Persistent Meta-Modeling Systems as Heterogeneous Model Repositories

Youness Bazhar[1], Yassine Ouhammou[1], Yamine Aït-Ameur[2], Emmanuel Grolleau[1], and Stéphane Jean[1]

[1] LIAS/ISAE-ENSMA and University of Poitiers, Futuroscope, France.
{bazhary, ouhammoy, grolleau, jean}@ensma.fr
[2] IRIT/INP-ENSEEIHT, Toulouse, France.
yamine@enseeiht.fr

Abstract. Model persistence has always been one of the major interests of the model-driven development community. In this context, Persistent Meta-Modeling Systems (PMMS) have been proposed as database environments dedicated to meta-modeling and model management. Yet, if existing PMMS store meta-models, models and instances, they provide mechanisms that are sometimes insufficient to accomplish some advanced model management tasks like model transformation or model analysis. In this paper we validate the work achieved in [5] by exploiting the support of user-defined operations in PMMS in order to perform model transformations and model analysis.

Keywords: meta-modeling, model management, database.

1 Introduction

Recently, the use of model-based engineering technologies and modeling tools has widely increased especially in industrial contexts. This generates very often, in case of an excessive use, large scale models which raise the issue of scalability as one of the major weaknesses of applying modeling in real industrial contexts. Moreover, industries need also to share and exchange voluminous models and data, and a simple file exchange may sometimes be insufficient. These issues are industrial and scientific challenges and consequently, we need platforms to (i) overcome issues of scalability, (ii) surmount problems related to the heterogeneity of models, and (iii) provide a common repository for model sharing. Thus, Persistent Meta-Modeling Systems (PMMS) have been proposed as the leading solution that satisfy all these requirements. Indeed, a PMMS is a meta-modeling and model management system equipped with (1) a database that stores meta-models, models and instances, and (2) an associated exploitation language that possesses meta-modeling and model management capabilities. But, if existing PMMS support the definition and the storage of meta-models, models and instances, they provide mechanisms that are not adapted to accomplish some advanced model management tasks like model transformation, code generation,

model analysis, etc. This is due to the lack of developed behavioral semantics in PMMS since current PMMS offer mechanisms that are either specific to the database or to the domain the PMMS is dedicated to. Thus, in a recent work [5], we have proposed an extension of PMMS with the support of behavioral semantics with wide programming capabilities. Indeed, this proposition consists of introducing dynamically user-defined operations that can be implemented by web services and external programs written in any language (e.g., Java, C++). These operations can manipulate complex types (e.g., classes and meta-classes) as well as simple types (e.g., string, integer). This work has been validated with an application for handling derived ontologies concepts [6]. The contribution of this paper is to show how model transformations and model analysis can be achieved in PMMS using the approach presented in [5]. This application is prototyped with an implementation on the OntoDB/OntoQL PMMS [9].

The remainder of this paper is organized as follows. Section 2 introduces a motivating example that raises the need of operations in PMMS for model management. Section 3 gives an overview on the state of the art. Section 4 exposes the OntoDB/OntoQL PMMS on which our approach is based. Section 5 presents our approach for extending PMMS with the support of user-defined operations. Section 6 presents a model transformation and a model analysis case studies that show the usefulness of extending PMMS with operations. Finally, Section 7 is devoted to a conclusion.

2 A Motivating Example

This section presents an example of a real-time system that we use throughout this paper. The aim of this example is to design an uniprocessor system with three periodic tasks (*T1*, *T2* and *T3*). Each task is characterized by a period P, a deadline D, and a worst-case execution time ET. The system scheduling follows the EDF (Earliest Deadline First) scheduling policy. This system is defined as a set of tasks: $S =< T1, T2, T3 >$, where:

$T1 =< P = 29ms, D = 29ms, ET = 7ms >$

$T2 =< P = 5ms, D = 5ms, ET = 1ms >$

$T3 =< P = 10ms, D = 10ms, ET = 2ms >$

This kind of systems can be designed using languages dedicated to design real-time and embedded systems like AADL [4] (see Figure 1) or MARTE [2]. Indeed, AADL (Architecture Analysis and Design Language) is an architecture description language dedicated to describe components and their hierarchical composition, while MARTE (Modeling and Analysis of Real Time and Embedded systems) is a modeling language dedicated to design both software and hardware aspects of real-time and embedded systems and supports schedulability analysis.

AADL and MARTE could express the system of our example using different constructors and following different methodologies. One of the major differences between these two languages is that AADL is more oriented towards architecture description and does not offer the capability to analyze the schedulability of

```
system embsys
end embsys;
system implementation embsys.Impl
  subcomponents
    cpu: processor cpu_embsys.Impl;
    proc: process process_embsys.Impl;
  properties
    Actual_Processor_Binding =>  reference cpu applies to proc;
end embsys.Impl;
processor cpu_embsys
end cpu_embsys;
processor implementation cpu_embsys.Impl
  properties
    Scheduling_Protocol => EDF;
end cpu_embsys.Impl;
process process_embsys
end process_embsys;
process implementation process_embsys.Impl
  subcomponents
    T1: thread Task.Impl1;
    T2: thread Task.Impl2;
    T3: thread Task.Impl3;
end process_embsys.Impl;
```

```
thread Task
end Task;
thread implementation Task.Impl1
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 7 Ms .. 7 Ms;
    Deadline => 29 Ms;
    Period => 29 Ms;
end Task.Impl1;
thread implementation Task.Impl2
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 1 Ms .. 1 Ms;
    Deadline => 5 Ms;
    Period => 5 Ms;
end Task.Impl2;
thread implementation Task.Impl3
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 2 Ms .. 2 Ms;
    Deadline => 10 Ms;
    Period => 10 Ms;
end Task.Impl3;
```

Fig. 1. The system S of our example expressed with AADL

systems, while MARTE meets this need. Here, we can already see the problem of heterogeneous modeling that appears in the design of complex systems. Thus, analyzing an AADL model schedulability must go through a model transformation to MARTE.

Our objective is to be able to share the system of our example independently of the formalism used to express it. For this, the PMMS shall provide mechanisms to be able to transform AADL models to MARTE ones and vice versa. Moreover, we would like to be able to analyze the schedulability of our system regardless the formalism used to design it. Achieving these model management tasks require operators.

Next section presents existing PMMS and discusses their capabilities and limitations concerning behavioral semantics.

3 Related Work

In our study of the state of the art, we have classified model persistence systems into two types: model repositories and their exploitation languages that only serve to store and retrieve models, and database environments for meta-modeling and model management. This section presents and discusses these two model persistence systems.

3.1 Model Repositories and their Exploitation Languages

Some meta-modeling systems are equipped with persistent repositories that are dedicated to store meta-models, models and instances [7]. These repositories use many back ends to store the different abstraction layers such as relational, NoSQL or XML databases. Main examples of these model repositories are dMOF

[8], MDR [19], EMFStore [18] and Morsa [10]. These repositories store MOF [1] and UML [3] models, and focus mainly, like in [11], on the architecture of the repository. As a consequence, they serve only as model warehouses in the sense that they do not offer a persistent environment for meta-modeling nor model management since all meta-modeling and model management tasks require loading models and instances from the repository and processing them in main memory.

Persistent model repositories are equipped with declarative query languages restricted only to querying capabilities. Main examples of model repositories query languages are mSQL [23], P-OQL [13], SQL/M [17], iRM/mSQL [22] and MQL [12]. These languages do not possess neither meta-modeling nor model management capabilities, and thus they remain high-level query languages only.

Persistent model repositories and their associated query languages do not offer persistent environments for meta-modeling and model management.

3.2 Persistent Model Management Systems

To manage models inside the database, several PMMS have been proposed like ConceptBase [15], Rondo [21], Clio [14] and OntoDB/OntoQL [9]. These PMMS handle the structural semantics of models by offering constructors of (meta-)classes, (meta-)attributes, etc. Yet, they provide hard-encoded mechanisms to express behavioral semantics such as predefined operators (like in Rondo and Clio), or use classic database procedural languages (e.g., PL/SQL) which cannot manipulate complex types (e.g., meta-classes or classes). The most advanced PMMS remain ConceptBase since it gives the possibility to introduce user-defined functions with external implementations. However, these implementations 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 [16].

As the previous overview of the state of the art shows, current PMMS do not support the definition on the fly of model management operations that can be implemented using external programs and web services hence the necessity to extend PMMS with such capabilities.

Next section presents the OntoDB/OntoQL PMMS that we use to implement our approach.

4 The OntoDB/OntoQL Persistent Meta-Modeling System

OntoDB/OntoQL [9] is a four-layered persistent meta-modeling system where only the meta meta-model is hard-encoded (see Figure 2) so that we can extend on the fly the meta-model layer in order to integrate different meta-modeling formalisms (warm start). This system is equipped with the OntoDB model repository and the OntoQL exploitation language that we present in the next subsections.

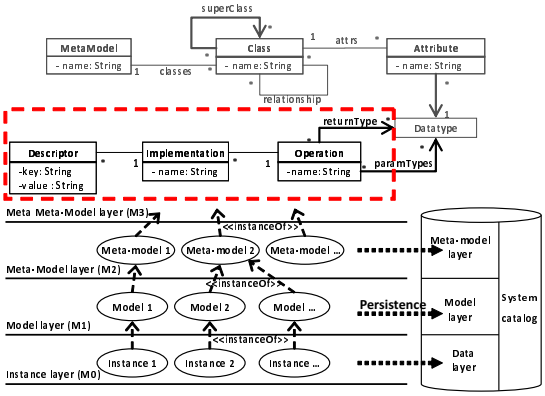

Fig. 2. The architecture of OntoDB/OntoQL

4.1 The OntoDB model repository

The OntoDB model repository architecture consists of four parts (Figure 2). The *system catalog* and *data layer* parts are the classical parts of traditional databases. The *system catalog* part contains tables used to manipulate the whole data stored in the the database, and the *data layer* part stores instances of models. The *meta-model layer* and *model layer* parts store respectively meta-models and models. Note that OntoDB respects the separation of the different storage layers and preserves the conformity of models and instances.

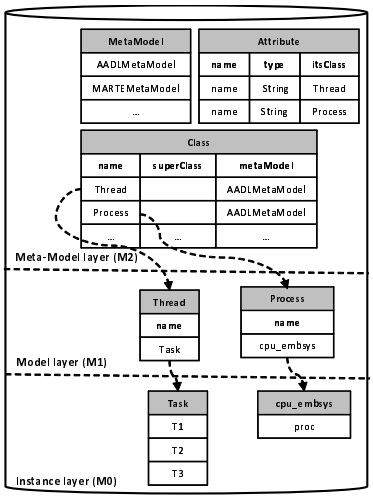

Fig. 3. Data representation in the OntoDB model repository

OntoDB stores data in relational tables since it is implemented on the PostgreSQL RDBMS. Figure 3 shows the representation of some concepts of our example. The meta-model layer contains three main tables: `MetaModel`, `Class` and `Attribute` that store respectively meta-models, classes and attributes. Each class is associated to a corresponding table at the model layer that stores class instances. Similarly, each concept at the model layer is associated to a corresponding table at the data layer to store instances.

4.2 The OntoQL meta-modeling language

OntoQL has been defined in order to facilitate the exploitation of model repositories. OntoQL is a declarative and object-oriented language owning meta-modeling and querying capabilities. Indeed, OntoQL has been proposed to create and manipulate meta-models, models and data without any knowledge of the structure of tables and their relationships. One of the benefits of OntoQL is that it guarantees a large flexibility of expressiveness so that we can create meta-models and models on the fly. This subsection shows how we can create meta-models and models using OntoQL.

Meta-model definition. the meta-model part of the OntoDB model repository can be enriched to support new meta-models using the OntoQL language. Below we give some of the OntoQL statements for defining the AADL meta-model (Listing 1.1).

Listing 1.1. A subset of OntoQL statements for creating the AADL meta-model

```
CREATE ENTITY #Property (
       #name STRING,
       #value STRING);

CREATE ENTITY #ProcessSubComponent;

CREATE ENTITY #ThreadClassifier UNDER #ProcessSubComponent;

CREATE ENTITY #ThreadType UNDER #ThreadClassifier (
       #name STRING
       #extends REF (#ThreadType));

CREATE ENTITY #ThreadImpl UNDER #ThreadClassifier (
       #name STRING,
       #properties REF (#Property) ARRAY,
       #implements REF (#ThreadClassifier),
       #extends REF (#ThreadImpl));
```

Model definition. once a meta-model is defined and supported by the OntoDB/OntoQL platform, we become able to create models conforming to that meta-model. Next statements create the AADL model of our example.

```
CREATE #ThreadType Task;

CREATE #ThreadImpl Task.Impl1
PROPERTIES (
```

```
            Dispatch_Protocol = Periodic,
            Compute_Execution_Time = 7Ms..7Ms,
            Deadline = 29Ms,
            Period = 29Ms)
IMPLEMENTS Task;
```

OntoQL possesses also querying capabilities so that it makes possible to query the different layers. Moreover, OntoQL supports the other persistence basics (UPDATE and DELETE) for meta-models, models and instances.

Limitations of OntoDB/OntoQL. at this level, we only obtain an AADL model of our system. And if we need to derive the corresponding MARTE model of our system, this requires (1) storing the MARTE meta-model in OntoDB and (2) defining a model transformation from AADL to MARTE using operations. The first step is feasible as OntoDB/OntoQL supports the definition on the fly of meta-models and models. Yet, the second step cannot be accomplished since OntoDB/OntoQL does not support the definition of operations on meta-models and models elements. Moreover, OntoDB/OntoQL cannot allow us to analyze the schedulability of our system for the same reason. Indeed, we need an operator that computes the schedulability by invoking an analysis test. Thus, OntoDB/OntoQL has to be extended in order to overcome this limitation. Next section presents the extension of OntoDB/OntoQL to support behavioral semantics.

5 Extending Persistent Meta-Modeling Systems with Behavioral Semantics

In order to handle behavioral semantics in PMMS, we have extended the meta meta-model supported by the OntoDB/OntoQL PMMS with the concepts of *Operation*, *Implementation* and *Descriptor* as shown in the dotted box of Figure 2. They represent respectively a function or a procedure, its associated implementations and implementations descriptors. The extension of the OntoDB/OntoQL system took place in two main stages detailed in next subsections.

5.1 Extending the OntoDB model repository

The first stage concerns the extension of the meta meta-model layer at repository level with tables that store operations definitions, implementations and implementations descriptions. Figure 4 shows the main tables resulted from the extension of the meta meta-model layer of OntoDB. The `Operation`, `Implementation` and `Descriptor` tables store respectively operations signatures (the operation name, inputs and outputs), implementations and descriptions of these implementations.

5.2 Extending the OntoQL meta-modeling language

The second stage of our PMMS extension consists in enriching the OntoQL language with the capability to create and exploit operations and implementations.

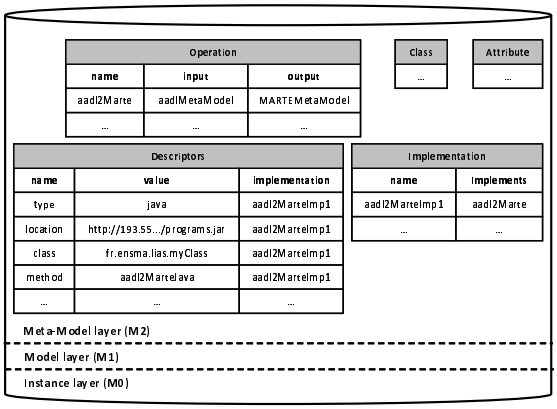

Fig. 4. The extension of the meta meta-model layer of OntoDB

OntoQL has been enhanced with CRUD (CREATE, READ, UPDATE and DELETE) basics permitting to create, read, delete and update operations and implementations. An example of the established OntoQL syntax is shown through the following statements.

```
CREATE OPERATION #rule1
INPUT (REF (#SystemType),
       REF (#SystemImpl))
OUTPUT (REF (#saAnalysisContext),
        REF (#gaResourcesPlatform));

CREATE IMPLEMENTATION #rule1JavaImp
DESCRIPTORS (
    type = 'java',
    location = '193.55.../programs.jar',
    class = 'fr.ensma.lias.AadlToMarte',
    method = 'rule1Imp')
IMPLEMENTS #rule1;
```

The first statement creates an operation that has an AADL `System` and `SystemImpl` input, and a MARTE `saAnalysisContext` and `gaResourcesPlatform` model output. The second statement defines an implementation of the operation previously defined by providing a set of meta-data of a remote Java program stored outside the database implementing this operation. This meta-data is exploited to run the remote program.

The other aspect of extending of the OntoDB/OntoQL PMMS was to set up mechanisms that 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 (Application Programming Interface) that serves as an intermediate layer between the OntoDB/OntoQL world and the external world. In particular, it provides generic infrastructures to specify data types correspondences between the two worlds, and to execute remote programs and services.

6 Managing models within PMMS

This section is devoted to show the usefulness, for model management, of extending PMMS with behavioral semantics. We firstly show how we can define transformation operations of AADL models to MARTE ones, and then we show how we can use operations for real-time models analysis.

	AADL	**MARTE**
rule 1	`SystemType` and `SystemImpl`	`saAnalysisContext` and `gaResourcesPlatform` stereotypes
rule 2	`SystemClassifier` subcomponent	Specified by `Resources` tagged value of `gaResourcesPlatform`
rule 2.1	`ProcessType` and `ProcessImpl`	`MemoryPartition` and `Scheduler`
rule 2.2	`ProcessorType` and `ProcessorImpl`	`hwProcessor` stereotype
rule 2.3	`ProcessorImpl` properties `scheduling_protocol` property	Specified by the `schedPolicy` tagged value of `Scheduler` stereotype.
rule 2.3.1	`ProcessClassifier` subcomponent	`SchedulableResources` tagged value of `Scheduler` stereotype
rule 2.3.1.1	`ThreadType` and `ThreadImpl`	`swSchedulableResource` and `saStep`
rule 2.3.1.2	`ThreadImpl` properties: - `dispatch_protocol` and `period` - `deadline` - `compute_execution_time`	Tagged values of `swSchedulableResource` - Specified by `type` tagged value (a type of `ArrivalPattern`) In this case: tagged values of `saStep` - Specified by `deadline` tagged value - Specified by `execTime` tagged value
rule 2.4	`SystemImpl` properties - `actual_procesor_binding`	Specified by `processingUnits` tagged value of `Scheduler` stereotype

Table 1. AADL to MARTE transformation rules

AADL to MARTE transformation: while AADL is dedicated to design system architectures, MARTE is structured around two main concerns: one to model the features of real-time and embedded systems and the other to annotate application models in order to support analysis of system properties.

Table 1 summarizes the different concept mappings from AADL to MARTE. These transformation rules are introduced in order to justify the operations, we define later in the paper, for transforming AADL models to MARTE ones.

6.1 Using Operations for Model Transformation

We precise that our objective is not to propose a new transformation approach, neither to guarantee a safe transformation from AADL to MARTE. Several works have addressed the transformation form AADL to MARTE.

As it has been stressed before, our objective is to use the possibility to introduce operations on the fly in the OntoDB/OntoQL system in order to transform

AADL models into MARTE ones. Indeed, we firstly create model transformation operations that will transform AADL concepts to MARTE ones, then we create corresponding implementations. These two steps are explained below in detail.
Definition of transformation operations: the definition of AADL to MARTE transformation operations consists of specifying for each operation its name, its eventual input and output types. The following statements define some of the essential transformation operations based on rules defined in Table 1.

Listing 1.2. AADL to MARTE transformation operations

```
CREATE OPERATION #rule1
INPUT (REF (#SystemType),           // AADL source elements
       REF (#SystemImpl))
OUTPUT (REF (#saAnalysisContext), // MARTE target elements
        REF (#gaResourcesPlatform));

CREATE OPERATION #rule2
INPUT (REF (#SystemSubComponent) ARRAY)
OUTPUT (REF (#Resource) ARRAY);

CREATE OPERATION #rule2.1
INPUT (REF (#ProcessType),
       REF (#ProcessImpl))
OUTPUT (REF (#MemoryPartition),
        REF (#Scheduler));

CREATE OPERATION #rule2.2
INPUT (REF (#ProcessorType),
       REF (#ProcessorImpl))
OUTPUT (REF (#hwProcessor));
```

The `#rule1` operation transforms a `SystemType` and its associated `SystemImpl` of an AADL model to their corresponding concepts in MARTE (`saAnalysisContext` and `gaResourcesPlatform`). The `#rule2` operation transforms `SystemClassifier` subcomponents of an AADL model to `Resouce` elements of the `gaResourcesPlatform`.
Definition of Implementations: once we have defined model transformation operations, we establish their associated implementations descriptions. The following statements define implementations descriptions of the operations previously defined.

```
CREATE IMPLEMENTATION #rule1JavaImp
DESCRIPTORS (
    type = 'java',
    location = '193.55.../programs.jar',
    class = 'fr.ensma.lias.AadlToMarte',
    method = 'rule1Imp')
IMPLEMENTS #rule1;


CREATE IMPLEMENTATION #rule2JavaImp
DESCRIPTORS (
    ...
    method = 'rule2Imp')
IMPLEMENTS #rule2;

CREATE IMPLEMENTATION #rule2.1JavaImp
DESCRIPTORS (
    ...
    method = 'rule2.1Imp')
IMPLEMENTS #rule2.1;
```

```
CREATE IMPLEMENTATION #rule2.1JavaImp
DESCRIPTORS (
    ...
    method = 'rule2.2Imp')
IMPLEMENTS #rule2.2;
```

Exploiting defined operations: after defining operations and their implementations descriptions, we become able to invoke the defined operations in order to transform AADL concepts to MARTE ones. For instance, we can use the established operations for obtaining our AADL source model expressed in the MARTE formalism. We can also transform the same model to a MARTE model in order to have two representations of our system.

```
CREATE #saAnalysisContext marte_embsys
AS SELECT #rule1 (embsys, embsys.Impl)
   FROM #SystemClassifier;

CREATE #gaResourcesPlatform marte_embsys
AS SELECT #rule1 (embsys, embsys.Impl)
   FROM #SystemClassifier;

CREATE #MemoryPartition marte_memory
AS SELECT #rule2.1 (proc_embsys, proc_embsys.Impl)
   FROM #ProcessClassifier;

CREATE #Scheduler marte_scheduler
AS SELECT #rule2.1 (proc_embsys, proc_embsys.Impl)
   FROM #ProcessClassifier;

CREATE #gaResourcesPlatform marte_embsys_cpu
AS SELECT #rule2(embsys_cpu) FROM #Processor;
```

The first statement selects the MARTE system resulted from the transformation of the AADL model of our example, while the second statement reads the resulted MARTE memory and scheduler concepts from the transformation of the `proc_embsys` process element of the AADL model of our system. Whereas, the last statement creates a `gaResourcesPlatform` instance from the resulting transformation of the `embsys_cpu` processor of the AADL model.

These are only examples of the multiple transformation operations and operation invocations we have written in order to permit a complete transformation and mapping from AADL to MARTE. This eases accessing, updating, deleting and transforming AADL models even if we do not adopt AADL as a main language for designing real-time and embedded system. We can also go further by setting up an operation that analyzes the schedulability of our AADL model that we detail in the next subsection.

6.2 Using Operations for Model Analysis

Once the MARTE model is obtained after this transformation, it becomes possible to trigger scheduler analysis on these MARTE models.

So, to analyze the schedulability of the system of our example, we define an operation whose objective is to analyze the schedulability of our system. To achieve this task, we create an operation `isSchedulable` that takes as input

a MARTE model and returns as output a boolean value which states whether the systems is schedulable or not. This operation is implemented with a Java program that invokes the MAST analysis tool [20].

```
CREATE OPERATION #isSchedulable
INPUT (REF (#MARTEModel))
OUTPUT (BOOLEAN);

CREATE IMPLEMENTATION #isSchedulableJavaImp
DESCRIPTORS (
      type = 'Java',
      location = '193.55.../programs.jar',
      class = 'fr.ensma.lias.Analyzer',
      method = 'isSchedulerImp')
IMPLEMENTS #isSchedulable;
```

Now, it becomes possible to run the analysis by invoking the previous operation in the following statement.

```
SELECT #isSchedulable(#aadl2Marte(embsys.Impl)) FROM #SystemImpl
```

This statement asserts whether the `embsys` system, transformed from AADL to MARTE by the `#aadl2Marte` operation, is schedulable or not. Here, we analyze the schedulability of the corresponding MARTE model of our system. Note also that the invocation of the `isSchedulable` operation requires providing a MARTE model as input and thus, we provide an operation invocation as an argument of the `#isSchedulable` operation since OntoDB/OntoQL supports such manipulation. The previous statement invokes the `isSchedulable` analysis operation which is specific to MARTE, using an AADL resource hiding the transformation process to the user.

7 Conclusion and Perspectives

In this paper, we have validated our approach presented in [5] by using operations in PMMS for model transformation and model analysis. The use of operations enables particularly sharing models regardless of the language used to design them.

The work presented in this paper opens many perspectives. We expect to enhance our approach by integrating membership constraints so that operations can be defined only for a specific class like in object-oriented programming. Another perspective consists of defining an object constraint language to express static semantics in persistent meta-modeling systems by storing and evaluating the expression of invariants, contracts, and so on that can be associated to classes and operations.

References

1. Meta object facility (mof). Technical report, Object Management Group, August 2011.
2. Uml profile for marte : Modeling and analysis of real-time embedded systems. Technical report, Object Management Group, June 2011.

3. Unified modeling language (uml). Technical report, Object Management Group, August 2011.
4. Architecture analysis & design language (aadl). Technical report, SAE International, September 2012.
5. Youness Bazhar, Yamine Aït Ameur, and Stéphane Jean. Bemore: a repository for handling models behaviors. In *SEKE*, June 2013.
6. Youness Bazhar, Chedlia Chakroun, Yamine Aït Ameur, Ladjel Bellatreche, and Stéphane Jean. Extending ontology-based databases with behavioral semantics. In *OTM Conferences (2)*, pages 879–896, 2012.
7. Philip A. Bernstein and Umeshwar Dayal. An overview of repository technology. In *VLDB*, pages 705–713, 1994.
8. Cooperative Research Centre for Distributed Systems Technology (DSTC). *dMOF version 1.1 user guide*, 2000.
9. Hondjack Dehainsala, Guy Pierra, and Ladjel Bellatreche. Ontodb: An ontology-based database for data intensive applications. In *DASFAA Conference*, 2007.
10. Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa: A scalable approach for persisting and accessing large models. In *MoDELS*, pages 77–92, 2011.
11. Javier Espinazo-Pagán and Jesús García-Molina. A homogeneous repository for collaborative mde. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 56–65, New York, NY, USA, 2010. ACM.
12. David Hearnden, Kerry Raymond, and Jim Steel. Mql: a powerful extension to ocl for mof queries. In *EDOC*, pages 264–277, 2003.
13. Andreas Henrich and Praktische Informatik Fachbereich Elektrotechnik. P-oql: an oql-oriented query language for pcte. In *In Proc. 7th Conf. on Software Engineering Environments*, pages 48–60. IEEE Computer Society Press, 1995.
14. Mauricio A. Hernández, Renée J. Miller, and Laura M. Haas. Clio: a semi-automatic tool for schema mapping. In *SIGMOD Conference*, 2001.
15. Matthias Jarke, Manfred A. Jeusfeld, Hans W. Nissen, Christoph Quix, and Martin Staudt. Metamodelling with datalog and classes: Conceptbase at the age of 21. In *ICOODB*, pages 95–112, 2009.
16. Manfred A. Jeusfeld, Christoph Quix, and Matthias Jarke. *ConceptBase .cc User Manual*. Tilburg University, RWTH Aachen, February 2013.
17. William Kelley, Sunit Gala, Won Kim, Tom Reyes, and Bruce Graham. Modern database systems. chapter Schema architecture of the UniSQL/M multidatabase system, pages 621–648. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.
18. Maximilian Koegel and Jonas Helming. Emfstore : a model repository for emf models. In *ICSE (2)*, pages 307–308, 2010.
19. Martin Matulla. *Netbeans Metadata Repository*. 2003.
20. Julio L. Medina, Julio L. Medina Pasaje, Michael Gonzlez Harbour, and Jos M. Drake. Mast real-time view: A graphic uml tool for modeling object-oriented real-time systems. In *In the 22nd IEEE Real-Time Systems Symposium (RTSS01*, pages 245–256, 2001.
21. Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD Conference*, 2003.
22. Ilia Petrov, Stefan Jablonski, Marc Holze, Gabor Nemes, and Marcus Schneider. irm: An omg mof based repository system with querying capabilities. In *ER*, 2004.
23. Ilia Petrov and Gabor Nemes. A query language for mof repository systems. In *OTM Conferences (1)*, pages 354–373, 2008.