecting a MMS to a database called *model repository* (e.g., EMFStore [6], TERESA model repository [7]). This approach uses a loose coupling between the MMS and the database and has two main drawbacks: (1) most model management operations require loading the whole model and instances in main memory and (2) the database exploitation language does not support the definition, manipulation and querying of models and metamodels (it only supports basic SQL operations). To address these problems, a second approach, followed in our work, has been developed. It consists of extending databases for the management of models and metamodels (e.g., [2]). These systems called *Persistent Model Management Systems* (*PMMS*) are composed of (1) a database that stores metamodels, models and instances and (2) an exploitation language for manipulating models and metamodels.

If PMMS solve the two drawbacks of the loose coupling of a MMS with a database, they currently do not support the same flexibility concerning the definition of behavioral semantics (procedural aspects) of models and metamodels. Indeed PMMS focus mainly on the definition of the structure of metamodels and models but provide a limited support for the definition of operations on models and metamodels. For example, some PMMS provide hard-encoded operators for model management (e.g., Match, Merge, Union [8], [1]), or only give access to the database procedural languages (e.g., PL/SQL) that do not support the manipulation of models and metamodels (they only manipulate relational tables). The most advanced PMMS concerning the definition of metamodels and models behaviors is ConceptBase [2]. Using this PMMS user-defined operations can be defined on models and metamodels as deductive rules implemented with a specific language (PROLOG). However this PMMS lacks the possibility to integrate operations that have already been implemented using a given programming language or provided as an external web service.

In this paper we propose an extension of PMMS to support the definition of behavioral semantics of models and metamodels using a wide range of programming capabilities. This extension has been motivated in a previous paper [9] by presenting a complete state of the art, and applied in the specific context of ontology-based databases in [10]. In this paper we make the following new contributions:

- definition of a set of requirements for a complete PMMS;
- definition of a PMMS including the behavioral aspect;
- implementation of our approach: the *BeMoRe* PMMS;
- first experiments to study the scalability of BeMore.

The remainder of this paper is organized as follows. Section II presents a set of requirements for a complete PMMS justified on a motivating example. Section III gives an overview of the state of the art by analyzing existing PMMS using our requirements. Section IV presents the formal definition of a PMMS handling behavioral semantics of models and metamodels. Section V overviews the implementation of

our approach and Section VI shows the experiments done to study its scalability. Finally, section VII concludes this paper and discusses ongoing works.

II. Requirements for a Complete PMMS

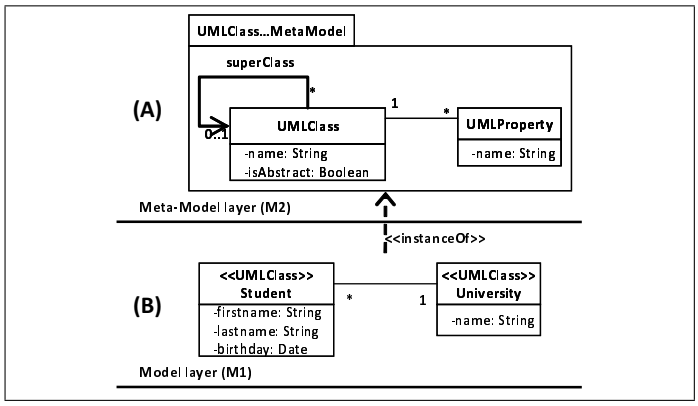

Fig. 1: A motivating example

Figure 1 presents a simple class diagram metamodel (A) and a model (B) conforming to that metamodel. Using this example, we are capable to define the set of requirements identified for a complete PMMS. Due to space limitation we use a very simple example. The interested reader may refer to [9] and [10] for more complex and real motivating examples.

Requirement 1 (*extensible metamodel layer*)

> *PMMS shall offer an extensible metamodel layer so that multiple modeling formalisms can be defined.*

Justification: in our example, we only have the UML class diagram metamodel defined at the metamodel layer. However sofware engineering uses a lot of different models (e.g., entity-relationship, functional, state transition models).

Requirement 2 (*structural and descriptive semantics*)

> *PMMS shall support the definition of structural and descriptive semantics of metamodels and models elements. For instance, the PMMS shall provide constructors of classes, attributes, inheritance and association relationships for defining models and metamodels.*

Justification: following the MOF specification, most models and metamodels (such as the ones of our example) can be expressed with object-oriented constructors.

Requirement 3 (*behavioral semantics*)

> *PMMS shall support introducing operations (functions, procedures) on metamodels and models elements.*

Justification: operations on models and metamodels elements are important to accomplish advanced model management tasks such as model transformation, code generation or constraints checking. For instance, in our example an operation could be defined to export the UML models in XML, or to compute the age of a student.

Requirement 4 (*flexible programming environment*)

> *PMMS shall provide an heterogeneous programming environment to implement operations. Particularly, it shall be able to use external programs written in any language (e.g., Java, C++) and remote services.*

Justification: as it is better to reuse existing pieces of software instead of rewriting them, a PMMS should be able to integrate existing implementations of operations whatever is the programming language used. For example, it is easy to find an existing code that exports an UML model in XML. So a PMMS should allow users to reuse this piece of software to implement an operation that exports UML models.

Requirement 5 (*hot-plug of implementations*)

> *PMMS shall support an immediate usage of the implementations for an operation without restarting the system (warm start).*

Justification: restarting a database system must be avoided for high availability applications. Thus the definition of an implementation of an operation, even if it is a web service, should not require restarting the PMMS (warm start).

III. Related Work

Several PMMS have been proposed. This section analyzes the most relevant PMMS according to our requirements.

ConceptBase [2] is a PMMS based on an object-oriented and deductive database. It is based on the Telos language that supports the definition of multiple abstraction layers with a set of constraints, rules and queries using meta-formulas. Furthermore, ConceptBase provides a set of predefined operators to manipulate simple and complex data types, and gives the possibility to introduce user-defined functions with membership constraints and external implementations. Yet, these implementation can only be done in the Prolog language. Besides external programs have to be stored in a special and internal file system, and requires restarting the server (cold start) in order to support the function newly introduced [11].

GeRoMe [3] is an extension of ConceptBase to define new operators from other ones by combining existing operators. However, this extension does not introduce a more flexible programming environment for these operations.

Rondo [4] is a PMMS that has a fixed and non extensible metamodel layer. It provides conceptual structures to define models and specify the behavioral semantics by providing a set of *primitive* high-level operators for model management and model mappings such as *Match*, *Delete* or *Extract*. Moreover, Rondo supports the definition of derived operators by composing basic and other defined operators.

Clio [5] is a PMMS defined for facilitating the tasks of heterogeneous data transformation and integration. These tasks are facilitated by mapping a source schema to a target schema with SQL statements.

DB-MAIN [12] is a PMMS designed for the management of database evolution. It is based on a fixed hard-encoded metamodel and offers a set of built-in high-level operators

for modifying the database structure and contents when an evolution is required.

OntoDB/OntoQL [13] is a PMMS initially defined for the management of ontologies and ontology models. It includes the OntoDB model repository and the OntoQL meta-modeling language. This PMMS is based on a fixed metametamodel to define and modify metamodels. Concerning the behavioral semantics, OntoDB/OntoQL uses only the PgPL/SQL procedural language of its back-end database management system (PostgreSQL). This language cannot manipulate complex types (e.g., meta-classes or classes) and consequently cannot define high-level operators.

As the previous overview of the state of the art shows, each existing PMMS presents some strengths and some limitations for the definition of structural and behavioral semantics of models and metamodels. The identified limitations are presented in Table I. Hence next section introduces the formal definition of an extension of PMMS to fulfill these requirements.

TABLE I: Synthesis of the state of the art

	Req. 1	Req. 2	Req. 3	Req. 4	Req. 5
ConceptBase	Yes	Yes	Yes	restricted	No
GeRoMe	Yes	Yes	Yes	restricted	No
Rondo	No	Yes	hard-coded	No	No
Clio	No	No	restricted	No	No
DB-MAIN	No	Yes	restricted	No	No
OntoDB/OntoQL	Yes	Yes	restricted	No	No

IV. Proposed Extension of PMMS

A PMMS is composed of a database data model and an exploitation language. The proposed extension of these two parts of a PMMS are presented in Subsection IV-A and Subsection IV-B.

A. Proposed Extension of PMMS Data Model

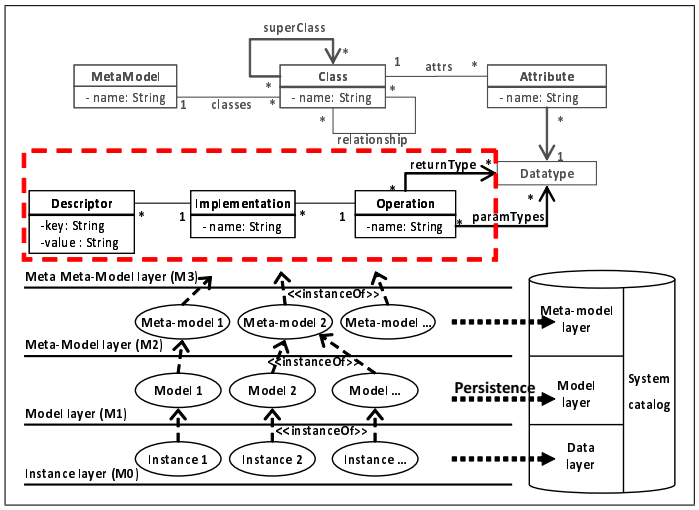

Fig. 2: The proposed data model for PMMS

Figure 2 gives an overview of the extended data model of PMMS that we propose. For conciseness we only detail the metametamodel layer but a similar extension has been be done at the metamodel level to be able to define operations at the different abstraction layers. Our model includes a set of classes that are described by attributes, and single class inheritance relationships are allowed. This part of our data model is usually available in all PMMS that, as we have seen in the previous section, focuses mainly on the structural and descriptive semantics of metamodels elements. The dashed area gives an overview of our proposed extension for the definition of behavioral semantics of metamodels elements. Our model supports the definition of operations with a list of input and an output. Furthermore, an operation can be associated to multiple implementations. Each implementation is itself described by a set of descriptors (couples of *key*, *value*). With these generic set of descriptors a new programming environment can be easily integrated in our approach.

This extension of PMMS can be formally defined by the following sets: MM, CL, ATT, DT, OP, IMP, $DESC$ that represent respectively sets of metamodels, classes, attributes, data types, operations, implementations and implementation descriptors. Table II gives the definition of these sets as well as a subset of constraints concerning these sets.

TABLE II: PMMS formal model

A metamodel is described by a set of classes:
$classes : MM \rightarrow P(CL)$
$\forall mm_i \in MM \Rightarrow \exists! E \in P(CL)/classes(mm_i) = E$
$\forall (mm_i, mm_j) \in MM/i \neq j \Rightarrow classes(mm_i) \cap classes(mm_j) = \emptyset$
A class may have a super class:
$superClass : CL \rightarrow CL$
A class may have inherited attributes from its super class:
$inheritedAttributes : CL \rightarrow P(ATT)$
$\forall cl_i \in CL \Rightarrow \exists! E \in P(ATT)/$
$\quad inheritedAttributes(cl_i) = attributes(superClass(cl_i))$
A class may be described by additional attributes:
$definedAttributes : CL \rightarrow P(ATT)$
$\forall cl_i \in CL \Rightarrow \exists! E \in P(ATT)/definedAttributes(cl_i) = E$
$\forall cl_i \in CL \Rightarrow$
$\quad definedAttributes(cl_i) \cap inheritedAttributes(cl_i) = \emptyset$
$\forall (cl_i, cl_j) \in CL/i \neq j \Rightarrow$
$\quad definedAttributes(cl_i) \cap definedAttributes(cl_j) = \emptyset$
The set of attributes of a class:
$\forall cl_i \in CL \Rightarrow attributes(cl_i) =$
$\quad inheritedAttributes(cl_i) \cup definedAttributes(cl_i)$
An attribute has a data type:
$typeOf : ATT \rightarrow DT$
$\forall att_i \in ATT \Rightarrow \exists! dt_j \in DT/typeOf(att_i) = dt_j$
An operation parameter has an order:
$input : OP$ x $\mathbb{N}^+ \rightarrow DT$
$\forall op_i \in OP \Rightarrow input(op_i) \in DT$
An operation can return a result:
$output : OP \rightarrow DT \cup \emptyset$
An operation can have several implementations:
$implementations : OP \rightarrow P(IMP)$
$\forall op_i \in OP \Rightarrow \exists! E \in P(IMP)/implementations(op_i) = E$
$\forall (op_i, op_j) \in OP/i \neq j \Rightarrow$
$\quad implementations(op_i) \cap implementations(op_j) = \emptyset$
An implementation is described by a set of descriptors:
$descriptors : IMP \rightarrow P(DESC)$
$\forall imp_i \in IMP \Rightarrow \exists! E \in P(DESC)/descriptors(imp_i) = E$
$\forall (imp_i, imp_j) \in IMP/i \neq j \Rightarrow$
$\quad descriptors(imp_i) \cap descriptors(imp_j) = \emptyset$

B. Proposed Extension of the PMMS Metamodeling Language

The exploitation language of a PMMS is composed of a model and a metamodel definition, manipulation and query language. Table III presents a subset of basic actions that a PMMS definition language should fulfill. These actions are defined by a signature (*SIG*), a precondition (*PRC*) and a

postcondition (*POC*). We only present in this table the *create* operations but the other operations (*alter*, *update* and *delete*) have been defined as well.

TABLE III: Main actions of the PMMS definition language

creation of a metamodel: SIG: $addMetaModel(mm) = MM'$ PRC: $mm \notin MM$ POC: $MM' = MM \cup \{mm\}$
creation of a class: SIG: $addClass : MM$ x $C \rightarrow MM$ PRC: $cl \notin CL$ POC: $classes(mm) = classes(mm) \cup \{cl\}$
creation of an attribute: SIG: $addAttribute : CL$ x $ATT \rightarrow CL$ PRC: $att \notin ATT$ POC: $attributes(cl) = attributes(cl) \cup \{att\}$
creation of an operation: SIG: $addOperation(op) = OP'$ PRC: $op \notin OP$ POC: $OP' = OP \cup \{op\}$
creation of an implementation: SIG: $addImplementation : OP$ x $IMP \rightarrow OP$ PRC: $imp \notin IMP$ POC: $implementations(op) = implementations(op) \cup \{imp\}$

Concerning querying, most PMMS have a query language whose algebra includes relational-like operators (e.g., projection or selection) for models and metamodels. These algebra should be extended to be able to execute operations that can be defined with our proposed extension. To fulfill this need, we define the *RUN* operator. We only give the signature of this operator in table IV since its semantics depends on the processing done in the corresponding operation.

TABLE IV: Formalization of the RUN operator

RUN : OP x INPUT $\rightarrow$ OUTPUT
INPUT is an expression of input values. $INPUT = (I_C \oplus I_{I_C} \oplus I_{DT})^+ \oplus \emptyset$ $instOf$ is a function that returns the set of instances of a concept. $I_C = instOf(c_1) \cup instOf(c_2) \cup ... \cup instOf(c_n)$ $I_{I_C} = instOf(instOf(c_1)) \cup ... \cup instOf(instOf(c_n))$ I_{DT} represents simple types values (string, boolean, integer, etc.).
OUPUT is the output value. $OUTPUT = I_C \oplus I_{I_C} \oplus I_{DT} \oplus \emptyset$ $\oplus$ is the sum of types operator.

Examples:

If we have an operation `UMLClass2Table` that transforms an `UMLClass` to a `Table`, we can use it for instance to transform the `Student` class to the `T_Student` table. Thus, in this case, the *RUN* operator is invoked as follows:

$RUN(UMLClass2Table, Student)$. It returns the table $T\_Student$.

If we want to compute the age of an instance of the Student class (`Student1`), the *RUN* operator is invoked as follows:

$RUN(computeAge, Student1)$. It returns the value 26.

Next section presents the implementation of our approach on the OntoDB/OntoQL PMMS.

V. Prototyping: the BeMoRe Proposal

Our implementation consists of an extension of the OntoDB/OntoQL PMMS. Let us first introduce this PMMS.

A. The OntoDB Model Repository

The architecture of the OntoDB repository consists of four parts: one part for each abstraction level (data instance, models and metamodels) and one part for the system catalog of the database. These four parts consist of relational tables since this PMMS is based on PostgreSQL. Figure 4 (except the dashed box part) shows the main tables used to store metamodels, models and data of our example in OntoDB. The metamodel layer contains two main tables: `Class` and `Attribute` that store respectively classes and attributes of metamodels. Each class is associated to a corresponding table at the model level where class instances are stored; and similarly, each concept of a model is associated to a table at the data level to store instances.

B. The OntoQL Meta-Modeling Language

OntoQL is a declarative and object-oriented language used to create, modify, drop and query metamodels, models and data. In this section we present the OntoQL statements used for defining the different abstraction layers in OntoDB. Then, in the next section we present the extension of this language we have proposed and implemented for the definition and usage of operations at these different abstraction layers.

1) Metamodel definition: the metamodel part of OntoDB can be enriched to support new metamodels using the OntoQL language. For instance, the metamodel of our example (Figure 1) can be created with the following statements.

Listing 1: Statements for creating the metamodel (A)

```
CREATE ENTITY #UMLClass (
              #name STRING,
              #isAbstract BOOLEAN,
              #superClass REF (#UMLClass));

CREATE ENTITY #UMLProperty (
              #name STRING,
              #itsClass REF (#UMLClass));
```

In this statement the # prefix indicates that this element definition must be inserted in the metamodel level of OntoDB (an element of the model level does not have a prefix).

2) Model definition: once a metamodel is defined, we can create models conforming to that metamodel. For instance, the model of our example is created using the following statements:

Listing 2: Statements for creating the model (B)

```
CREATE #UMLClass University
PROPERTIES (name STRING);

CREATE #UMLClass Student
PROPERTIES (firstname STRING,
            lastname STRING,
            birthday DATE,
            itsUniversity REF (University));
```

3) Instance definition: similarly to the previous step, once models have been created with OntoQL and stored in OntoDB, they can be instantiated to create classes instances with a syntax similar to SQL. Next statements create instances of our example.

Listing 3: Statements for creating instances

```
INSERT INTO University VALUES ('ISAE–ENSMA');

INSERT INTO Student
VALUES ('Dupond', 'Durand', '06/21/1986', 123);
```

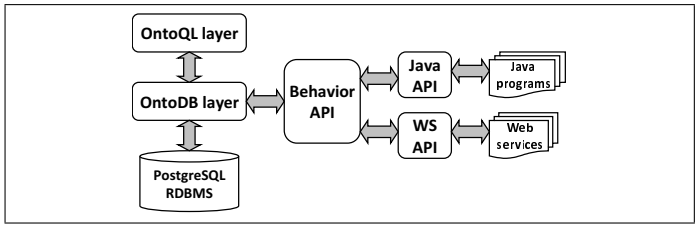

Fig. 3: BeMoRe architecture

Now that we have presented the OntoDB/OntoQL, we can describe the three main steps that we have followed to extend it with our approach (Figure 3). The first step consists of extending the model repository with structures and tables to store operations signatures and implementations descriptions and their dependencies. The second step consists of extending the exploitation language to create and to use operations and implementations. Finally, the third step consists of setting up an application programming interface (API) to make a bridge between the PMMS and the external programming environments. We detail each of these steps in next subsections.

C. Extending the OntoDB Architecture

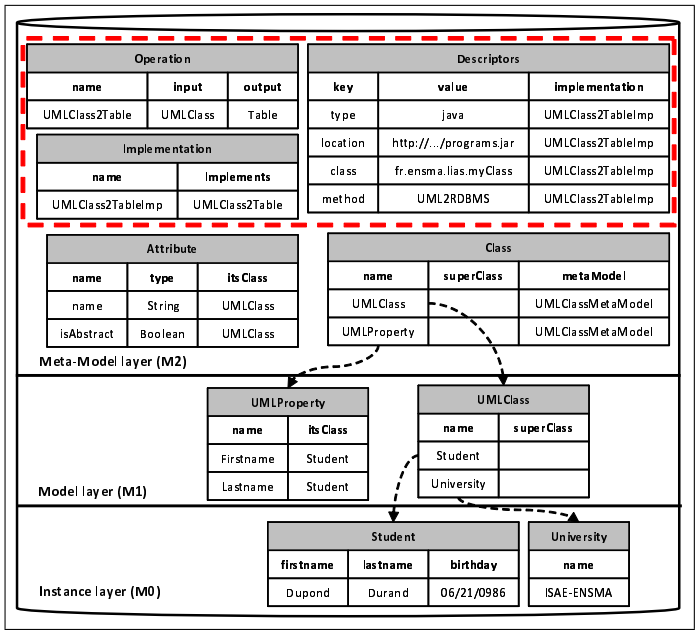

Fig. 4: Representing different model layers in OntoDB

The dashed box part of Figure 4 shows the three main tables resulting from the extension of the metametamodel layer of the OntoDB model repository. The `Operation`, `Implementation` and `Descriptors` tables store respectively operations definitions (the operation name, input and output), their associated implementations, and descriptions of implementations.

D. Extending the OntoQL Meta-Modeling Language

1) CRUD for Operations and Implementations: firstly, we have extended the OntoQL exploitation language with CRUD operations (Create, Retrieve, Update and Delete) to create, read, delete and update operations and implementations. For instance, the syntax to create an operation that transforms an UML class to a table is given below.

Listing 4: Statement for creating an operation

```
CREATE OPERATION #UMLClass2Table
INPUT (REF (#UMLClass))
OUTPUT (REF (#Table));
```

Once an operation is defined, we can define one or many associated implementations. The following statement creates an implementation of the `UMLClass2Table` operation.

Listing 5: Statement for creating an implementation

```
CREATE IMPLEMENTATION #UMLClass2TableImp
DESCRIPTORS (type = 'java',
             location = 'http://.../programs.jar',
             class = 'fr.ensma.lias.UMLClassUtils',
             method = 'class2Table')
IMPLEMENTS #UMLClass2Table;
```

This statement creates an implementation of the `UMLClass2Table` operation. It provides descriptors of a Java program stored outside the database. In particular, these descriptors specify the file location of the external program, the java class where the method is defined and the method to run.

2) Exploiting operations in Query: when an operation and at least one associated implementation are defined, this operation can be invoked in an OntoQL statement:

Listing 6: Example of an operation invocation

```
CREATE #Table T_Student AS
  SELECT #UMLClass2Table(c) FROM #UMLClass AS c;
```

This statement creates a Table (`T_Student`) from the resulting transformation of the `Student` class. Let us explain the process to answer this query. When we face an operation invocation in an OntoQL statement, we look into the repository to check the existence of the called operation, then we verify the compatibility of the arguments types with the operation parameters types. Next, if no implementation is specified in the statement, we look at the default implementation in the implementation table. After this process, we transmit the arguments and the implementation descriptors to the behavior API (see next subsection) in order to run the program and return the result.

3) Choosing a default implementation: we have also extended the OntoQL language with the possibility to define the default implementation if several implementations are available for an operation. Next statement defines a default implementation for an operation.

Listing 7: Specifying the default implementation for an operation

```
SET DEFAULT IMPLEMENTATION #UMLClass2TableImp
FOR #UMLClass2Table;
```

4) Specifying an implementation: we can also explicit the implementation that must be executed for an operation directly in an OntoQL statement. The statement below shows an example of this behavior.

Listing 8: Specifying the implementation to run in a statement

```
CREATE #Table T_Student AS
    SELECT #UMLClass2Table(c) FROM #UMLClass AS c
    USING IMPLEMENTATION #UMLClass2TableImp->#UMLClass2Table;
```

Our implementation requires to make a bridge between the PMMS and external programming environments. The solution we have adopted is presented in next subsection.

E. The Behavior API

An important part of our extension of the OntoDB/OntoQL PMMS consists of defining a mechanism to make the mapping between data types of the OntoDB/OntoQL system, and data types of the external implementations. Thus, we have set up a behavior API (Figure 3) that serves as an intermediate between the OntoDB/OntoQL PMMS and the external programming environments. In particular, it provides generic infrastructures (1) to specify data types correspondences between the two environments, (2) to execute remote programs and services, and (3) to generate a *wrap* that can be plugged on the top of the behavior API . For example, to support web services and Java methods invocation, we have implemented primitives of the behavior API and plugged on it the resulting wraps.

VI. Performance evaluation

As stated before in Section I, we have performed multiple applications of our work (e.g., [10]) in order to validate functionally our approach. In this section, we focus on performance evaluation only.

As a first step to study the scalability of our implementation, we compare the execution time of the following model query (similar results were obtained for a metamodel query).

Listing 9: The query used for our experimentations

```
SELECT computeAge(s) FROM Student AS s
```

We execute these queries using three types of implementations of the *computeAge* function: native stored procedure (*NSP*), external Java program (*EJP*) and local web service (*LWS*) on three different sizes of data (1000, 100000 and 300000 instances). These experiments were run on the OntoDB/OntoQL PMMS based on PostgreSQL 8.2 installed on a standard Intel Core Duo E6550 2.33 Ghz 3GB of RAM desktop machine.

The performance numbers for the query on the three data sizes and for the three implementations are shown in Figure 5. All times presented (in seconds) are the average of three runs of the queries.

As expected the invocation of NSP performs a factor of 4-5 faster than EJP and largely faster than LWS. As the EJP and LWS are called one time for each instance, the time of queries increases nearly linearly with the size of data. To optimize this process, this result suggests to design a Java method or a web services that takes as input a set of data instead of an individual data. A more complete study of the problem of query optimization for PMMS is part of our future work.

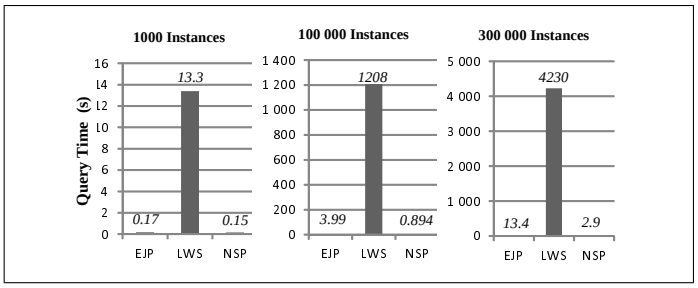

Fig. 5: Performance comparison

VII. Conclusion

In this paper, we have presented an extension of PMMS with a generic and flexible support of behavioral semantics. Our approach consists of providing the capability to introduce dynamically user-defined operations with multiple and heterogeneous implementations (external programs, web services, etc.). Our proposal has been implemented on the OntoDB/OntoQL PMMS and we have run several experiments to study the scalability of this implementation.

This work opens multiple perspectives. One of these perspectives consists of studying how derived operations could be defined using existing ones. Our idea is to be able to combine operations implemented with programs written in different languages and stored in different locations while respecting the order of the execution. Another perspective consists of choosing automatically the more efficient implementations to run when several implementations are available for a given operation. This feature will be especially useful for external web services that are not always available.

References

[1] P. A. Bernstein, A. Y. Halevy, and R. Pottinger, "A vision of management of complex models," *SIGMOD Record*, pp. 55–63, 2000.

[2] M. Jarke, M. A. Jeusfeld, H. W. Nissen, C. Quix, and M. Staudt, "Metamodelling with datalog and classes: Conceptbase at the age of 21," in *ICOODB*, 2009, pp. 95–112.

[3] D. Kensche, C. Quix, M. A. Chatti, and M. Jarke, "Gerome: A generic role based metamodel for model management," *J. Data Semantics*, vol. 8, pp. 82–117, 2007.

[4] S. Melnik, E. Rahm, and P. A. Bernstein, "Rondo: A programming platform for generic model management," in *SIGMOD Conference*, 2003, pp. 193–204.

[5] M. A. Hernández, R. J. Miller, and L. M. Haas, "Clio: a semi-automatic tool for schema mapping," in *SIGMOD Conference*, 2001.

[6] M. Koegel and J. Helming, "Emfstore : a model repository for emf models," in *ICSE (2)*, 2010, pp. 307–308.

[7] "project teresa." [Online]. Available: http://www.teresa-project.org/

[8] P. A. Bernstein and E. Rahm, "Data warehouse scenarios for model management," in *ER*, 2000, pp. 1–15.

[9] Y. Bazhar, "Handling behavioral semantics in persistent meta-modeling systems," in *RCIS*, 2012, pp. 1–6.

[10] Y. Bazhar, C. Chakroun, Y. A. Ameur, L. Bellatreche, and S. Jean, "Extending ontology-based databases with behavioral semantics," in *OTM Conferences (2)*, 2012, pp. 879–896.

[11] M. A. Jeusfeld, C. Quix, and M. Jarke, *ConceptBase .cc User Manual*, Tilburg University, RWTH Aachen, February 2013.

[12] J.-M. Hick and J.-L. Hainaut, "Strategy for database application evolution: The db-main approach," in *ER*, 2003, pp. 291–306.

[13] H. Dehainsala, G. Pierra, and L. Bellatreche, "Ontodb: An ontology-based database for data intensive applications," in *DASFAA Conference*, 2007.