Handling Behavioral Semantics in Persistent Meta-Modeling Systems

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

***Abstract*—The increasing number of information systems modeling formalisms raises several problems such as data integration or data exchange. To address these problems, several meta-modeling systems have been proposed. However, few of them use a database as a back-end repository in order to offer a persistent solution for addressing over sized models. Yet, with the growing size of data manipulated in information systems, there is a need to exploit databases properties like scalability and querying capabilities. In this paper, we present persistent meta-modeling systems with their meta-modeling capabilities. We show that these systems support the definition of structural and descriptive semantics of models, but they can not express the behavioral semantics of models. Therefore, the aim of our work is to combine benefits of classic modeling systems, typically their capability to express behavioral semantics of models elements, together with advantages of databases i.e. their scalability and querying capabilities. Our approach focuses on the capability to dynamically introduce new operators that could be exploited by model-based databases exploitation languages. In particular, such operators could be implemented with an external program stored outside the database, or with a web service. As a consequence of this extension, we will be able to perform model transformations in database, trigger web services from relational databases, information exchange and data integration could be also supported in a persistent context.**

I. Introduction

Several modeling techniques and/or notations are currently used in models design. UML for object-oriented software engineering and made generally for system design, BPMN for business process specification or Entity-Relationship for database modeling are some examples of such modeling techniques and notations. The diversity of such model notations raises two major problems : data integration of multiple models and data exchange between models. Addressing these problems requires the manipulation of models as primary objects. As a consequence, many *model management systems* (MMS) have been developed during the last two decades [1], [2], [3], [4]. These systems handle not only models and data, but also meta-models specifying the structure of these models, and support model manipulations. When these systems are equipped with operators for models processing (embedded in an exploitation language), model manipulations become possible. However, current MMS present two limitations. (1) Most existing MMS do not use a database to store all the data. Therefore, only models and data that fit in memory can be handled by these MMS. Yet many domains such as e-commerce or engineering produce over sized models and data. Thus, a major challenge is to efficiently manipulate models describing large data sets. (2) The supported operators for model management and the definition of the structural and behavioral semantics of model elements are often provided by external tools using a given specific technology. Yet these operators may pre-exist in the MMS and thus there is a need to be able to integrate them even if they are implemented with various technologies (e.g, as web services or stored procedures). To tackle these two challenges, our work proposes to use a persistent solution within database systems. More precisely, the solution we suggest is based on the OntoDB/OntoQL meta-modeling system. OntoDB is a model-based database with a prototype implemented on the PostgreSQL Database Management System (DBMS) as back-end repository. If this system supports the structural manipulation of data, models and meta-models through its associated language OntoQL, the definition of the behavior of models elements is not yet supported. Consequently, this system is not complete (in the computational sense) and needs to be extended in order to support the expression of behavioral semantics (e.g. operations, constraints, expressions, derivations, etc.). Since this system supports the manipulation of models through its meta-modeling capabilities, this extension will complement the OntoDB/OntoQL meta-modeling system and will allow us to process models in order to achieve operations such as model transformations, data integration, constraints checking, etc. The remainder of this paper is organized as follows. Section II presents a running example. Section III presents requirements for a complete meta-modeling system. Section IV exposes classic meta-modeling systems using the running example presented in section II. Section V addresses persistent meta-modeling systems. It presents model-based databases and their associated exploitation languages. The same section introduces the OntoDB/OntoQL persistent meta-modeling system (PMMS) on which our approach is based. It describes how this system supports the storage and the manipulation of meta-models, models and instances in the same database. Then we show, through an example, the limitations of this platform to define the behavior of models elements. Based on this motivating example, section VI presents our research agenda together with our approach of extending persistent meta-modeling systems and the expected results. Finally, Section VII is devoted to a conclusion.

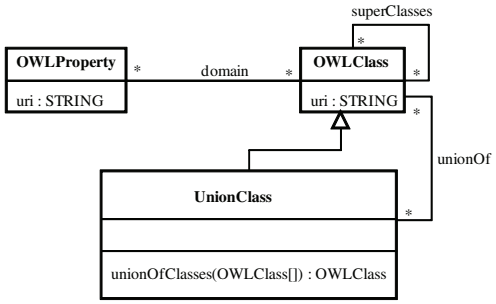

Fig. 1. A fragment of a simplified OWL meta-model

II. A Running Example

As a running example, let's consider the simplified OWL meta-model presented in figure1. The structural definition of this meta-model is the following. It defines three entities `OWLProperty`, `OWLClass` and `UnionClass` with different attributes. An `OWLClass` may have many OWL properties and many super classes. The `UnionClass` entity inherits from the `OWLClass` entity and is linked to several `OWLClass` through the `unionOf` attribute. This meta-model has also a behavioral semantics. In particular a `UnionClass` is an `OWLClass` that is derived from the union of several classes. This class may become a super class of the classes that it unifies and its instances can be computed from the instances of these classes. Using this example, the two next sections review existing MMS and show their limitations.

III. Requirements for a complete meta-modeling system

A complete model management system should offer meta-modeling capabilities, i.e. the creation and manipulation of meta-models and models with their structural, descriptive and behavioral semantics. Moreover, users need a flexible approach to define the behavior of model elements since it can be expressed with an external program, written with a given programming language, stored outside the database or with a web service. To define more precisely, the requirements for a complete persistent meta-modeling system we propose the following criteria.

1) Capability to express the structural and descriptive semantics of meta-models and models elements.
2) Capability to express the behavioral semantics of meta-models and models elements.
3) Persistence of the structural and descriptive semantics of model elements in a persistent environment, typically in a database.
4) Persistence of the behavioral semantics of model elements in a database.
5) Capability to define the behavioral semantics with a web service, an external program written with various programming language and stored outside the database, and the capability to define the behavioral semantics with both declarative languages (e.g., deductive rules) or procedural languages (e.g., Java or C++)
6) Capability to express constraints definition on meta-models and models elements and check them.

IV. Classic Meta-modeling systems (CMMS)

Classic meta-modeling systems are systems that operate in programming environments. They are equipped with a modeling language that supports the creation and the manipulation of models and their instances. This category of modeling systems supports both the definition of the structural semantics of a model and its behavioral semantics. The behavioral semantics can usually be expressed through a programming language (e.g. Java or C++). These systems offer also the possibility to define and check constraints on models.

We propose a classification of classic meta-modeling systems into the following two different categories.

Three-levels systems are systems where the meta-model part is fixed (e.g., Papyrus [5]). These systems support the definition of models with their instances conforming to meta-models supported in the meta-model level.

Four-levels systems. Unlike the two-levels or three-levels systems, this system category is compatible with the MOF [6] architecture where the meta-model part is not frozen (eg EMF [7], Kermeta [8]). Therefore, these systems support the creation of new meta-models.

If we want to manage both the meta-model presented in section II with the ontology that instanciate this meta-model and their instances (three levels of modeling), only four-levels systems can be used. We choose to use EMF to implement this example. Figure 2 gives the representation of the structural and descriptive semantics of the meta-model of our example. EMF also offers the possibility to define

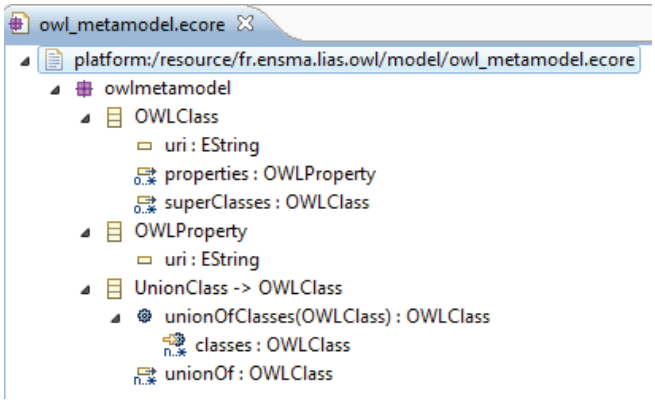

Fig. 2. Our running example defined with EMF

the behavioral semantics of model elements with the Java programming language. For example, the following operation can be defined to encode that a `UnionClass` class becomes a super class of the classes it unifies.

```
public static OWLClass unionOfClasses(List<OWLClass>
owlClasses){
    owlClass OWLClass = new OWLClass();
    owlClass.setUnionOf(owlClasses);
    for (OWLClass oc : owlClasses){
       oc.addSuperClass(owlClass);
    }
    return owlclass;
}
```

This example shows that some classic meta-modeling systems supports the definition of structural, descriptive and behavioral semantics of models elements. However, these systems present two limitations. First, the behavioral semantics of model elements can only be defined with a given programming language (e.g, Java for EMF). Yet, these behavioral semantics could also be implemented in external tools. For example, the `unionOfClasses` operation is implemented in several reasoners such as RACER in Lisp or FACT in C++. Second, these systems do not provide a persistent storage in a database for the three levels of modeling (meta-model, model and instances) with an exploitation language that supports the manipulation of all these levels.

V. Persistent Meta-modeling systems (PMMS)

By persistent meta-modeling systems we mean databases, named model-based databases (MBDB), dedicated to the storage of meta-models, models and instances and an associated exploitation language that supports the management (create, update, delete and query) of meta-models, models and instances.

A. Model-based databases

Three types of MBDBs architectures have been identified in the literature.

Type1 MBDB (Figure3) stores models and their instances in a single part. For example, Jena [9] uses a triple table to store these two levels of information. In these MBDBs the meta-model is fixed and can not be accessed.

Type2 MBDB (Figure4) separates the representation of the model and instance levels (e.g., Sesame [10], RDFSuite [11], OntoMS [12], SOR [13], DLDB [14]). Indeed, this meta-schema is composed of a set of tables that are used to store models. Another database schema is used to store instances. Thus, this architecture respects the separation of the three modeling levels. Models stored in a MBDBs type2 conform to the meta-model supported by the MBDB. But, in these MBDBs, the meta-schema is fixed but can be accessed.

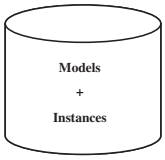

Fig. 3. Type1 MBDB architecture

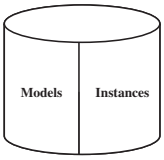

Fig. 4. Type2 MBDB architecture

Type3 MBDB (Figure5) stores explicitly the four modeling levels of the MOF [6] (e.g., OntoDB [15]). This architecture contains three parts: an extensible meta-model part dedicated to store different meta-models, a model part and a data part devoted to store respectively models and their instances. In this MBDB architecture, all the modeling levels can be manipulated.

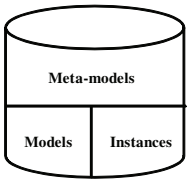

Fig. 5. Type3 MBDB architecture

Up to now, model-based databases only store the structural and descriptive semantics of meta-models and models. They do not take into account the behavioral semantics of models elements. Next subsection addresses the exploitation languages part and presents the associated MBDBs exploitation languages that are used to create and manipulate meta-models, models and instances.

B. MBDBs Exploitation languages

Model-based databases are equipped with exploitation languages to manage meta-models, models and instances. Particularly, these languages support the creation of meta-models and models by defining their structural and descriptive semantics. Some languages provide predefined operators and functions encoded in native languages (e.g. Rondo [2]) to express behavioral semantics of meta-models and models elements, but these operators are in some cases, not powerful enough to express the behavior of model elements. Other languages express the behavior using deduction rules (eg. Datalog [16]). Thus, MBDBs exploitation languages did not follow the evolution of structures and descriptors. Indeed, they do not offer the same capabilities of expressiveness as the one offered by languages of the classic meta-modeling systems, particularly for the definition of the behavioral semantics. Moreover, MBDBs exploitation languages do not offer the capability to define and check constraints on models. They do not support the multi-language aspect to support different programming languages or web services to implement the behavior of a model element. Consequently, MBDBs exploitation languages need to overcome these limitations. Indeed, they should support the expression of behavioral semantics while keeping the flexibility of the implementation. They should as well support the definition of constraints on models. Table I shows the capabilities offered by CMMS and PMMS compared to the criteria identified in section III.

C. The OntoDB/OntoQL persistent meta-modeling system

This subsection presents the OntoDB/OntoQL persistent meta-modeling system on which our approach is based. It

	1	2	3	4	5	6
CMMS	YES	YES	NO	NO	NO	YES
PMMS	YES	NO	YES	NO	-	NO

TABLE I
LIMITATION OF EXISTING MMS ACCORDING TO OUR CRITERIA

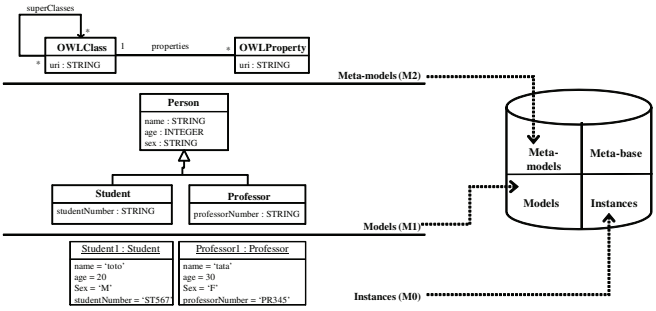

Fig. 6. Storing data of different modeling levels in OntoDB

exposes the OntoDB [15] model-based database that supports the storage of meta-models, models and instances in the same database, together with its associated exploitation language OntoQL [17]. Then, we show through our running example the limits of this system concerning the definition of the behavioral semantics of meta-model.

1) The OntoDB model-based database:

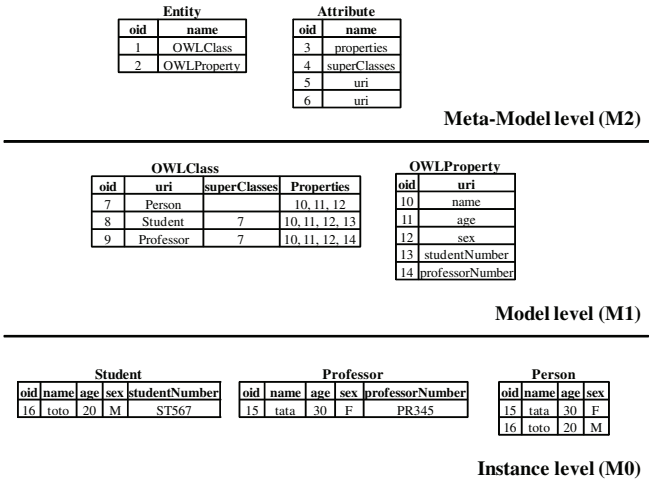

Fig. 7. Representing data of the different modeling levels in OntoDB MBDB

OntoDB is an operational type3 MBDB whose architecture is divided into four parts as illustrated in Figure 6. The first part is the traditional part available in all DBMS, namely the system catalog. It contains the system tables used to manage all the data contained in the database. The other parts of the OntoDB architecture represent different levels of abstraction of information: meta-model, model and instance levels. In the repository, data related to the different levels of information are stored in relational tables of OntoDB thanks to the services offered by the meta-model part. In the example of Figure 6, three modeling levels are represented. The meta-model level defines two entities: `OWLClass` and `OWLProperty`. `OWLClass` is the abstract description of one or many similar objects. Classes are organized in a hierarchy linked by an inheritance relationship (`superClasses`). `OWLProperty` describes properties of a class. The model level defines the `Person`, `Student` and `Professor` classes with various properties as instances of the meta-model previously defined. Finally, the instance level defines instances of the `Student` and `Professor` classes.

2) The OntoQL meta-modeling language: Since the whole data of OntoDB are stored in a database, one can think that SQL could be used to manipulate them. In this case, users need to have a deep knowledge of the internal table representation used by OntoDB to encode the three modeling levels. To overcome this limitation, OntoDB is equipped with an exploitation language named OntoQL that hides the internal representations and directly manipulates concepts of different levels. This language can be used to create meta-models, models and instances. The following two OntoQL statements illustrate how the meta-model part of OntoDB is extended with the `OWLPRoperty` and `OWLClass` concepts:

```
CREATE ENTITY #OWLProperty (#uri STRING);
CREATE ENTITY #OWLClass (#uri STRING, #superClasses
REF(#Class) ARRAY, #properties REF(#OWLProperty) ARRAY);
```

After creating the meta-model, we are able to instantiate it. The next statements show an example of the meta-model instantiation.

```
CREATE #OWLClass Person (name STRING, age INT, sex
STRING);
CREATE #OWLClass Student UNDER Person (stdtNumber INT);
CREATE #OWLClass Professor UNDER Person (ProfNumber
INT);
```

Models instances can be defined by simple SQL statements as it is presented bellow:

```
INSERT INTO Professor VALUES ('tata', 'F', 30, 'PR345');
INSERT INTO Student VALUES ('toto', 'M', 20, 'ST567');
```

The first two statements define the meta-model. The entity `OWLClass` is defined with an `uri` and the classes it extends `superClasses` (references to other classes). The entity `OWLProperty` has also an `uri` and is linked to classes it describes. In these queries, the names of entities and descriptions of entities are prefixed by the character `#`. Indeed, the meta-model level of OntoDB can be extended and modified and thus this level of information is not encoded as keywords of the OntoQL language. The next three statements define the different classes of our example with their properties using the `CREATE #OWLClass` clause. A class is defined as a subclass of an other one using the keyword `UNDER`. Finally, the last two statements define

instances of our classes using an `INSERT INTO` syntax similar to the one of SQL. Figure 7 shows how data of the different modeling levels are represented in the OntoDB MBDB. Notice that OntoDB separates the four modeling levels of the MOF architecture and only the meta meta-model level is frozen.

D. Limitation of the OntoQL/OntoDB meta-modeling system

As shown in the previous section, OntoQL can be used to create meta-models, models and instances and store them in OntoDB. However, we only saw that OntoQL can express the structural semantics of models and meta-models elements. Let us consider, for example, the `UnionClass` constructor. It supports the definition of a class as a union of other classes. Using this constructor, a class called `SchoolMember` could be defined as the union of the `Professor` and `Student` classes of our example. The OntoQL statement to extend the meta-model with this new constructor is:

```
CREATE ENTITY #UnionClass UNDER #OWLClass (#unionOf
REF(#OWLClass) ARRAY);
```

This OntoQL statement states that the structure of this constructor (i.e, an `UnionClass` is defined by a set of classes). However, we can not state within this statement nor within any other OntoQL statement that the resulting class of the union operation becomes a super class of classes that it unifies, and the instances of an `UnionClass` can be computed as the union of the instances of the classes (*Instances(C) = Instances(C1) U Instances(C2)*) used in its definition. Thus, OntoQL can not define the behavioral semantics of meta-models and models elements. This semantics could be defined by operations, like functions or procedures, and implemented with web services or external programs stored *inside* or *outside* the OntoDB database. The former is already available in some database systems but the later is still not available. For example, we should be allowed to define the semantics of the *unionOfClasses* operator through a statement such as :

```
CREATE UnionClass SchoolMember
AS unionOfClasses (Professor, Student)
```

Where *`unionOfClasses`* is an operation for which the implementation could be internal/external procedures or web service invocation. Therefore, extending OntoQL to support web services, procedures, functions triggering and constraints checking will make it possible to offer the definition of the behavioral semantics of model elements. *Offering such a capability is the main objective of our thesis.*

VI. Research agenda

A. Objectives

We have seen that persistent meta-modeling systems do not support the dynamic definition and storage of behavioral semantics and the definition and checking of constraints. These systems should therefore be extended to be as complete as classic four-levels meta-modeling systems. Indeed, MBDBs should provide structures to store information about behavioral semantics and constraints, and their associated exploitation languages should be extended to hold the expression of constraints and behavior with procedural and multi-language aspects. This extension will complement type3 MBDBs architecture and their associated exploitation languages. Consequently, we will be able to process models and achieve operations like model transformations, data integration, migration of instances in persistent environment while ensuring scalability, etc. The objective of our work is to be able to define and store the behavioral semantics of models elements and implement it with any language (Java, C++, etc) in order to exploit the power of different languages. Operations expressing the behavior of a concept may be defined and stored either inside the database (as internal programs written in a procedural language like PL/SQL), or outside the database by associating a web service as behavior or an external program.

B. Extending persistent meta-modeling systems

To fulfill the objectives previously presented, our research agenda is organized in the following way. The first part of our agenda is divided into two steps. First step consists on extending the meta meta-model part of MBDB to allow the storage of behavior. Second step consists on extending the exploitation language meta-model with a behavior meta-model supporting the creation and the invocation of operations, and constraints definition and checking. Once the previous steps are achieved, the second part of our research agenda consists in freezing the syntax of our extension of the exploitation language, implementing it and proposing a case study with a significant amount of data in order to validate our work and show the differences between classic systems and our persistent meta-modeling system.

C. Prototyping

Currently, the first steps are already experimented. Indeed, we have already demonstrated on the OntoDB/OntoQL system the feasibility of our proposed approach using a set of OntoQL statements using web service implementation and external Java function. We are currently formalizing our proposal before moving to the second part. We overview the first steps we have followed.

1) Extending OntoDB:

We have extended OntoDB architecture with the entity *Operation* at the meta meta-model level. This extension allows us to store information about concepts behaviors (operations names, input types, output types) and their implementations details like a web service location, the operation name of the service to invoke, the location of a JAVA function, etc. Figure 8 shows the new entity (*Operation*) added at the meta meta-model level of the OntoDB MBDB.

2) Extending OntoQL: We have extended the OntoQL exploitation language with the support of operation creation. The following statement shows an example of creating an

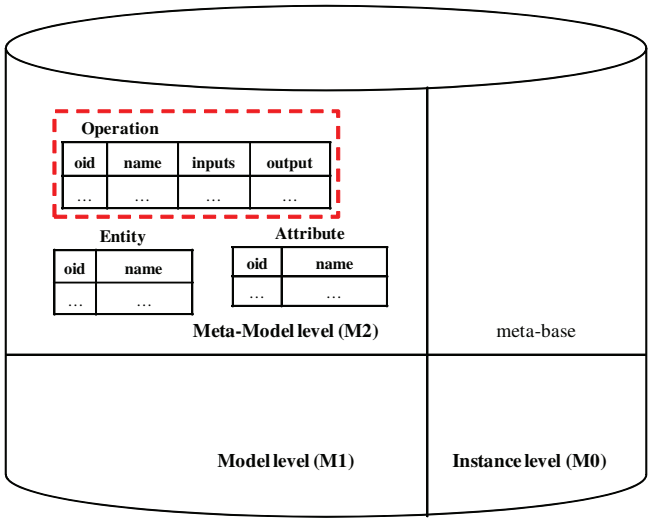

Fig. 8. The extension of the meta meta-model level of OntoDB with the Operation entity

operation with OntoQL.

```
CREATE OPERATION UnionOfClasses
INPUTS REF(#OWLClass) ARRAY
OUTPUT REF(#OWLClass);
```

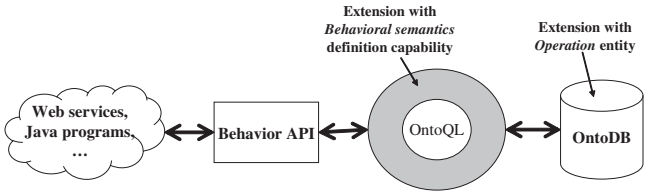

Fig. 9. The behavior API

We are currently working on extending OntoQL to integrate the support of operations invocations in OntoQL statements to exploit operations defined. In this direction, we are setting up an application programming interface (API) (Figure 9) that interacts with the external world (web services and external programs). This API will overcome problems of mapping data types defined in the OntoDB/OntoQL system and data types used in the external implementations. It will establish data types correspondences between those defined in our systems and those defined in the external environment. The API will also define a format for data exchange.

VII. Conclusion

In this paper we have introduced our work on meta-modeling systems. Our objective is to design a complete meta-modeling system that (1) manages all data in a database to get benefits from the properties of databases (2) and is able to define the dynamic semantics of model elements through procedural components. Our research agenda consists in defining different meta-models to support a wide range of procedural components that can be defined inside or outside the repository with different programming languages. Since these procedural components can be applied on stored models, our proposed extension could be used to perform model transformations in databases and thus to transform a big amount of data. As a perspective of this work, we expect to apply our approach to manage models in databases like in classic systems. Indeed, we expect to apply our approach to perform model transformations inside a MBDB.

References

[1] M. A. Jeusfeld, M. Jarke, and J. Mylopoulos, *Metamodeling for Method Engineering*. MIT press, 2009.

[2] S. Melnik, E. Rahm, and P. A. Bernstein, "Rondo: a programming platform for generic model management," in *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*, 2003.

[3] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008.

[4] P. Atzeni, P. Cappellari, and P. A. Bernstein, "Model independent schema and data translation (extended abstract)," in *SEBD*, 2005, pp. 177–183.

[5] Www.papyrusuml.org/.

[6] Http://www.omg.org/mof/.

[7] Www.eclipse.org/emf/.

[8] Www.kermeta.org/.

[9] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: Implementing the Semantic Web Recommendations," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW'04)*. New York, NY, USA: ACM Press, 2004, pp. 74–83.

[10] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *Proceedings of the 1st International Semantic Web Conference (ISWC'02)*, ser. Lecture Notes in Computer Science, I. Horrocks and J. Hendler, Eds., no. 2342. Springer Verlag, July 2002, pp. 54–68.

[11] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle, "The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases," in *Proceedings of the 2nd International Workshop on the Semantic Web*, 2001, pp. 1–13.

[12] M. J. Park, J. H. Lee, C. H. Lee, J. Lin, O. Serres, and C. W. Chung, "An Efficient and Scalable Management of Ontology," in *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, ser. Lecture Notes in Computer Science, vol. 4443. Springer, 2007, pp. 975–980.

[13] J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, and Y. Yu, "Sor: a practical system for ontology storage, reasoning and search," 2007, pp. 1402–1405.

[14] Z. Pan and J. Heflin, "DLDB: Extending Relational Databases to Support Semantic Web Queries," in *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, 2003, pp. 109–113.

[15] H. Dehainsala, G. Pierra, and L. Bellatreche, "Ontodb: An ontology-based database for data intensive applications," in *Proc. of the 12th Int. Conf. on Database Systems for Advanced Applications (DASFAA'07). LNCS*. Springer, 2007.

[16] 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 *Proceedings of the Second international conference on Object databases*, ser. ICOODB'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 95–112. [Online]. Available: http://dl.acm.org/citation.cfm?id=1883713.1883719

[17] S. Jean, Y. Aït-Ameur, and G. Pierra, "A language for ontology-based metamodeling systems," in *Proceedings of the 14th east European conference on Advances in databases and information systems*, ser. ADBIS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 247–261. [Online]. Available: http://portal.acm.org/citation.cfm?id=1885872.1885894