sosym manuscript No.
(will be inserted by the editor)

A repository for scalable model management

Javier Espinazo Pagán[1], **Jesús Sánchez Cuadrado**[2], **Jesús García Molina**[1]

[1] Universidad de Murcia, Spain e-mail: jespinazo@um.es, e-mail: jmolina@um.es
[2] Universidad Autónoma de Madrid, Spain e-mail: jesus.sanchez.cuadrado@uam.es

Received: date / Revised version: date

Abstract Applying Model-Driven Engineering (MDE) in industrial-scale systems requires managing complex models which may be very large. These models must be persisted in a way that allows their manipulation by client applications without fully loading them.

In this paper we propose Morsa, a model repository that provides scalable manipulation of large models through load on demand and incremental store; model persistence is supported by a NoSQL database. We discuss some load on demand and incremental store algorithms as well as a database design. A prototype that integrates transparently with EMF is presented and its evaluation demonstrates that it is capable of fully managing large models with a limited amount of memory. Moreover, a set of benchmarks has been executed, exhibiting better performance than the EMF XMI file-based persistence and the most widely used model repository, CDO.

Key words MDE, model persistence, model repositories, model scalability, large models, NoSQL, document databases

1 Introduction

The increasing maturity of Model-Driven Engineering (MDE) technologies is promoting their adoption by large companies [1][2], taking advantage of their benefits in terms of productivity, quality and reuse. However, applying MDE in this context requires industry-scale tools that can operate with very large and complex models. Model-driven modernization [3][4] is an example of scenario where these tools would be needed in order to efficiently manage very large and complex models extracted from source code [3] or data [5] of legacy artifacts. One basic operation of such tools is model persistence and the corresponding model access, and they must satisfy two essential requirements: scalability and tool integration.

One critical concern for the industrial adoption of MDE is the *scalability* of tools when accessing large models. As noted by [6], *"scalability is what is holding back a number of potential adopters"*. Several scenarios can be defined for scalability on client applications, depending on the kind of access and manipulation done to persisted models; e.g. a *user-oriented* scenario is the one where human users do small edits on models and visualize whole models concurrently. The scenario we address in this paper is an *application-oriented* one, where applications such as model transformations read only small portions of models and process them (e.g. to create new models or to generate different artefacts such as source code or documentation); a persistence solution that tackles such scenario must provide means to traverse specific parts of a model efficiently instead of fully loading it.

One approach for tackling scalability is to partition models via some modularization construct provided by the modeling language [6]. Instead of having to manage large models, modularization would allow to keep the models at a reasonable size. However, the complexity of large models makes it difficult to automatically partition them into fragments that are easily accessible [7] hence having a scalable model persistence solution would be mandatory. For example, source code models extracted from a legacy system being modernized may not be properly modularizable because of the complexity of their interconnections.

The XMI (XML Metadata Interchange) format [8] is normally used for the serialization (i.e. persistence) of models. When some operation (e.g. a model transformation) is performed on a model, the stored XMI file has to be parsed in order to build the model in memory as

This work is funded by the Spanish Ministry of Science (project TIN2009-11555) and Fundación Séneca (grant 14954/BPS/10).

an instance of its metamodel. For example, in the widely used Eclipse Modeling Framework (EMF) [9] the usual approach consists of a SAX parser that fully reads an XMI file and builds the entire model in memory at once. However, large models may not be fully kept in memory, causing the parser to overflow the client. Although XMI files support modularization through references between modules (i.e. files), requesting a single element from a referenced module would require its full load, so this solution does not scale. Therefore, as noted in [7], handling large models requires some mechanism that allows the client to load only the objects that will be used. This load on demand behaviour is a must when pursuing scalable model persistence.

To overcome the limitations of XMI-based persistence, model repositories [10] [11] are emerging as persistence solutions for large models, providing remote model access with advanced features such as concurrent access, transaction support and versioning; some available model repositories are discussed in Section 9. Currently, CDO [10] is the most mature repository for EMF; however, it does not scale properly as shown in Section 10.

Tool integration is another concern that arises when client applications access persisted models. The integration between a persistence solution and any client should be transparent, that is, it should conform to the standard model access interface defined by the considered modeling framework (e.g. the Resource interface of EMF). Moreover, a persistence solution that integrates transparently must not require any pre or post-processing on the (meta)models in order to load or store them, e.g. requiring source code generation for the persisted (meta)models [10][11].

In this paper we present Morsa, a model repository aimed at achieving scalability and transparent tool integration. The problem of scalability is tackled using load on demand and incremental save mechanisms supported by an object cache which is configurable with different policies. We discuss how these policies fit for common model traversals such as depth-first order and breadth-first order. The design of Morsa's data model is heavily inspired on the document-based NoSQL database paradigm (although it could be deployed over any kind of database). The NoSQL database paradigms are gaining popularity because of their approach to define highly-scalable databases with simple data architectures. The document-based paradigm consists of maps of key-value pairs (see Section 2.3) which provide a more natural persistence for models than object-relational mappings; for instance, a many-to-many relationship could be represented as a key-value pair instead of using intermediate tables as object-relational mappings do. We have dealt with the problem of transparent tool integration by implementing the EMF interface and by designing our load and store algorithms so that no pre or post-processing is required.

We contribute a prototype implementation for EMF [12] that uses a MongoDB [13] NoSQL backend and integrates transparently with client tools such as model transformation languages. A set of benchmarks has been executed to evaluate Morsa, demonstrating that it is capable of fully loading large models with a limited amount of memory. Moreover, it also exhibits better performance for EMF than the XMI file-based persistence and CDO [10].

A paper introducing a preliminary version of Morsa [14] was presented in the MODELS 2011 Conference. The current paper shows a change in the design of the repository and a more detailed description of its capabilities, including full and incremental model store, update and delete. The evaluation has also been extended with benchmarks that test the performance of these capabilities. Moreover, the background and related work of the former paper have also been enriched. For the sake of readability, the changes done from the previous version to the current one are explained separately in Section 11.1. Finally, a few recommendations for selecting persistence solutions are given.

The rest of the paper is structured as follows: Section 2 presents the concept of model persistence and some terminology about MDE and the NoSQL movement that ease the understanding of our proposal; Section 3 introduces the running example; Section 4 shows the architectural and data design of our repository; Sections 5, 6, and 7 explain how Morsa stores, loads, updates and deletes models, respectively; Section 8 discusses the integration and implementation of Morsa; Sections 9 and 10 comment the related work and the evaluation and finally Sections 11 and 12 shows the knowledge gained with our previous and current experience on model persistence and our conclusions and further work.

2 Background

This section defines some basic concepts related to the persistence of models for a better understanding of the approach presented in this paper. The concept of model persistence is introduced after a brief discussion of the representation of (meta)models as object graphs. Moreover, the NoSQL movement is introduced as an alternative to relational databases and object-relational mappings for model persistence.

2.1 Metamodeling

In MDE, the four-level metamodeling architecture [15] is normally used to explain the relationships between models, metamodels and meta-metamodels. A model represents some aspect of a particular domain and is described by a metamodel which establishes its structure.

A metamodel is a model that describes the concepts and relationships of a certain domain. A metamodel is

commonly defined by means of an object-oriented conceptual model expressed in a metamodeling language such as Ecore [9] or MOF [16]. A metamodeling language is described by a model called the meta-metamodel. Metamodeling languages provide four main constructs to express metamodels: *classes* (normally referred as metaclasses) for representing domain concepts, *attributes* for representing properties of a domain concept, *association relationships* between pairs of classes for representing connections between domain concepts and *generalizations* between child metaclasses and their parent metaclasses for representing specialization between domain concepts. We will use the term *structural feature* to refer to both attributes and relationships.

Figure 1 shows a metamodel that represents a simple object-oriented programming language with concepts such as module, class, feature, field, method and parameter, which are represented by metaclasses. A special OOModel metaclass has been introduced to aggregate all modules. All these metaclasses have a name attribute, shown only in the OOModel metaclass for the sake of readability. The example shows also several association relationships among the metaclasses (modules, classes, returnType, features, type) and a generalization between Feature, Method and Field. Figure 2 shows a model that represents an object-oriented program consisting of two modules, two classes, one method, one field and two method parameters.

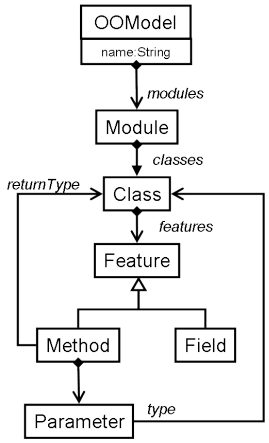


Fig. 1 Metamodel for a simple object-oriented programming language

An *instance-of* (or conformance) relationship is given between a model and its metamodel as well as between a metamodel and its meta-metamodel. The elements of a (meta)model are instances of (conform to) the metaclass of its (meta)metamodel. For example, object modelOne of Figure 2 is an instance of metaclass OOModel of Figure 1, which is in turn an instance of the element from the metamodeling language that represents metaclasses (e.g. EClass in the Ecore meta-metamodel). A (meta)model can be represented as a directed labelled graph whose

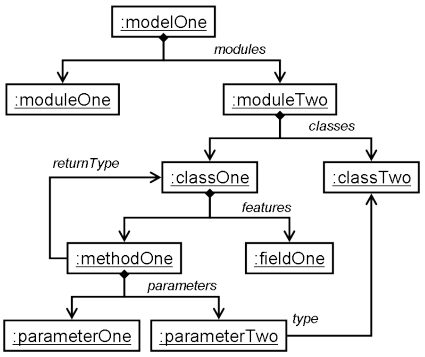


Fig. 2 Example model representing an instance of a simple object-oriented programming language

nodes are instances of metaclasses and whose arcs are determined by association relationships, being the labels on those arcs their association kinds (for models) or the name the associations (for metamodels).

Two kinds of association relationships can be established between metamodel metaclasses (and therefore between model elements): *containment* and *reference*. A reference relationship is a reference from a source metaclass (or model element) to a target metaclass (or model element). For instance, the relationship returnType between metaclasses Method and Class shown in Figure 1 is a reference; this relationship is also shown between instances classOne and methodOne in Figure 2. A containment relationship is a kind of part-of relationship (or aggregation) from a container element (i.e. a metaclass or model element) to a contained element. Such relationship has three properties:

- *Exclusive ownership*: the contained element cannot be part of more than one container element.
- *Dependency*: the lifetime of a contained element is the same as the one of its container element.
- *Transitivity*: if an element A is contained by an element B and B is also contained by another element C, then A is contained by C.

For instance, the relationship between metaclasses Module and Class in Figure 1 is a containment; this relationship is also shown between instances moduleTwo and classOne in Figure 2. Although containment relationships are not compulsory in all metamodeling languages, we assume their existence because it is the way EMF is designed and also because they can be semantically emulated by regular relationships.

The approach presented in this paper to persist models is based on the fact that models are object graphs, so some terminology defined for graphs which will be used throughout this paper is introduced below. Given an object (i.e. a model element):

- an *ancestor* is an object that transitively contains it;
- a *descendant* is an object transitively contained by it;

- a *child* is an object that is directly contained by it;
- a *parent* is an object that directly contains it;
- a *sibling* is an object that shares the same parent with it;
- its *breadth* is its position inside the list that contains it;
- its *depth* is the number of ancestors that contain it.

Moreover, a *subgraph* (i.e. model partition) is a graph whose nodes and arcs are a subset of a given graph and a *root object* is an object that has no ancestors. We illustrate the meaning of these concepts using the model on Figure 2, which shows the containment relationships between model elements. In this figure we can observe the following:

i. modelOne is the root object, so its depth is 0.
ii. methodOne's ancestors are classOne, moduleTwo and modelOne, so its depth is 3.
iii. classTwo is a sibling of classOne and its breadth is 2.
iv. classOne's parent (container) is moduleTwo.
v. classOne's children are methodOne and fieldOne.
vi. classOne's descendants are methodOne, fieldOne, parameterOne and parameterTwo. A subgraph could be formed by these objects.

2.2 Model persistence

Different approaches are used to permanently store models and metamodels. The three main persistence solutions are: i) XML serialization based on the XMI format, ii) relational databases through object-relational mappings such as Teneo[17] and, iii) at a higher abstraction, model repositories such as CDO [10].

Model persistence is a service normally provided by modeling frameworks (e.g. EMF). These modeling frameworks usually define persistence interfaces that allow client applications to access persisted models, e.g. the EMF Resource interface. These interfaces provide methods for the four basic operations involved in moving models between memory and persistence:

- *Load*: a model or a model partition is transferred from a persistence solution to the client's memory. It involves rebuilding a persisted object (sub)graph into a set of model elements. If the whole object graph is rebuilt, the model is fully loaded; otherwise, if only a subgraph (i.e. model partition) is rebuilt, the model is partially loaded.
- *Store*: a model or a model partition is transferred from the client's memory to a persistence solution. It involves representing an in-memory model in the format used by the persistence solution (e.g. relational tuples). If the whole object graph is stored at once, the model is fully stored; otherwise, if only a subgraph is stored, the model is incrementally stored.
- *Update*: a model or a model partition that is already persisted is modified in the client's memory and then transferred to the persistence solution. It involves modifying the already persisted objects to reflect the changes done by the client application. An update is usually done in an automatic fashion when a modified model or model partition is stored.
- *Delete*: a model or model partition is removed from the repository. Deletion may be performed automatically when a model or model partition is updated and some model elements have been removed from it.

These operations are needed when client applications access models and traverse them for different purposes. For example: a model-to-model transformation may look for a particular object that satisfies a given condition and then traverse all its descendants in order to generate a new target model element; a model-to-code transformation may simply traverse a whole model, processing each object once or twice, etc. Both transformations require loading a model, traversing it and, in the case of a model-to-model transformation, build model elements in memory. Because the loaded models may be very large, persistence support for partial load may be crucial for achieving scalability at the client. Incremental store is also very important because it can provide the client methods to discard already generated objects, freeing memory.

A persistence solution provides *transparent integration* when client applications may access it using the persistence interface defined by the corresponding modeling framework without any form of pre or post-processing, such as changing the models or metamodels to add persistence data or generating persistence-specific source code for the metamodels. For example, the XMI file-based persistence solution for EMF does not require generating metamodel-specific Java classes because it may use dynamic objects, which can be generically built at runtime. On the other hand, EMFStore [11] requires the modification of a metamodel in order to persist its instances.

2.3 The NoSQL movement

As mentioned in the introduction, we propose a model repository whose data model is heavily inspired on the document-based NoSQL databases. Two decades ago, new database applications that would require managing complex objects (e.g. geographic information or multimedia systems) evidenced the limitations of the relational model for the representation and processing of that sort of data. Then, new kinds of database management systems (DBMS) were defined, such as object-oriented and object-relational database systems [18].

More recently, database applications for domains such as searching text on the web or processing data streams

have again exposed that relational DBMSs are not adequate for the new user requirements and hardware characteristics (distribution, scalability, etc.) [19]. The number of applications for which the *"one size fits all"* approach of the commercial SQL solutions does not apply is increasing. This approach is too general to achieve certain degrees of scalability and performance and leads to an excessive deployment complexity from a design and architectural point of view [19].

The NoSQL [20] term is used to refer to different new database paradigms which are an alternative to the predominant relational DBMSs. Web applications such as social networks (e.g. Facebook), text searching (e.g. Google) and e-commerce (e.g. Amazon), which manage very large and complex data, are some examples of scenarios where different NoSQL databases have been successfully used. The main difference between NoSQL databases and relational databases is the set of properties they provide; while relational databases provide all the ACID (Atomicity, Consistency, Isolation and Durability) properties, NoSQL databases provide a subset of the CAP properties: *Consistency* (whenever a writer updates, all readers see the updated values), *Availability* (the system operates continuously even when parts of it crash) and *Partition tolerance* (the system copes with dynamic addition and removal of nodes).

The main flavours of NoSQL are the key-value stores, the document databases and the column-oriented ones. *Key-value stores* have a single map/dictionary that allows clients to put and request values per key. Key-value stores such as Dynamo [21] favor high scalability over consistency and omit rich querying and analytics features. *Document databases* such as MongoDB [13] and CouchDB [22] encapsulate key-value pairs in composite structures named documents, providing more complex and meaningful data than key-value stores without any document schema, thus eliminating the need of schema migration efforts. Finally, *column-oriented databases* such as Bigtable [23] store and process data by column instead of rows in a similar way as the analytics and business inteligence solutions.

The application of NoSQL databases to MDE provides a natural mapping between models and their persisted counterparts: as explained above, models can be seen as graphs, and some kinds of NoSQL databases such as the document-based ones are well-suited for representing graphs; on the other hand, the mapping from graphs to tables and rows used by relational databases and object-relational mappings is cumbersome, less natural and less readable. Morever, there are some features of the NoSQL databases that may be be beneficial to the persistence of models:

i. *Scalable*: as explained before, many MDE applications involve large models. Applications that involve large amounts of data representing object models scale better in NoSQL than in relational databases [20].
ii. *Schemaless*: having no schemas means having no restrictions to co-evolve metamodels and models. Relational repositories usually create database schemas for each stored metamodel, making their evolution more difficult and the conformance of existent models to the newer versions of their metamodels [10].
iii. *Accessible*: many NoSQL databases offer their data as JSON objects [24] through APIs that can be accessed via HTTP or REST calls. This provides additional opportunities to access models from web browsers, web services, etc. The integration of MDE and web-based technologies could lead to the storage of models in the Cloud [25].

3 Running example

A running example is used to illustrate the design of our approach. It is based on the reverse engineering case study of the Grabats 2009 contest [26]. This case study consisted in executing a particular query on five very large test models representing Java source code. The *JDTAST* metamodel that defines these models is composed of three packages: the Core package includes metaclasses that represent logical units such as projects, packages or types; the DOM package includes metaclasses for representing abstract syntax trees for Java source code, e.g. compilation units, methods, packages and type declarations, literals and annotations; finally, the PrimitiveTypes package includes metaclasses that represent Java primitive types such as String or Integer.

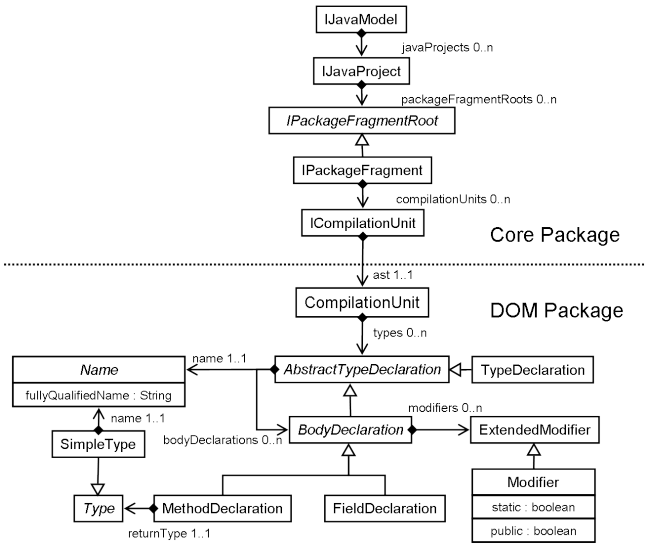


Fig. 3 Grabats 2009 contest JDTAST metamodel simplification

The query proposed in the case study consists in obtaining every class that declares a static public method

whose returning type is that same class. Figure 3 shows the subset of the *JDTAST* metamodel involved in this query. The information about Java modifiers and returning types is specified in the DOM package. However, there is no explicit reference from a method's returning type (Type object) to the declaration of that type (TypeDeclaration object); the matching between both objects must be done by their name. The query basically consists in the following steps for each TypeDeclaration object of the model:

1. Get the *fullyQualifiedName* of the Name object referenced by its *name* relationship.
2. Find at least one MethodDeclaration object referenced by its *bodyDeclarations* relationship that:
   - has a Type object referenced by the *returnType* relationship and
   - that Type object has a Name object referenced by the *name* relationship and
   - that Name object has a *fullyQualifiedName* attribute that matches the *fullyQualifiedName* obtained in step 1.

Of course, this query could be implemented in efficient ways that do not involve iteratively checking every TypeDeclaration object. We have chosen this running example for three reasons: (i) the test models are very large and capable of overloading the memory of a client application, which is one of the issues that we address; (ii) these models have been extracted from the source code of real applications and (iii) the test query is a realistic example of the kind of access done by client applications such as model transformations.

4 Design

In this paper we present Morsa, a model repository for managing large models. As commented in Section 1, Morsa has two main design goals: *transparent integration* and *scalability*. The goal of transparent integration requires an *architectural design* that allows client applications to use the repository without doing any specific modifications on the modeling artifacts or the source code, such as editing the (meta)models or using specific programming interfaces. The architectural design of our solution is described in Section 4.1.

The goal of scalability requires a *data design* that is loosely coupled enough to support the load and store of model partitions or single objects from a large model in an efficient way for the client. The data design of our solution is described in Section 4.2. Moreover, the architectural design is also involved in the goal of scalability since its components support the data design.

4.1 Architectural design

The architecture of Morsa consists of a client side and a server side, as shown in Figure 4. The client side is hosted on the client machine, i.e. the one that runs the client application and the server side is hosted on a remote machine, e.g. a dedicated server (although it may be the same machine).

The *client side* of Morsa supports integration through a driver (MorsaDriver) that implements the modeling framework persistence interface, allowing client applications to manipulate models in a standard way. Since Morsa is aimed at manipulating large models, a load on demand mechanism has been designed to provide clients with efficient partial load of large models, achieving scalability [7]. This mechanism relies on an object cache (ObjectCache) that holds loaded model elements in order to reduce database queries and manage memory usage; it is managed by a configurable cache replacement policy (CachePolicy) that decides whether the cache is full or not and which objects must be unloaded from the client memory if needed. The client side communicates with the server side using a backend adapter (MorsaBackend) that abstracts it from the actual database. An encoder (MorsaEncoder) is used to create and manipulate repository objects and backend queries.

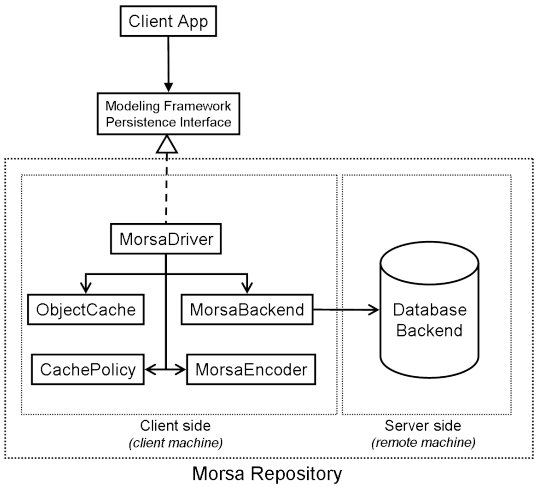


Fig. 4 Architecture of the Morsa repository

On the *server side* of Morsa, a database backend provides the actual storage of models. Thanks to the MorsaBackend component, which is an adapter, any kind of database can be used for persisting models. Moreover, the data design of Morsa has been devised having in mind a NoSQL document database, so the mapping between the client side and the server side for such databases is natural and almost direct. Mappings between Morsa and other databases can be applied, but their implementation could be less direct and hence less efficient.

4.2 Data design

On the *client side* of Morsa, a data model has been designed to represent the objects stored in the repository in a way that provides independence from the actual database backend.

As explained in Section 2.1, a model can be seen as a graph whose nodes are the model elements and whose arcs are the relationships among them. A model is represented in Morsa as a collection of MorsaObjects connected through MorsaReferences. Such a collection is called MorsaCollection and has an identifier (e.g. the URL of the (meta)model in EMF); MorsaCollections can also represent model partitions (i.e. a subgraphs). Since a metamodel can also be seen as a model that conforms to a meta-metamodel, the representation of both models and metamodels is homogeneous. Figure 5 shows an example of the representation of a model in the Morsa repository. On the left side, an instance of the *JDTAST* metamodel (see Figure 3) is shown; on the right side, a set of MorsaObjects represents both the model and the part of the metamodel referenced by the model elements. Solid arrows represent relationships between elements and dashed arrows represent *instanceOf* relationships between objects and their metaclasses.

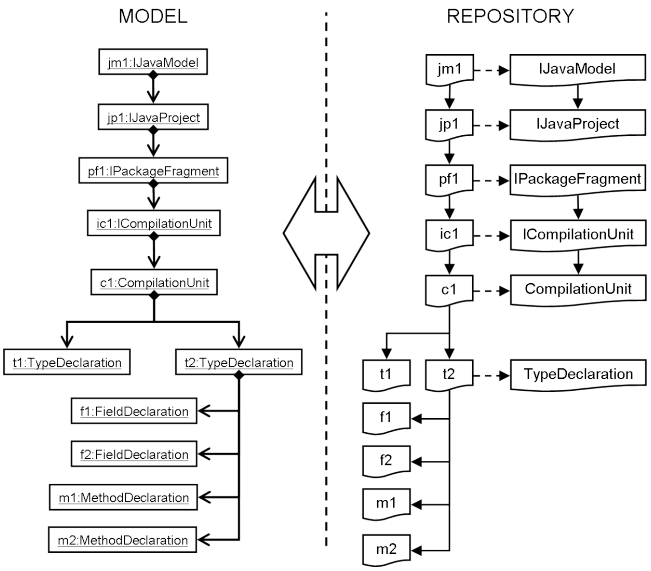


Fig. 5 Example of repository persistency for the running example

Each MorsaObject represents a model element and is composed of a set of key-value pairs that encode the structural features of that element. The key is the name of the structural feature and the value may be a primitive value if the feature is an attribute or a MorsaReference if the feature is a relationship; multi-valued attributes (e.g. collections in Ecore) are represented as collections of values (e.g. arrays in MongoDB). A MorsaObject also contains a descriptor of metadata used for identification, querying and optimization. This descriptor is also encoded as a set of key-value pairs. For a given MorsaObject representing a model element, its descriptor specifies the following features:

i. *MorsaID*: repository-unique, backend-dependent identification (e.g. a UUID).
ii. *Metatype*: MorsaReference to the MorsaObject representing the metaclass from which the model element has been instantiated.
iii. *Container*: MorsaReference to the MorsaObject representing the model element that contains this one (see Section 2.1).
iv. *Ancestors*: a list of MorsaReferences to the MorsaObjects that represent the ancestors of the model element (see Section 2.1).
v. *Breadth*: the position of the model element inside its containing relationship.
vi. *Depth*: the number of ancestors of the model element.

The *MorsaID* is a key feature because it allows the ObjectCache to uniquely identify loaded objects in the client side. The *Metatype* feature allows the client side to infer the objects' structural features. *Breadth*, *Depth*, *Ancestors* and *Container* features represent the structure of the object graph and are used for partial loading as explained later in Section 6.1.2.

A MorsaReference is composed of, at least, the *MorsaID* of the referenced element and the identifier of the MorsaCollection that holds it, i.e. its containing model; depending on the implementation of MorsaBackend being used, it may contain additional information.

Figure 6 shows the internal structure of the MorsaObjects that represent the elements t2 and TypeDeclaration of the model and metamodel respectively shown in Figure 5. The *MorsaID*, *Container*, *Ancestors* and *Metatype* values for this example have been simplified to the name of the object for the sake of readibility. Note that the *Breadth* feature of t2 has a value of 2 because t2 is located on the second position of the *typeDeclarations* relationship of the c1 CompilationUnit; also note that its *Metatype* feature references the MorsaObject that corresponds to the TypeDeclaration metaclass.

A special MorsaCollection called the *index collection* holds the *index object*, which is a singleton MorsaObject whose keys are the identifiers of the (meta) models stored in the repository (e.g. URIs for EMF) and whose values are MorsaReferences to the root objects of those (meta)models. For each metamodel package a MorsaCollection is created. The index object is used by the MorsaDriver to access (meta)models. Figure 7 shows how the index object references the root objects of the three packages (Core, DOM and PrimitiveTypes) defined in the metamodel of Figure 3. Note that the index object points to the jm1 model element, which is the root of the javaModel1 model, described in Figure 5.

MorsaObjects provide the client side a way to transfer model elements from/to a Morsa repository that is

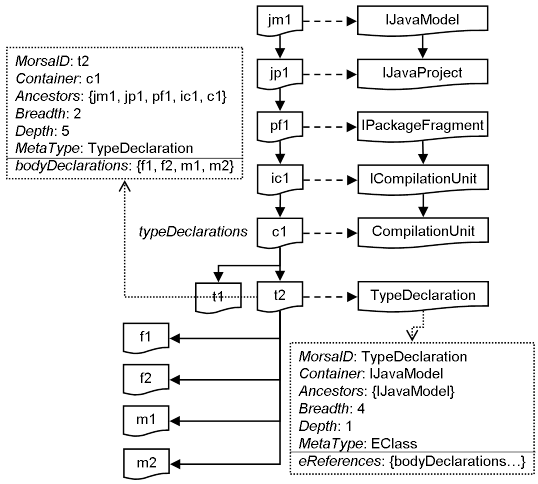


Fig. 6 Internal structure of two MorsaObjects

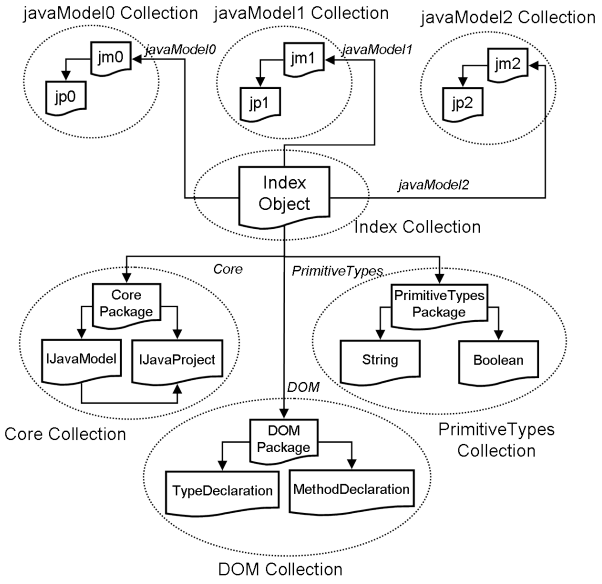


Fig. 7 Collections contained by the repository for the *JDTAST* metamodel packages and three sample models

backend-independent. Figure 8 illustrates the components of the client side architecture that interact in order to load a model element; for the sake of readability, some components and operations such as the ObjectCache and its related mechanisms have been omitted. The following steps are executed:

i. The client application requests a model element.
ii. The MorsaDriver (which is accessed transparently by the client application since it implements the modeling framework persistence interface) passes the request to the MorsaBackend.
iii. The MorsaBackend encodes the request using the MorsaEncoder and sends it to the database.
iv. The object returned by the database is decoded by the MorsaEncoder into a MorsaObject by request of the MorsaBackend, who returns it to the MorsaDriver.
v. The MorsaDriver transforms the MorsaObject into an object that conforms to the modeling framework (e.g. EObject for EMF), which is returned to the client application.

The interaction for storing a model element is very similar. The actual processes of storing and loading model elements are more complex and will be explained in Sections 5 and 6, respectively. The following sections describe the algorithms for the store, load and update operations on models and model partitions, as defined in Section 2.2. These algorithms are explained in terms of the presented data model, so transfer of objects from/to the database is obviated for the sake of simplicity and the persistence backend is seen as a set of MorsaObjects rather than a database.

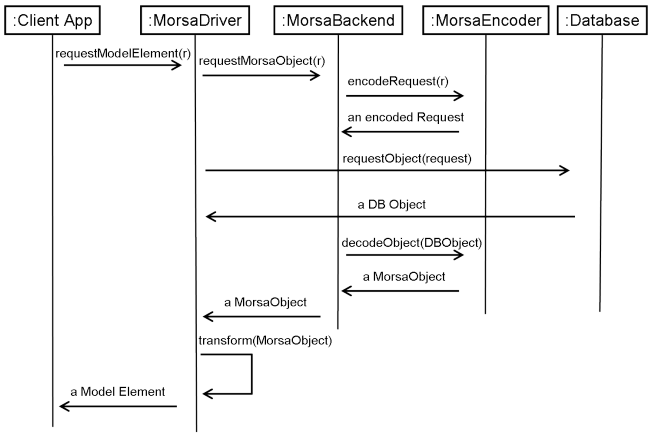


Fig. 8 Simplification of the interaction between client side components for loading a model element

5 Model storage

When a model is created in client memory from scratch, for example by means of a model transformation, it has to be stored in the repository to become persistent. *Model store* is the operation of storing a model in the repository for the first time or fully replacing a model that is already persistent. Storing a model may be seen as a simple task: basically, the MorsaDriver transforms the elements of the input model into MorsaObjects and saves them into the persistence backend. If the input model is too large to store it in one single operation due to network latency or to be kept it in the client's memory, it may be stored in several steps. We call the simplest scenario *full store*; the other scenario is called *incremental store*. Metamodels are stored prior to its conforming models using full store if they are not already persisted in the repository.

5.1 Full store

The *full store* algorithm is executed when a model is stored for the first time or when it is fully replaced. This algorithm uses a fixed-size queue, namely *pending object queue*, to optimize the access to the database backend. When an input model is traversed to generate the persistent model, the created MorsaObjects are temporally stored in the pending object queue rather than sent to the database backend, hence the queue acts as a buffer. By sending a batch of stores instead of many individual ones, the communication between the client side and the server side is optimized, avoiding overheads. After being sent to the persistence backend, the MorsaObjects are discarded from the client memory.

The first step of the algorithm is to create the new MorsaCollection that will represent the stored model in the repository. Then, the algorithm traverses the whole model in a depth-first order, executing the following steps for each model element:

1. A MorsaObject is created, storing all the feature values of the model element:
   (a) Attributes are encoded by the MorsaEncoder as primitive type values.
   (b) Relationships are encoded by the MorsaEncoder as MorsaReferences.
   (c) References to model elements that have not been already stored imply the creation of new *MorsaID*s that will be assigned to those model elements at the time they are stored.
   (d) The descriptor of the model element (see Section 4.2) is calculated and encoded. If the model element does not have any corresponding *MorsaID*, a new one is created and assigned to it.
2. The newly created MorsaObject is added to the *pending object queue*. If the queue is full or if the last model element has been traversed, all its MorsaObjects are sent to the persistence backend in its own representation.

5.2 Incremental store

When a model is stored for the first time or when it is fully replaced but is too big to be kept in the client's memory or to be stored in a single operation, the *incremental store* algorithm is used. A scenario for incremental store could be one where a client extracts objects from an external resource in a streaming fashion and stores them in several steps, i.e. every step stores a model partition. The incremental store algorithm consists in executing the already described full store algorithm for every model partition that has to be stored, but with two differences: (i) only one MorsaCollection is created for all the model partitions (since they all belong to the same model) and (ii) every time a model partition is saved, all the objects that represent it are unloaded. To *unload* an object is to remove it from memory, making room for the objects that represent the next model partition. The process of unloading will be explained in Section 6. Morsa keeps track of the already processed objects using a *save cache* that maps them to their MorsaIDs. When the last model partition is stored, this map is deleted.

Since relationships between objects can be stored in the repository prior to the referenced objects, the incremental store scenario may lead to dangling references if the process is stopped before completion. To solve this, Morsa provides a special operation that eliminates all references to objects that have not been actually stored in the repository. There are two possible scenarios, depending on the connection between the driver and the repository: on the one hand, if the driver has been connected to the repository over all the incremental store process and it still is, it calculates the difference between the save cache (i.e. the model elements that have a *MorsaID* assigned to them) and the model elements that have been actually stored in the repository and then removes or updates all the stored MorsaReferences that reference them; on the other hand, the calculation is done traversing the whole persisted model. Since such updates of the repository are very expensive, they are natively executed at the database where possible (e.g. using JavaScript server-side functions in MongoDB).

6 Model loading

This section is dedicated to the load operation on models, as described in Section 2.2 First, the two diferent scenarios that we have identified for model loading will be described; then, the load on demand algorithm will be explained and finally the cache management and replacement policies that run on the client side will be described.

6.1 Loading scenarios

Since our approach is intended to manipulate large models, two scenarios have been considered: *full load* and *load on demand*. These scenarios are explained in detail below. The load on demand scenario has been tackled using an *object cache* managed by a *cache replacement policy*. Metamodels are always fully loaded and kept in memory for efficiency reasons: they are relatively small compared to models and it is worth loading them once instead of accessing the persistence backend every time a metaclass is needed. Each object is identified in the persistence backend by its *MorsaID* feature. A mapping between loaded objects and their *MorsaID*s is held by the object cache (ObjectCache) in order to know which objects have been loaded, preventing the driver to load

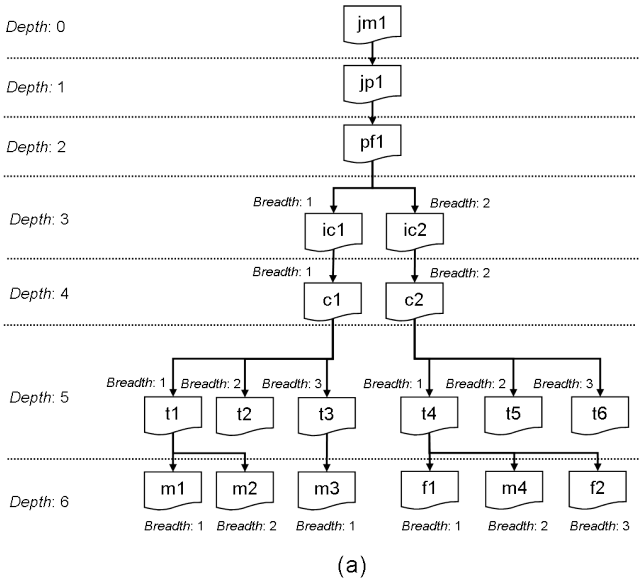
them again. The selection and configuration of each scenario is done by the client application by parameterization of the MorsaDriver; this implies gathering as most information as possible about the access pattern that is going to be performed.

6.1.1 Full load Consider a small or medium-sized model that can be kept in memory by a client application. If the whole model is going to be traversed, it would be a good idea to load it once, hence saving communication time with the persistence backend. We call this scenario *full load* and this is the way EMF works when loading XMI files. We aim at supporting full load with the least memory and time overhead possible. The Morsa full load algorithm works simply by fetching all the MorsaObjects of a model following its containment relationships. A new model element is created in the client memory for every MorsaObject, filling its features with the values stored in that MorsaObject.

6.1.2 Load on demand Consider a model that is too large to be kept in memory by a client application; consider also a model that can be kept in memory but only a part of it is going to be traversed. An efficient solution for loading models in both cases would be to load only the necessary objects as they are needed and then unload the ones that eventually become unnecessary to save client memory. This scenario is called *load on demand*. We define two load on demand strategies: single load on demand and partial load on demand.

A *single load on demand* algorithm fetches objects from the database one by one. This behavior is preferred when the objects that need to be accessed are not closely related (i.e, they are not directly referenced by relationships) and memory efficiency is more important than network performance, that is, when the round-trip time of fetching objects from the persistence backend is not relevant. The resulting cache will be populated only with the traversed objects.

On the other hand, a *partial load on demand* algorithm fetches an object subgraph from the persistence backend starting from a given root object. The structure of the subgraph to be fetched is customizable: given a requested root object, its subgraph contains all its descendants within a certain depth and breadth values. For example, consider that in the model shown in Figure 9(a) objects jm1 and jp1 have already been loaded and pf1 is requested with a maximum subgraph depth of 4 and maximum subgraph breadth of 2. Because the *Depth* feature value of pf1 is 2, the maximum depth will be 6. Objects pf1, ic1, ic2, c1, c2, t1, t2, t4, t5, m1, m2, f1 and m4 will be included in the subgraph, but t3, f2 and t6 will not, because their *Breadth* feature value is 3 (greater than 2). Note that m3 has a depth of 6 and a breadth of 1, but since its parent t3 is not included in the subgraph, it doesn't get loaded either. This behavior is preferred when all the objects that are related to an object will be traversed soon and memory efficiency is less important than network performance, that is, when the round-trip time of fetching objects from the persistence backend is critical. The resulting cache will be populated with the objects that have been traversed and those expected to be traversed in the near future, as shown in Figure 9 (c). For the sake of readability, the *MorsaID*s shown in this figure are the names of the corresponding objects. This is a simple form of prefetching that tries to take advantage of spatial locality.



Object	MorsaID
jm1	*jm1*
jp1	*jp1*
empty	
empty	
empty	
empty	
empty	

(b)

Object	MorsaID
Jm1	*jm1*
Jp1	*jp1*
pf1	*pf1*
ic1	*Ic1*
ic2	*ic2*
c1	*c1*
c2	*c2*
t1 (proxy)	*t1*
t2 (proxy)	*T2*
t3 (proxy)	*T3*
t4 (proxy)	*T4*
t5 (proxy)	*T5*
t6 (proxy)	*T6*

(c)

Fig. 9 Partial load on demand in the running example: a) model b) object cache before partial load c) object cache after partial load

6.2 Load on demand algorithm

The load on demand algorithm is triggered whenever a model element that is not in the client's memory (i.e. in the ObjectCache) is requested; this can be done by explicit request from the client application or by implicit

request when a relationship is traversed and the referenced element is not in the client's memory. Our load on demand algorithm work as follows:

1. A model element is requested.
2. The MorsaDriver requests the fetching of the corresponding MorsaObject to the MorsaBackend.
3. A new model element is created, filling its attributes with the values stored in the MorsaObject and its relationships with *proxies* that allow the load on demand of the referenced model elements. These proxies are special objects that have the same structure as model elements, but hold no feature values. Instead, they hold a URI containing a MorsaReference that allows their resolution by the repository. When a proxy is resolved, it becomes a model element with all its feature values filled. In EMF, the idea of proxies is used to represent cross-resource references.
4. The new model element and its proxies are stored in the ObjectCache, mapping them to their corresponding *MorsaID*s.
   (a) If single load on demand is used, go to step 5.
   (b) If partial load on demand is used, a request is sent to the MorsaBackend to get all the objects of the defined subgraph. The MorsaBackend uses the *Ancestors* feature to calculate which objects are descendants of the requested one and then to filter the results using their *Depth* and *Breadth* attributes. Each one of these objects is then loaded executing the steps 1 to 3 of this algorithm.
5. If the cache becomes overloaded, some objects of the cache are unloaded as explained in the next section.
6. The new model element is returned to the client application, which can use it as a regular element.

6.3 Cache management

The object cache holds the objects that have been loaded from the repository for three purposes: (i) memory management, (ii) object identification and (iii) prevent loading objects that have been already loaded. The object cache is parameterized by a size limit and a replacement policy; both parameters are set by the client application, which passes them to the Morsa driver.

The *size limit* is the amount of objects that can be held by the cache; however, this limit is soft because some modeling frameworks such as EMF require model elements to have their relationships filled, that is, their values must be fetched in the form of proxies or actual model elements. For example, consider again the model in Figure 9 (a): elements jm1 and jp1 have already been loaded and are stored in the cache, which has a maxium size of 7 objects, as shown in Figure 9 (b). The partial load on demand of pf1 is requested with a subgraph depth of 2 and a subgraph breadth of 2, meaning that pf1, ic1, ic2, c1 and c2 will be loaded and stored in the cache. However, since the modeling framework requires the direct relationships of every object to be fully filled, when c1 and c2 are loaded, their children t1..t6 must be fetched as proxies and stored in the cache, causing it to be overloaded to a size of 13 model elements as shown in Figure 9 (c).

Whenever the cache becomes overloaded, the exceeding model elements must be *unloaded*. A *cache replacement policy* algorithm selects the elements to be unloaded from the client memory. Unloading an element implies downgrading it to a proxy, i.e. unsetting all its features. A proxy requires less memory than an actual model element and it can be discarded by the underlying language if it is not referenced by any other object.

When a modified model element is unloaded, all its changes must be persisted in some way to prevent losing them. Storing the element in the MorsaCollection that corresponds to its model would not be appropriate since the unloading mechanism is not triggered by the client, who sees the model as it is entirely in-memory and may want to persist changes only at a certain moment. Because of this, modified elements are stored as MorsaObjects in a special MorsaCollection called the *sketch collection*; this collection is also persistent in the repository. Whenever a model element is requested, the sketch collection must be examined in the first place to check if that element has been modified and unloaded previously. The presence of modified model elements in the sketch collection partly invalidates the representation of the graph structure of the model built by the *Ancestors* feature values since modified ancestors and descendants are not updated in the persistence backend. A partial load on demand of a subgraph that contains modified ancestors or descendants would ignore objects that are contained in the subgraphs of the modified ones. Elements are removed from the sketch collection when they are loaded into memory or when the model is explicitly stored by the client. The definition of a modified model element is explained in Section 7.

6.4 Cache replacement policies

A *cache replacement policy* is encapsulated in a CachePolicy object. We have considered four cache replacement policies:

i. A FIFO (First In-First Out) policy would unload the oldest objects of the cache. This policy is useful when a model is traversed in depth-first order, but only if the cache can hold the average depth of the model. On the contrary, it would cause objects to be unloaded after being traversed and then loaded again when requested for traversal.
ii. A LIFO (Last In-First Out) policy would unload the most recent objects of the cache. This policy is useful when a model is traversed in breadth-first order, but only if the cache can hold the average breadth of the model. Both the LIFO and the FIFO policies

calculate the size of the subgraph directly contained by the object that caused the cache overload and unload that many objects. In the example of Figure 9, a LIFO policy would unload the objects t1...t6, while a FIFO policy would unload jm1, jp1, pf1, ic1, ic2, and c1.

iii. A LRU (Least Recently Used) policy would unload the least used objects of the cache. The LIFO, FIFO and LRU policies are well known in the area of operating systems. A LRU policy would be equivalent to a FIFO one for depth-first and breadth-first traversals.

iv. A LPF (Largest Partition First) policy would unload all the elements that conform the largest model partition contained by the cache. This is a conservative solution that is useful when a model is traversed in no specific order. It does not consider if the selected elements are going to be traversed so it may lead to multiple loads of the same objects. This policy unloads at least an amount of objects proportional to the maximum size of the cache.

The choice of which cache replacement policy is used is currently made by the end user. However, it could be automatically made by the MorsaDriver by analysis of (meta)models and access patterns (i.e. prefetching).

7 Model updating and deleting

When a model is loaded (fully or partially), modified and then stored back, an *update* operation takes place. As mentioned in Section 2.2, an update is a store operation where the stored elements have been modified. Therefore, the update algorithm is an extension of the one described in Section 5.1 for the full store: model elements are traversed in the same way, but modified and deleted objects must be treated differently. Another scenario that involves model update is when a model is generated in several steps: as each step is finished, the generated subgraph is no longer necessary and hence it can be unloaded from the client's memory; some of the unloaded objects may be loaded back and modified to perform further steps.

We consider that a model element is *modified* if any of its feature values has changed or if it has been moved from one parent to another. We classify modified elements in three categories: *modified elements* are model elements whose feature values have changed, *modified parents* are model elements whose containment relationships have changed and *modified children* are model elements that have been moved from one parent to another. Note that while a modified parent is a special kind of modified element, a modified child may not have any of its feature values changed. Modified elements are updated to the repository. Modified parents must update also the *Breadth* feature values of their children because a new child has been added or removed. Finally, modified children must update their *Container* and *Ancestors* feature values because they are now in a different part of the object graph, and also update the *Ancestors* feature values of their descendants in order to faithfully reflect the new structure of the object graph. Modified elements, ancestors and children all retain their original *MorsaID*s.

A model element is *deleted* when it is not contained by any other model element and it is not identified as a root of the model by the modeling framework (i.e. in EMF, roots elements are the ones directly contained by a Resource object). A deleted object is also a modified child but since it is not going to be persisted anymore, there is no need to update its descendants. Because containment relationships are exclusive, when an element is deleted its children become deleted and so on, deleting the whole subgraph formed by the descendants of the deleted element. While other behaviors may be performed (e.g. moving the descendants of the deleted element to their nearest ancestor), we have decided to implement the semantics defined on Ecore [9] and MOF [16], which are the most widely used metamodeling languages.

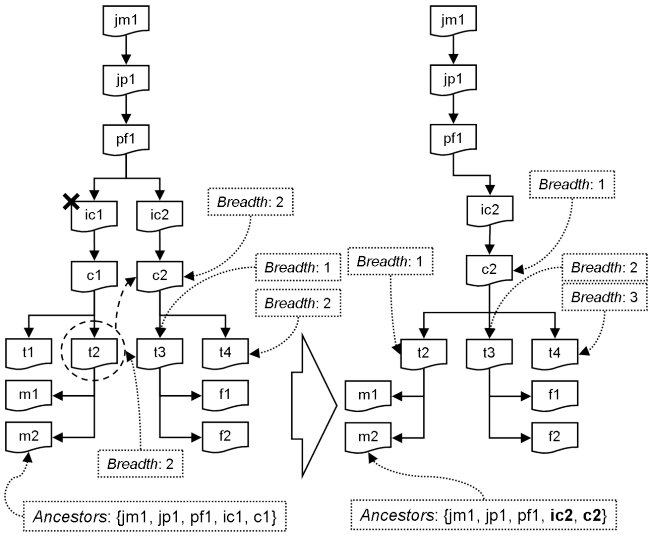


Fig. 10 Modifications and deletions over the running example

Figure 10 shows en example of update; modifications done on the source model (left side) are: 1) TypeDeclaration t2 is moved from CompilationUnit c1 to CompilationUnit c2 and 2) ICompilationUnit ic1 is deleted. Therefore, t2 is a modified child, because it has been moved from one parent to another; c2 and pf1 are modified parents, because a child has been removed and added, respectively; t1 and c1 will also be deleted since they no longer have any parent and they are not root objects. The result of the update can be seen in Figure 10 (right side): the *Ancestors* feature value of MethodDeclaration m2 has changed, replacing ic1 and c1 with ic2 and c2, and the *Breadth* feature values of ic2, c2, t2, t3 and t4

have also changed to faithfully represent the new object graph structure.

Deleting an entire model in Morsa is very simple: the MorsaDriver requests the MorsaBackend the removal of the MorsaCollection that holds the model. Depending on the underlying database backend, this could be implemented as a table drop (relational), collection drop (NoSQL), etc. Dangling references from other models to deleted objects could be eliminated using the special operation commented in Section 5.2. When some model elements are deleted rather than the entire model, a model update is performed instead.

8 Integration and implementation

Morsa is intended to be integrated with modeling frameworks and their applications. Our current prototype is integrated with EMF [9]. A transparent way of achieving integration is to design the MorsaDriver as an implementation of the persistence interface of the modeling framework (Resource in EMF). Persisting a model in Morsa is done without any preprocessing, since there is no need of generating model-specific classes, modifying metamodels or registering them into the persistence solution, as opposed to other approaches [10][11][27]. Metamodels are seamlessly persisted if they are not already in the database. Additional information for persistence configuration can optionally be passed to the driver; Morsa uses the standard parameters of the EMF load and save methods to pass this configuration information.

Morsa supports both *dynamic* and *generated* EMF. A dynamic model element is generated at runtime using EMF dynamic objects (DynamicEObjectImpl instances) which use reflection to generically instantiate metaclasses. On the other hand, a generated model element is an instance of a metamodel-specific class that has been explicitly generated through an EMF generator model. Dynamic objects are preferred for tool integration since they do not require code generation. Other approaches [10] support only generated model objects reimplementing part of the EMF framework to handle persistency. Morsa uses a subclass from DynamicEObjectImpl called MorsaEObject that handles proxy resolution automatically: if a feature of a proxy is accessed by a client (see Section 6.1.2), the proxy itself requests its own resolution to the MorsaDriver.

We have developed a prototype that exhibits all the features described previously: EMF integration, full load, single and partial load on demand, cache replacement policies, full and incremental store, update and deletion. Its integration with EMF includes all the methods defined in the Resource interface.

Since the data design is heavily inspired on the document database paradigm as explained in Section 4.2, we wanted to have our prototype implemented for such a database, although the architecture of Morsa can be implemented for other database paradigms such as the relational or other NoSQL approaches. The choice for the database engine was between CouchDB [22] and MongoDB [13], since they are the most relevant document-based NoSQL databases. On the one hand, CouchDB consists of a flat address space of JSON [24] documents that can be selected and aggregated using JavaScript in a Map/Reduce [28] manner to build views which also get indexed. It supports multiple concurrent versions of the same document, detecting conflicts among them. On the other hand, MongoDB allows grouping documents in collections and provides multi-key indexing and sophisticated querying using a dedicated language or JavaScript Map/Reduce functions. MongoDB stores BSON [29] documents, which are different from the ones of CouchDB as they can include nested documents, providing a more objectual data schema. A MongoDB database can also be automatically sharded to distributed database servers. We have chosen MongoDB as the database engine for our prototype mainly because of its dynamic queries (as opposed to the static views of CouchDB), its server-side JavaScript programming and its lightweight BSON support for communicating objects. BSON provides fast and bandwith-efficient object transfer between the client and the database.

Being our data model very close to the one of MongoDB, most of the concepts supporting Morsa can be directly mapped to MongoDB: MorsaObjects are mapped to MongoDB DBObjects (i.e., BSON objects), MorsaCollections are mapped to DBCollections (i.e, collections of documents) and *MorsaID*s are represented as ObjectIds. This allows for an easy and natural implementation that performs efficiently as shown in Section 10.

9 Related work

Model persistence is not a novel research field. As the interest in MDE grows many approaches have been proposed to solve this problem. The standard EMF solution is to persist models in XMI resources. Although there are other approaches such as using binary indexed files [30], model repositories are the most appropriate persistence solutions for MDE. A *repository* is a persistence solution remotely accessible by users and tools. Repositories usually rely on databases and provide additional features such as transactions and versioning. There are many EMF model repositories available today, being the most mature ones CDO [10], ModelBus [27] and EMF-Store [11].

The *ModelBus* repository is a web service application that manages an embedded Subversion [31] engine which implements the actual repository; however, Subversion is not designed to be integrated in client applications that access parts of persisted elements, i.e. it does not support load on demand. There have been attempts to make model access scalable in ModelBus [32]; however,

the official release does not implement them. *EMFStore* implements a different architecture but shares the same philosophy as Subversion: models are fully loaded and stored by human clients using a GUI. This solution does not scale and it is best suited for design environments.

Currently *Connected Data Objects* (CDO) is the only model repository that is capable of managing large models using load on demand; it is also the most widely used. CDO provides a rough version control system and EMF integration through its EMF Resource implementation, CDOResource; however, its integration is not transparent. First of all, although its documentation states that it can handle dynamic model objects, we could not make it work with them. Moreover, CDO requires metamodels to be pre-processed in order to persist their instances. One kind of pre-processing is to generate the Java model classes of a metamodel. This allows CDO to work with *legacy objects*. The other kind is to generate CDO-aware model classes from a generator model. This allows CDO to work with *native objects*. The main difference between legacy and native objects is that legacy objects cannot be demand-loaded or unloaded, having a huge impact on performance as will be shown in the next section. Native objects are unloaded from a CDO client when its memory becomes full using a *soft reference* approach, i.e. an object is removed by the garbage collector when no other object refers to it with a reference that is not soft.

MongoEMF [33] is a MongoDB-based model repository for EMF. It provides simple queries and transparent integration with the modeling framework. However, it only manages scalability on the client through model partitioning using cross-resource references, an EMF mechanism designed to provide simple load on demand through proxies. Our approach can easily simulate cross-resource references using references between elements in different MorsaCollections.

Some graph-based formalisms have been proposed to represent models and metamodels in a uniform way, such as [34], [35], [36] and [37]. The fist work is used for representing data models in the World Wide Web, while the other three implement multi-level modeling: [35] represents models as graphs in a mathematical fashion that has been also applied to model querying [38], while [36] and [37] use *clabjects* [39] for equally representing classes and objects.

A first approach on client scalability and transparent integration was presented in [40]. This repository also used load on demand to deal with the problem of client scalability and EMF integration to deal with transparent integration. Its data design resembled the preliminary version of Morsa [14], but its implementation on a relational database (MySQL) and its focus on version control delivered poor performance. However, it served as an inspiration for Morsa and made us put into consideration the possibility of using a NoSQL database backend.

There are other domains where large and complex data needs to be accessed; for example, many solutions have been proposed for managing large and complex ontologies, such as creating higher-level descriptions [41] which are similar to database views. Client scalability has been also tackled in the field of object-relational mappings, proposing prefetching mechanisms that load subgraphs that will be used by the client application [42][43]. Object caching has also been a subject of study in the field of object databases, with mathematical approaches to optimizing cache coherence, replacement and invalidation [44][45]. Our approach could benefit from this research to improve caching and prefetching with adaptive mechanisms.

10 Evaluation

As stated in the previous section, CDO is the most widely used model repository, so the evaluation consisted in executing a set of benchmarks with three different persistence solutions (Morsa, CDO and the standard EMF XMI parser) and comparing their performance results. The same set of test models has been used in all the benchmarks.

10.1 Test models

We have considered the models proposed in the Grabats 2009 contest [26]. They conform to the *JDTAST* metamodel explained in Section 3. There are five models, from Set0 to Set4, each one containing its predecessor. Table 1 shows the size of the XMI file corresponding to each model, the number of Java classes represented, the number of model elements contained and the size of the number of objects that satisfy the test query.

Name	XMI Size	Java classes	Model Elements	Result size
Set0	8.8MB	14	70447	1
Set1	27MB	40	198466	2
Set2	283MB	1605	2082841	41
Set3	598MB	5796	4852855	155
Set4	646MB	5984	4961779	164

Table 1 Test models

10.2 Test benchmarks

We have built a different benchmark for each of the four basic operations defined in Section 2.2: store, load, update and delete. In addition to those, a benchmark for the Grabats 2009 contest query has also been implemented:

i. The *model store* benchmark consists in storing the set of test models into each of the solutions. Full store

is executed over every solution and incremental store is executed over Morsa, being the only solution that supports it. Each test model is loaded from its XMI file in the first place and then stored in each solution.

ii. The *model load* benchmark consists in loading the set of test models from each solution. Full load is executed over every solution and load on demand is executed over CDO and Morsa. Both kinds of load consist in traversing the whole models in a depth-first order and in a breadth-first order.

iii. The *model update* benchmark consists in executing the Grabats 2009 test query described in Section 3 for each test model and then switching the container objects of the first and the last results, deleting the middle result and finally updating the model on each solution. If only one object is returned by the query, it is deleted; if only two objects are obtained by the query, their containers are switched and no object is deleted.

iv. The *model delete* benchmark consists in deleting the test models from each solution. Since the deletion of a XMI model does not imply any XMI processing but just a file deletion, it has not been considered.

v. The *model query* benchmark consists in executing the Grabats 2009 test query described in Section 3 for each test model. Since XMI does not support load on demand, the whole models must be loaded prior to query them. In CDO and Morsa, a simple method that fetches all the TypeDeclaration objects is executed and then the results are traversed to check whether they are eligible or not.

10.3 Results

Each benchmark has been executed using the EMF XMI loading facility, a CDO repository in legacy and native mode and a Morsa repository using single and partial load on demand. Both repositories have been configured to achieve best speed or least memory footprint, depending on the test; their configuration parameters have been fine-tuned based on their documentation and our empirical experience. All tests have been executed under a Intel Core i7 2600 PC at 3.70GHz with 8GB of physical RAM running 64-bit Windows 7 Professional Sp1 and JVM 1.6.0. CDO 4.0 is configured using DBStore over a dedicated MySQL 5.0.51b database. Morsa has been deployed over a MongoDB 1.8.2 database. Memory is measured in MegaBytes and time is measured in seconds.

10.3.1 Model store Table 2 shows the results of the model store benchmark. The incremental store scenario has been tested with incremental store as described in Section 5.2 (*Inc* mode) and also as described in Section 7 (*Inc by update*), i.e. taking the source model, partitioning it and storing each partition separately updating the root of the model in each step. We were not able to either incrementally store the test models on CDO or fully store the Set3 and Set4 models because even with the maximum available memory for both the server and the client, a timeout exception was always thrown.

XMI is obviously the fastest solution and the one consuming the least memory by far because it does not involve either network communication or object marshalling. In addition, the test models have been loaded from the XMI files in order to store them for CDO and Morsa, which implies a memory overhead that has been reduced as much as possible using incremental save in Morsa. In a full store scenario, CDO performs better in memory but worse in time (except for Set2). However, using incremental store Morsa consumes much fewer memory, which is comparable to that used by XMI (but aprox. 100 times slower) and Morsa is faster than CDO and uses less memory for the Set2 model when using incremental store by update,.

10.3.2 Model load Table 3 shows the results of the model load benchmark. Again, XMI is the fastest. For the best speed test case, a full load has been executed over CDO and Morsa, showing that our repository is aprox. 40% faster. For the least memory test case, depth-first and breadth-first order have been considered because of their relevance on memory consumption. Morsa uses less memory than CDO (aprox. 2.5 times less) in all cases and in most of them is even faster. Moreover, Morsa also requires less memory than XMI, although it is much slower. The difference in performance between single load on demand and partial load on demand is due to the fact that a very simple prefetching algorithm is used for partial load on demand, hence not optimizing the subgraph that is loaded from the repository. The cache configuration for single and partial load on demand is 900 objects size cache for Set0 and Set1 and 9000 for Set2, Set3 and Set4 with FIFO and LIFO cache replacement policies for depth-first and breadth-first order, respectively.

10.3.3 Model update Table 4 shows the results of the model update benchmark. These results reflect only the update process, leaving the load and query apart. For the best speed test case, CDO is always slower than XMI (except for the Set1 test model). On the other hand, CDO uses far less memory than XMI for the least memory test case. Morsa is faster and uses less memory than CDO and XMI in all cases except Set0 and Set1, where CDO uses less memory, and Set2, where XMI is faster. These results show that the update of the *Ancestors*, *Depth* and *Breadth* attributes, which support partial load on demand and querying, is not very expensive.

10.3.4 Model delete Table 5 shows the results of the model delete benchmark. Since the current Morsa prototype uses MongoDB, the deletion of a model consists in

Solution	Mode	Set0		Set1		Set2		Set3		Set4	
		Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
XMI	-	38	0.507	102	1.319	812	10.522	2043	17.098	2075	13.260
CDO	Legacy	100	21.397	254	60.621	2430	571.577	-	-	-	-
CDO	Native	91	20.815	202	55.167	2239	596.507	-	-	-	-
Morsa	Full	584	10.121	602	35.670	2739	617.223	4234	2225.906	5831	2225.906
Morsa	Inc	47	24.300	129	73.843	985	1119.038	2280	2820.556	2292	2988.422
Morsa	Inc by update	140	19.140	278	49.314	1856	529.565	2890	1650.324	2900	1805.952

Table 2 Performance results of the model store benchmark

Order	Opt	Solution	Mode	Set0		Set1		Set2		Set3		Set4	
				Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
-	-	XMI	-	40	1.074	154	1.899	1060	15.259	1998	75.665	2245	122.210
-	Speed	CDO	Legacy	151	12.343	370	31.955	2610	401.234	-	-	-	-
-	Speed	CDO	Native	60	9.256	307	23.759	2044	242.545	-	-	-	-
-	Speed	Morsa	Full	82	5.932	713	13.722	1913	165.144	2835	611.584	3256	488.683
-	Mem	CDO	Legacy	87	16.127	267	38.633	2403	426.501	-	-	-	-
Depth	Mem	CDO	Native	47	11.984	173	32.910	420	325.206	-	-	-	-
Depth	Mem	Morsa	Single	27	7.962	54	19.402	173	166.163	352	364.261	387	387.519
Depth	Mem	Morsa	Partial	23	16.023	131	39.864	262	246.402	776	733.254	793	777.505
Breadth	Mem	CDO	Native	50	15.273	170	31.566	412	381.257	-	-	-	-
Breadth	Mem	Morsa	Single	32	14.241	122	35.464	275	250.677	1415	729.431	1489	877.938
Breadth	Mem	Morsa	Partial	35	28.895	121	77.840	381	917.196	1420	2540.299	793	2936.594

Table 3 Performance results of the model load benchmark

Opt	Solution	Mode	Set0		Set1		Set2		Set3		Set4	
			Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
-	XMI	-	38	246	199	497	955	2.680	1961	5.838	2.562	6.304
Speed	CDO	Native	23	327	19	358	174	7.816	-	-	-	-
Speed	Morsa	Single	25	185	44	247	224	6.116	685	4.539	702	4.671
Mem	CDO	Native	4	344	6	297	62	11.326	-	-	-	-
Mem	Morsa	Single	9	189	13	382	17	7.207	41	6.973	44	8.549

Table 4 Performance results of the model update benchmark

Solution	Mode	Set0		Set1		Set2		Set3		Set4	
		Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
Morsa	-	4	90	4	72	4	161	4	202	4	206
CDO	Native	103	24.289	289	64.480	2284	686.554	-	-	-	-

Table 5 Performance results of the model delete benchmark

dropping a MongoDB collection, which is a very fast operation that demands almost no memory from the client application. On the other side, a model deletion in CDO implies finding and deleting all model elements, which is a very heavy and slow process.

10.3.5 Model query Table 6 shows the results of the model query benchmark. CDO and Morsa use less time and memory than XMI, mainly because XMI requires the full model to be loaded into memory prior to its traversal, while CDO and Morsa can fetch all the instances of a given metaclass at once and then traverse only those objects. Morsa is slower than CDO for the smaller test models, but when they grow, it becomes faster. For the least memory test case, Morsa always uses less memory than CDO; even more, its memory consumption is nearly the same for all test models, requiring only 38MB for a model with almost 5 million objects and 6000 classes.

10.3.6 Overall assessment The execution of the test benchmarks has shown that Morsa is indeed faster and uses less memory than CDO for all the basic operations and the Grabats 2009 contest query as the size of the input model grows. Moreover, CDO cannot handle the store of the two largest models. Compared to XMI, Morsa is usually faster and uses less memory when only a model partition is needed, e.g. the update and query test cases. On the other hand, Morsa is slower than XMI when a full model must be traversed. However, when client memory is an issue, the growth of the memory needed by Morsa as the models become larger is less dramatic than the one of XMI. Finally, it must be noted that the traversal algorithm has a remarkable impact on the performance of Morsa, so choosing the cache replacement policy that best matches it and configuring the Morsa driver properly is important to achieve the best performance. Note that both Morsa and CDO are client-server persistence solutions, so there is a communication overhead between the client application and the repository that is not present in XMI, which is a local solution. We have chosen these asymmetrical conditions instead of storing the XMI files on a remote server to show that even though there is a communication overhead, Morsa and CDO still can perform better than XMI in some cases. Finally, CDO provides features that are not yet supported by Morsa, such as version management and fault tolerance. These features are outside the

Opt	Solution	Mode	Set0		Set1		Set2		Set3		Set4	
			Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
-	XMI	-	51	1.098	172	1.914	1221	16.008	2004	77.083	2256	126.548
Speed	CDO	Native	18	335	19	558	91	8.466	-	-	-	-
Speed	Morsa	Single	16	660	21	851	208	6.916	807	17.467	1002	19.514
Mem	CDO	Native	10	340	13	596	26	20.123	-	-	-	-
Mem	Morsa	Single	6	873	6	1.139	12	9.607	38	19.923	38	25.387

Table 6 Performance results of the model query benchmark

scope of this paper, but its is worth noting them because their implementation may have had an impact on the results of the benchmarks; however, in order to minimize such impact, CDO was configured for read-only access (that is, without versioning) were applicable and a single repository was setup.

11 Lessons learned

As commented in the Introduction, this paper extends the work presented at the MODELS 2011 conference [14]. The process of extending our previous work and also the whole design and development of Morsa has taught us several lessons on how a model repository should be conceived and implemented to achieve both good performance and independency from the database backend, as well as more knowledge on the area of model persistence, especially the different requirements of a model persistence solution, how current solutions address them, and how these solutions could they be classified depending on what they are focused on. In this section, the lessons learned by designing and implementing our model repository and choosing a model persistence solution will be explained.

11.1 Repository design and implementation

During the development of Morsa we have identified some issues related to the implementation of the repository that have been addressed in the version described in this paper. First, we identified several elements in which performance could be improved. Secondly, we devised a new design in which independency from the database backend is achieved.

In order to achieve *independency* from the database backend, our new architectural design includes several new components such as the MorsaBackend and the MorsaEncoder, described in Section 4.1. Also for this purpose, the MorsaObject, MorsaReference and MorsaCollection concepts have been added to the data design; these concepts decouple the implementation of the database from MongoDB's DBObject, making it possible to develop prototypes of Morsa for other database backends.

After testing the previous Morsa prototype, we found out that its *performance* could be improved in terms of database querying, model updating and proxy resolution. Regarding to to *database querying*, the previous repository design was composed of a collection for each different metaclass; this was initially conceived to provide faster queries for all the elements of the same type; however, when elements from different metaclasses were requested simultaneously (e.g. in partial load on demand) or when a query involved checking relationships between different metaclasses was requested (in an SQL *join* fashion), one query had to be performed over the MongoDB database for each metaclass, since it does not support multi-collection querying. To solve this, we changed the data design of the repository from one collection per metaclass to one collection per model, as explained in Section 4.2; this boosted the performance of the partial load on demand by lowering its communication overhead, and also provided us ways to design a rich querying interface that is now under development.

Regarding to *model updating*, we found that the update of the metadata related to the graph representation of a model was quite expensive, as it required accessing many elements only for updating. We solved this in our current prototype by using JavaScript scripts that are executed directly on the MongoDB database, hence saving much communication time and client application memory. However, this improvement is optional because it requires server-side scripting support on the database backend. Finally, *proxy resolution* has been improved by using MorsaEObjects (see Section 8), which resolve themselves automatically when any of its features are accessed in a faster way than regular DynamicEObjects.

11.2 Choice of a model persistence solution

The development and evaluation of Morsa has been an enriching experience that has helped us to identify the different needs that each persistence solution covers best. First of all, we have identified three kinds of persistence solutions: *user-oriented*, *application-oriented* and *basic*.

A *user-oriented* persistence solution is one that is intented to be used by humans. Such a solution provides graphical user interfaces, versioning and fine-grained concurrency between clients, even with change notification. It is not focused on managing large models or providing scalability on the client side, but on giving the users support for teamwork and model visualization. CDO and others such as ModelBus and EMFStore fall into this category. On the other side, an *application-oriented* persistence solution is focused on the integration of scalable persistence solutions with tools and applications that manipulate models (e.g. model transformations).

Rather than user interfaces, they provide rich application interfaces that support clean parameterization of model access. Morsa falls into this category. Finally, a *basic* persistence solution just serializes models without any special concerns on scalability, usability, versioning or concurrency. XMI falls into this category. A basic solution is usually the easiest to manage.

In our experience, and given the tools currently available, the choice of what persistence solution to use is rather simple: if the persisted model is going to be manipulated by human users in a distributed environment and it is not very large or complex, a user-oriented one (e.g. CDO) would be the recommended solution; on the other hand, if a large model is going to be manipulated by some application with efficiency constraints for memory and time, an application-oriented one (e.g. Morsa) is recommended. For the rest of the cases (e.g. small models with no teamwork or distribution), a basic one (e.g. XMI) is a good choice.

12 Conclusions and further work

We have presented Morsa, a model repository aimed at achieving scalability for client applications that access large models. Morsa uses load on demand and incremental store mechanisms to allow large models to be persisted and accessed without overloading the client application's memory. We have developed several cache replacement policies that cover different model access patterns. A document-based NoSQL database is used as persistence backend, which is a novel feature since model repositories usually work with object-relational mappings.

We have implemented a prototype for EMF that exhibits promising performance results. An evaluation of our prototype is shown, executing five benchmarks against large models and comparing their results with the ones of XMI and the well-established CDO repository. This comparison demonstrates that Morsa suits better for partial model access and model querying than XMI and CDO, and that it handles larger models than CDO does. Our purpose on the development of Morsa is to build an *application-oriented* repository, that is, one focused on application integration and client scalability; on the other hand, our experience using CDO tells us that it is a *user-oriented* repository, that is, mainly designed for teamwork and model manipulation by human users, and scalability seems not to be one of its design goals, while versioning and fault tolerance are.

Our future work is to continue optimizing Morsa while implementing new features. Among others, these features include: a *query API*, support for *query languages* such as OCL and making our load on demand algorithms and cache replacement policies more adaptative by *collecting metadata information* about the structure of the persisted models. Database support for *metamodel and model analysis* providing knowledge about the structure of models in terms of average connection, depth, breadth, etc. that will be used by load on demand algorithms and cache replacement policies to execute more efficiently. We are also considering *version control* and *synchronization* in order to favour team development.

References

1. Monaheghehi et al.: MDE Adoption in Industry: Challenges and Success Criteria. Proceedings on the Workshop on Challenges in MDE, MoDELS september 2008, Toulouse (France), pages 54-59, Springer.
2. John Hutchinson, Jon Whittle, Mark Rouncefield, Steinar Kristoffersen: Empirical assessment of MDE in industry. ICSE 2011: 471-480
3. Cánovas, J., García, J.: An architecture-driven modernization tool for calculating metrics. IEEE Software, Vol. 27, No. 4. (July 2010), pages 37-43.
4. Gray, J., Zhang, J., Roychoudhury, S., Wu, H., Sudarsan, R., Gokhale, A., Neema, E., Shi, F., Bapty, T.: Model-driven program transformation of a large avionics framework. Generative Programming and Component Engineering (GPCE), pages 361-378, 2004, Springer-Verlag.
5. Díaz, O., Puente, G., Cánovas, J., García, J. Harvesting models from web 2.0 databases. Software Systems and Modeling, 2011. Springer.
6. Kolovos, D., Paige, R., Polack, F.: Scalability: The Holy Grail of Model-Driven Engineering. Proceedings on the Workshop on Challenges in MDE, MoDELS september 2008, Toulouse (France), pages 35-47, Springer.
7. Selic, B.: Personal Reflections on Automation, Programming, Culture and Model-based Software Engineering. Automated Software Engineering vol. 15, number 3-4, pages 379-391, 2008, Springer.
8. The XML Metadata Interchange: http://www.omg.org/spec/XMI/.
9. The Eclipse Modeling Framework: http://www.eclipse.org/emf.
10. The CDO Model Repository: http://www.eclipse.org/cdo.
11. Koegel, M., Helming, J.: EMFStore: A model repository for EMF models. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 2010, Cape Town (South Africa) vol. 2, pages 307-308. http://www.emfstore.org.
12. Morsa prototype URL: http://www.modelum.es/morsa.
13. MongoDB: http://www.mongodb.org.
14. Espinazo-Pagán, J., Sánchez Cuadrado, J., García Molina, J.: Morsa: A Scalable Approach for Persisting and Accessing Large Models. Proceedings on the 14th International Model Driven Engineering Languages and Systems (MoDELS) Conference, Wellington (New Zealand), pages 77-92. 2011. Springer.
15. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling: A Foundation for Language Driven Development. 2004. Ceteva.
16. Meta-Object Facility: http://www.omg.org/spec/MOF/2.0/
17. Teneo: http://www.eclipse.org/modeling/emft/?project=teneo#teneo

18. Stonebraker, M., Moore, D.: Object Relational DBMSs: The Next Great Wave. 1995. Morgan Kaufmann.
19. Stonebraker, M.: SQL Databases vs NoSQL Databases. Communications of the ACM, vol. 53, issue 4, pages 10-11, 2010. ACM.
20. Strauch, C.: NoSQL Databases. Stuttgart Media University, 2011. http://www.christof-strauch.de/nosqldbs.pdf.
21. DeCandia, G. et al.: Dynamo: Amazon's Higly-Available Key-value Store. Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles, pages 205-220, 2007. ACM.
22. CouchDB: couchdb.apache.org.
23. Chang, F. et al.: Bigtable: A Distributed Storage System for Structured Data. 2006.
24. JavaScript Object Notation: http://www.json.org.
25. Caue, C., Didonet, M., Tisi, M.: Transforming Very Large Models in the Cloud: A Research Roadmap. CloudMDE 2012 Workshop at the ECMFA 2012 Conference, Copenhagën, 2012. http://hal.inria.fr/hal-00711524.
26. Grabats 2009 5th International Workshop on Graph-Based Tools: a reverse engineering case study, july 2009, Zurich (Switzerland) http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/.
27. Blanc, X., Gervais, M., Sriplakich, P.: ModelBus: Towards the Interoperability of Modelling Tools. MDA-FA 2003/2004, LNCS vol. 3599, pages 17-32, 2005, Springer. http://www.modelbus.org.s
28. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. 2004. http://labs.google.com/papers/mapreduce-osdi04.
29. Binary JSON: http://www.bsonspec.org.
30. Jouault, F., Sottet, J.:An AmmA/ATL Solution for the Grabats 2009 Reverse Engineering Case Study. Grabats 2009 5th International Workshop on Graph-Based Tools, july 2009, Zurich (Switzerland).
31. Subversion: http://subversion.apache.org.
32. Sriplakich, P., Blanc, X., Gervais, M.: Collaborative Software Engineering on Large-scale models: Requirements and Experience in ModelBus. Proceedings on the 2008 ACM Symposium on Applied Computing, pages 674-681, ACM.
33. MongoEMF: http://bryanhunt.wordpress.com/2011/03/15/mongo-emf/.
34. Resource Description Framework: http://www.w3.org/RDF/
35. Varrò, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Software and System Modeling 2(3). pages 187-210 (2003).
36. Aschauer, T., Dauenhauer G., Pree W.: Representation and Traversal of Large Clabject Models. In Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09), Springer-Verlag, pages 17-31 (2009).
37. De Lara, J., Guerra, E.: Deep Meta-modelling with MetaDepth. TOOLS, pages 1.20 (2010).
38. Bergmann, G. et al.: Integrating efficient model queryies in state-of-the-art EMF tools. TOOLS, Springer-Verlag, pages 1-8, 2012.
39. Atkinson, C., Kühne, T.: Meta-level Independent Modelling. International Workshop on Model Engineering, Paris (2000).
40. Espinazo-Pagán, J., García-Molina, J.: A homogeneous repository for collaborative MDD. Proceedings on the 1st International Workshop on Model Comparison in Practice IWCMP 2010 (Málaga, Spain), pages 56-65, 2010, ACM.
41. Böhm, C., Lorey, J., Fenz, D., Kny, E., Pohl, M., Naumann, F.: Creating voiD Descriptions for Web-scale Data. Winner of the 2010 Billion Triple Track Semantic Web Challenge.
42. Ibrahim, A., Cook, W.: Automatic by Traversal Profiling in Object Persistence Architectures. Proceedings on the 20th European Conference on Object-Oriented Programming, july 2006, Nantes (France), pages 50-73, Springer.
43. Han, W., Whang, K., Moon, Y.: A Formal Framework for Prefetching Based on the Type-Level Access Pattern in Object-Relational DBMSs. IEEE Transactions on Knowledge and Data Engineering, vol. 17, pages 1436-1448, 2005, IEEE.
44. Leong, H., Si, A.: On Adaptive Caching in Mobile Databases. Proceedings of the 1997 ACM symposium on Applied computing, pages 302-309, 1997. ACM.
45. Rathore, R., Prinja, R.: An Overview of Mobile Database Caching. 2008. http://http://www-users.cs.umn.edu/∼rohinip/Rohini_Prinja/Research_files/8701Project.pdf