UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

Un Repositorio de Modelos para Acceso Escalable

D. Javier Espinazo Pagán

2013

UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

A Model Repository
for Scalable Access

D. Javier Espinazo Pagán

2013

Un Repositorio de Modelos para Acceso Escalable

Tesis Doctoral

Memoria que presenta para optar al título de Doctor en Informática

D. Javier Espinazo Pagán

Dirigida por el Doctor

D. Jesús Joaquín García Molina

Departamento de Informática y Sistemas
Facultad de Informática
Universidad de Murcia

Septiembre 2013

A mis padres, por ser mi mejor ejemplo en los estudios y en la vida.
A mi mujer, por acompaarme hasta el fin del mundo y volver para contarlo.

Agradecimientos

Aunque pareciera ayer, ya han pasado unos cuantos años desde aquella defensa de proyecto de fin de carrera donde daba mis primeros pasos en el Desarrollo Software Dirigido por Modelos, al final de la cual mis padres me animaron a emprender la carrera investigadora. Ha sido un camino tortuoso, incierto, como viene siendo habitual en el mundo de la investigación, muy alejado del trabajo mecánico y repetitivo, con todo lo bueno y lo malo que ello implica.

Las primeras personas a las que tengo que agradecer el haber llegado hasta aquí son, por supuesto, mis padres y Jesús García Molina; los primeros, por animarme y apoyarme y el segundo, por saber canalizar todo este esfuerzo, con paciencia y esmero, hacia el fin que representa esta tesis. De mis padres he aprendido a intentar llevar una vida ejemplar y respetar a los demás, valores que sin duda me serán tan útiles en el futuro como lo han sido hasta ahora, que no es poco. De Jesús he aprendido a pensar, leer y escribir científicamente, poderosas herramientas para este mundo donde ya nada se puede dar por supuesto.

No puedo olvidarme ni mucho menos de Isa, mi novia al principio de mi carrera investigadora y ahora convertida en mi mujer, que tan bien me ha acompañado durante este periplo, que tantas veces ha soportado la inquietud de un artículo a medio terminar, de una revisión exigente o de un plazo acuciante, siempre con la palabra adecuada en los labios para ayudarme. Ella ha alimentado la imaginación tan necesaria para la investigación y me ha enseñado el valor de la libertad.

Todos los compañeros que han pasado por el laboratorio han dejado también su granito de arena en esta experiencia personal, formativa y laboral: desde los que coincidieron conmigo más brevemente, como Joaquín, Fernando y Fran, hasta los que han estado más tiempo y han compartido conmigo un pedacito de sus vidas: Javi Cánovas, Javi Bermúdez, Óscar, Jesús Sánchez, Jesús Perera y Juanma. Mis mejores deseos para que terminen la tesis exitosamente (los que están en ello) y para que sigan contribuyendo con su esfuerzo e imaginación a la investigación. Debo una especial mención a Jesús Sánchez, por haber tenido tanta paciencia con mis manías y mi cabezonería; de no ser por él, Morsa no existiría.

Ka mate, ka mate! Ka ora, ka ora!
Ka mate, ka mate! Ka ora, ka ora!
Tēnei te tangata pūhuruhuru
Nāna nei tiki mai whakawhiti te rā
Ā, upane! Ka, upane!
Ā, upane! Ka, upane! Whiti te ra!
KA MATE - Te Rauparaha

Contents

Figures

Tables

Listings

Resumen de la Tesis

Sitting on a cornflake,
waiting for the van to come

La persistencia de modelos

La *Ingeniería del Software Dirigida por Modelos* (Model-Driven Engineering, MDE) ha surgido en los últimos años como una nueva área de la Ingeniería del Software que pone un especial enfásis en el uso sistemático de modelos a lo largo del ciclo de vida software, mejorando la productividad y calidad del mismo en aspectos como el mantenimiento y la interoperabilidad. Las técnicas MDE, como el metamodelado y las transformaciones de modelos, son útiles para abordar la complejidad del software ya que incrementan los niveles de abstracción y automatización [BCW12]. Sin embargo, aunque la MDE ha ido ganando aceptación por parte de la comunidad software, *"su adopción está siendo sorprendentemente lenta"* [Sel12] y uno de los factores críticos para dicha adopción es la existencia de herramientas robustas, usables y eficientes que estén orientadas a la creación de aplicaciones industriales; la persistencia y la consulta de modelos son dos ejemplos de los servicios que tales herramientas deben ofrecer, ya que la capacidad de almacenar eficientemente modelos y de extraer información de ellos es clave para su gestión y uso.

El trabajo presentado en esta tesis aborda los problemas de la persistencia de modelos y de la consulta de modelos. La *persistencia de modelos* es el servicio que proporciona el almacenamiento de modelos residentes en memoria en medios persistentes tales como un conjunto de ficheros o una base de datos. La *consulta de modelos* es el proceso de buscar dentro de un modelo un conjunto de elementos que satisfagan una determinada condición y transferirlos al cliente desde el almacenamiento persistente. Hemos realizado un análisis y caracterización del estado del arte en ambas áreas para obtener el conocimiento que nos era necesario para diseñar e implementar nuestra propia propuesta de persistencia de modelos, *Morsa*, así como nuestra propia propuesta de consulta de modelos, *MorsaQL*, la cual está integrada en la anterior.

Otro aspecto crítico para la adopción de la MDE por parte de la industria es la *escalabilidad* de las herramientas que acceden a modelos grandes. Como se comenta en [KPP08], *"la escalabilidad es lo que está echando para atrás a un gran número* *de potenciales usuarios"*. Para el desarrollo de nuestras propuestas de persistencia y consulta de modelos hemos abordado un escenario *orientado a aplicaciones*, en el que aplicaciones tales como las transformaciones de modelos leen únicamente pequeños trozos de los modelos para procesarlos (por ejemplo, creando nuevos modelos o generando diferentes artefactos como código fuente o documentación); una solución de persistencia que aborde tal escenario debe proporcionar los medios para recorrer partes concretas de un modelo de forma eficiente, en vez de cargarlo por completo en memoria. Como se comenta en [Sel08][KRM$^+$13], el manejo de modelos grandes requiere de algún mecanismo que permita a un cliente cargar sólo los objetos que van a ser usados. Este comportamiento de carga bajo demanda es imprescindible cuando se persigue una persistencia de modelos escalable.

Los repositorios de modelos son una tecnología emergente para la persistencia de grandes modelos, surgida para superar las limitaciones de la persistencia basada en XMI [XMI11], que proporcionan acceso remoto a éstos, así como características avanzadas tales como acceso concurrente, manejo de transacciones y versionado. Actualmente, CDO [CDO12] es el repositorio de modelos más maduro disponible para la plataforma EMF [SBPM08] (Eclipse Modeling Framework), aunque presenta problemas de escalabilidad, como se ha demostrado en esta tesis.

Cuando una aplicación cliente accede un modelo persistente, la *integración* entre dicha aplicación y la solución usada para la persistencia (por ejemplo, un repositorio) debería ser transparente, esto es, conforme a la interfaz de acceso a modelos estándar que se haya definido en el framework de modelado que se esté usando (por ejemplo, la interfaz Resource de EMF) y sin imponer ningún tipo de pre o post procesado en los (meta)modelos para su carga o guardado, como la generación de código fuente específico a partir de los (meta)modelos a almacenar [CDO12][KH10].

La extracción de información de un modelo puede ser realizada de forma visual cuando el tamaño y la complejidad de un modelo son reducidos, como sucedía en las primeras aplicaciones basadas en modelos. Sin embargo, en la actualidad un modelo puede ser muy grande y complejo, por lo que su inspección visual puede ser muy difícil [JS09]. Por otro lado, programar aplicaciones MDE se ha vuelto más complicado en aquellos aspectos relacionados con la extracción de información: las implementaciones proporcionadas por los frameworks de modelado requieren la programación de una lógica de navegación, que con la creciente complejidad de los modelos de entrada, puede ser una tarea tediosa y proclive a errores, dando como resultado un código que no es lo suficientemente legible ni expresivo como para resultar mantenible. Por lo tanto, para la extracción de información contenida en modelos, se hace necesario tener métodos legibles, expresivos y mantenibles de *consulta* de modelos.

Contribuciones

El objetivo principal de esta tesis es la creación de un repositorio de modelos con especial énfasis en la escalabilidad para el manejo de grandes modelos y en la integración transparente. Este repositorio ofrece dos servicios básicos:

- la *persistencia de modelos*, que permite manejar modelos grandes y proporciona a las aplicaciones cliente un acceso escalable integrado de forma transparente y
- la *consulta de modelos*, que permite a un cliente obtener elementos de un modelo almacenado en el repositorio mediante la especificación de las restricciones que deben de ser satisfechas por dichos elementos.

En esta tesis presentamos *Morsa*, una aproximación a la persistencia de modelos enfocada en la **escalabilidad** e **integración**. El problema de la escalabilidad es abordado usando mecanismos de carga bajo demanda y guardado incremental, soportados por una caché de objetos configurable mediante diversas políticas. El diseño del modelo de datos de Morsa está fuertemente inspirado en el paradigma de las bases de datos de documentos NoSQL. Por otro lado, el problema de la integración mediante la implementación de la interfaz EMF de acceso a modelos persistentes, así como mediante el diseño de algoritmos de carga y almacenaje que no requieren ningún pre o post procesado de (meta)modelos. Hemos realizado una implementación para EMF [Mor13] que usa una base de datos MongoDB [Ban11] y que se integra de forma transparente con herramientas cliente tales como los lenguajes de transformación de modelos.

Además de la persistencia de modelos, el repositorio creado ofrece el lenguaje de consultas *MorsaQL*, que ha sido definido como un lenguaje específico del dominio (DSL) interno [Fow10]. MorsaQL ha sido diseñado con los objetivos de **usabilidad**, **seguridad** y **eficiencia** para la definición y ejecución de consultas sobre el repositorio de modelos Morsa.

Finalmente, para estudiar el estado del arte en las áreas de la persistencia y consulta de modelos y obtener el conocimiento necesario para desarrollar nuestras propias soluciones, hemos realizado un estudio y una comparativa de aproximaciones a ambas áreas. Este estudio y comparativa constituyen una contribución valiosa de esta tesis, ya que pueden ser útiles a la comunidad MDE a la hora de decidir qué solución es la idónea en base a las necesidades que se presenten.

Desarrollo

Encontramos por primera vez la necesidad de desarrollar una solución de persistencia de modelos en el contexto de un proyecto de migración dirigida por modelos de

Oracle Forms a Java. En dicho proyecto, se generaban modelos de tamaño medio a grande que hacían necesario tener un almacenamiento persistente eficiente para su almacenaje y acceso; además, se definieron complejos metamodelos de forma colaborativa en el marco de un equipo de investigación, por lo que empezaron a surgir problemas de versionado, demostrándose que el soporte de Subversion [CSFP04] no era adecuado.

Para abordar los problemas de la persistencia y versionado de modelos, desarrollamos un repositorio llamado *Metarep* [EPGM10], del cual presentamos un primer prototipo en el taller ICWMP en Málaga (España), donde recibió un feedback positivo. Sin embargo, algunos miembros de la comunidad MDE expresaron que estaban más interesados en un repositorio escalable para modelos grandes que en una solución de versionado de modelos, por lo que decidimos descartar Metarep y comenzar un nuevo desarrollo, escogiendo MongoDB como su base de datos en lugar de MySQL, que había sido la base de datos de Metarep.

Aunque descartamos Metarep, su desarrollo nos enseñó varias lecciones sobre la persistencia y versionado de modelos que consideramos útil extender, investigando más en el área de la persistencia de modelos. El resultado de dicha investigación fue un estudio que nos sirvió para guiarnos en la elicitación de los objetivos y requisitos de la solución de persistencia de modelos que queríamos desarrollar.

Una vez realizada la investigación en persistencia de modelos, comenzamos a desarrollar *Morsa*, presentando un primer prototipo en el congreso MoDELS en Wellington (Nueva Zelanda) en 2011 [EPSGM11], donde tuvo una muy buena acogida, siendo elegida una de las cinco mejores contribuciones del congreso. Extendimos Morsa para soportar todas las operaciones de persistencia de modelos: almacenaje, carga, actualización, borrado y consulta; usamos Morsa en el contexto del proyecto de Oracle Forms a Java, para el cual el candidato recibió una beca de la Fundación Séneca (Agencia Regional de Ciencia y Tecnología, CARM).

El problema de las consultas a modelos surgió cuando Morsa tuvo que ser usado para obtener elementos que satisfacieran condiciones, mostrando que tanto el código EMF como las aproximaciones de consulta de modelos disponibles en la actualidad para EMF no eran lo suficientemente efectivas, usables ni eficientes. Se tomó la decisión de desarrollar un lenguaje de consultas integrado en Morsa, realizando un paso preliminar de análisis del estado del arte. Con el conocimiento adquirido en dicho análisis desarrollamos el *Morsa Query Language (MorsaQL)*, un DSL interno implementado en Java e integrado en Morsa.

Resultados

Como paso previo al diseño de nuestras soluciones de persistencia y consulta de modelos, realizamos un estudio del estado del arte en ambas áreas (ver los Capítulos

3 y 7, respectivamente). Hicimos esto identificando un conjunto de dimensiones para cada área que nos ayudó tanto a definir las características que una solución de persistencia de modelos debería proporcionar como a evaluar soluciones de consulta de modelos. Usamos estas dimensiones para caracterizar seis soluciones de persistencia diferentes (XMI, ModelBus, EMFStore, CDO, MongoEMF y OOMEGA, ver la Sección 3.2) y cinco soluciones de consulta de modelos diferentes (EMF, EMF Query, MDT OCL, CDO OCL e Inc Query, ver la Sección 7.5) y para compararlas entre sí (ver la Sección 3.3 y el Capítulo 9, respectivamente).

Con el conocimiento adquirido, diseñamos e implementamos Morsa, un repositorio de modelos enfocado a la **escalabilidad** de las aplicaciones cliente que acceden a modelos grandes. El diseño arquitectónico y de datos de Morsa (ver la Sección 5.1) ha sido ideado para soportar el acceso de grano fino a grandes modelos, permitiendo la carga bajo demanda y el almacenaje incremental. La arquitectura de Morsa consiste en un manejador que usa una caché de objetos con políticas que deciden qué elementos del modelo deben ser obtenidos de la base de datos y cuáles no son necesarios, por lo que pueden ser descargados del cliente par ahorrar memoria. El manejador implementa la interfaz de persistencia del framework de modelado para proporcionar **integración**, imponiendo muy pocos cambios en las aplicaciones existentes para poder ser usado como almacenamiento persistente. En el Capítulo 6 se demuestra que el prototipo de Morsa para EMF y MongoDB tiene un mejor comportamiento en el acceso parcial a modelos que XMI y CDO y que es capaz de manipular modelos de mayor tamaño que CDO.

Tras conseguir un prototipo estable de Morsa, diseñamos e implementamos un lenguaje de consultas para Morsa llamado MorsaQL (ver el Capítulo 8), con los objetivos de **usabilidad**, **seguridad** y **eficiencia**. La sintaxis abstracta de MorsaQL (ver la Sección 8.2) proporciona una serie de conceptos que permiten la definición de contextos (esto es, espacios de búsqueda) y condiciones (es decir, restricciones) sobre atributos y relaciones, la navegación de relaciones y la parametrización de la ejecución de una consulta en términos de profundidad, anchura, número de resultados y resolución de *proxies* (ver las Secciones 2.3 y 5.3.1.2). La sintaxis concreta de MorsaQL (ver la Sección 8.3) ha sido implementada como un DSL interno que se asemeja a SQL y EMF Query. Una combinación de los patrones *method chaining*, *nested functions* y *expression builder* (ver la Sección 2.5) permite obtener una sintaxis concreta legible que prorporciona comprobación sintáctica en tiempo de diseño. Por otro lado, dada su naturaleza de DSL interno, MorsaQL puede ser combinado con sentencias Java para extender su funcionalidad. Su eficiencia queda demostrada en la evaluación del Capítulo 9.

Además de Morsa y MorsaQL, esta tesis presenta un análisis, evaluación y comparativa de siete soluciones de persistencia de de modelos y seis soluciones de consulta de modelos (incluyendo Morsa y MorsaQL), lo cual es muy útil para guiar a los desarrolladores de MDE sobre qué solución deberían usar en función de sus necesidades. En las Secciones 3.3, 6.2 y 9.4 se muestran los cuadros comparativos de este estudio. La Sección 10.2 muestra la comparativa entre Morsa y el resto de las soluciones de persistencia de modelos analizadas.

Conclusiones

Esta tesis analiza el problema de la persistencia y consulta de modelos, proponiendo Morsa, un repositorio de modelos orientado a la escalabilidad de las aplicaciones cliente y la integración transparente con el framework de modelado, y MorsaQL, un lenguaje de consulta de modelos integrado en Morsa que permite la definición y ejecución de consultas de forma usable, segura y eficiente.

El desarrollo de Morsa y MorsaQL, junto con el estudio del estado del arte realizado de forma previa a ellos, nos ha proporcionado un conocimiento más profundo de la problemática asociada a la persistencia y consulta de modelos, en dimensiones como la arquitectura de las soluciones desarrolladas, la granularidad de su acceso a modelos, la preservación de la semántica de los metamodelos almacenados, las necesidades sintácticas de un lenguaje de consultas, etc. Por otro lado, la evaluación de Morsa y MorsaQL y su comparación con las principales soluciones de persistencia y consulta de modelos disponibles en la actualidad aportan un importante valor añadido en forma de recomendaciones para la utilización de unas u otras soluciones en función del uso que se les vaya a dar.

Finalmente, el conocimiento adquirido nos ha permitido definir unas líneas generales de investigación que, a nuestro parecer, sería beneficioso explorar de cara a conseguir una mayor alineación de las tecnologías de la Ingeniería del Sofware Dirigida por Modelos para, en la medida de lo posible, paliar la situación actual, la cual *"sufre de un exceso de balas de plata"* [Sel12].

Abstract

Sitting in an English garden
waiting for the Sun

The paradigm of Model-Driven Engineering (MDE) has emerged as a new area of Software Engineering that uses models to improve the productivity and reusability of software in order to achieve industrial standards. Models are the central artefacts in MDE, and a way to efficiently persist them is crucial for the scalability, usability and distribution of model-driven tools such as model transformations. As models grow in size and complexity, the need of model persistence and model querying solutions arises to efficiently storage large models and obtain information from them in a expressive, readable and reliable way.

In this thesis we present 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 a database design as well as some load on demand and incremental store algorithms. A prototype that integrates transparently with the Eclipse Modeling Framework (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.

In order to design and implement Morsa, we analyzed six representative model persistence solutions (XMI, ModelBus, EMFStore, CDO, MongoEMF and OOMEGA) through the identification of seven dimensions on model persistence. A categorization of model persistence solutions was made based on the defined dimensions and the empyrical knowledge achieved with the analysis of the different solutions.

A model query language has been defined for the Morsa repository, providing an efficient, usable and safe means to query models stored in the repository. This language, called MorsaQL, has been implemented as an internal DSL. A metamodel for the abstract syntax of MorsaQL has been defined, paired with a textual notation and a semantics that is given by an interpreter. A set of benchmarks has also been executed, comparing MorsaQL to other model querying approaches and demonstrating its better fitness for our goals.

As a prior step to the development of MorsaQL, we made an analysis and comparison of five different model querying approaches currently available for EMF.

To do so, we identified a set of dimensions on model querying and used them to characterize the different approaches.

Therefore, the contributions of this work are: (i) the Morsa repository, a client-scalable model persistence approach that integrates transparently with applications; (ii) the MorsaQL query language, an efficient, usable and safe querying approach for Morsa and (iii) a survey on model persistence and model querying approaches, including guidelines for future research and for helping MDE developers choose the model persistence and querying approaches that best fit their needs.

1 Introduction

I am he as you are he
as you are me
and we are all together

Since most current human activities are software-dependent, building software has become a strategic activity for the economic and social development of countries. The main goal of the Software Engineering discipline is to achieve the full industrialization of software, that is, to create an industry that produces high-quality software at a low cost. The transition from craftwork to industry is mainly the result of the mechanization of production processes, the standardization and composition of components, the use of tools for task automation and the creation of product lines. A significant improvement in the industrialization of software has been achieved thanks to the application of automation and reuse techniques and the adoption of software standards, but more progress is still needed in order to achieve a true software industry [GSKC04]. The model-based software lifecycle has matured along the last decade as a new paradigm that promises a significant increase in software automation and industrialization, specially when combined with the reusability provided by the software product lines approach.

Model Driven Engineering (MDE) has emerged as a new area of software engineering that emphasizes the systematic use of models in the software lifecycle in order to improve its productivity and quality in some aspects such as maintainability and interoperability. MDE techniques, e.g. metamodeling and model transformations, allow tackling the complexity of software by raising its abstraction and automation levels [BCW12]. These techniques have been proven useful not only for developing new software applications [MDA03][KT08] but also for reengineering legacy systems [ADM07][UN10] and dynamically configuring running systems [BBF09]. Although MDE is gaining acceptance in the software community, *"the adoption of this approach has been surprisingly slow"* [Sel12] and one of the critical factors for its successful adoption is the existence of robust, usable and efficient tools aimed to create industrial applications; model persistence and model querying are two examples of the services that such tools must offer, since the ability to efficiently persist models and retrieve information from them is essential for model management.

The aim of this thesis has been the construction of a model repository, called *Morsa*, that supports scalable access and provides a model querying language called *MorsaQL*. Therefore, we had to tackle the issues derived from the model persistence and model querying research areas areas. *Model persistence* is the service that provides the serialization of in-memory models into a persistent storage such as a file set or a database. *Model querying* is the process of searching a model for a set of model elements that satisfy a given condition and transfer it from the persistent storage to the client application. An analysis and characterization of the state of the art in both areas has been made in order to gain the knowledge we needed to design and implement Morsa and MorsaQL, our own approaches to model persistence and model querying, respectively.

The rest of this chapter is organized as follows: first, the motivation of the work is presented; then, the main contributions of the thesis are enumerated; afterwards, the development of this thesis is explained; finally, the contents of the rest of this thesis are outlined.

1.1 Motivation

The increasing maturity of MDE technologies is promoting their adoption by large companies [MFM+08][HWRK11], 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 complex models with a size of millions of objects. Model-Driven software modernization [CGM10][GZL+04] 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 [CGM10] or data [DPCGM13] 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 [KPP08], *"scalability is what is holding back a number of potential adopters"*. According to [KRM+13], achieving scalability in MDE involves: (i) being able to construct large models, (ii) enabling large teams of modelers to construct and refine large models collaboratively, (iii) advancing the state of the art in model querying so it can cope with large models and (iv) providing an infraestructure for efficient storage, indexing and retrieval of such models. In this thesis we have tackled the third and fourth dimensions — model querying and model persistence, focusing on the scalability on client applications.

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 thesis is an *application-oriented* one, where client 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 [KPP08]. 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 [Sel08] 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 [XMI11] 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 an instance of its metamodel. For example, in the widely used Eclipse Modeling Framework (EMF) [SBPM08] 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 [Sel08], 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 [CDO12] [KH10] 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 3.2. Currently, CDO [CDO12] is the most mature repository for EMF; however, it does not scale properly as shown in Chapter 6.

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, as in CDO [CDO12] and EMFStore [KH10].

Information can be retrieved from models mainly in two ways: visually and programmatically. On the one hand, a developer can visually inspect a model using a model visualizer that represents models as graphs, trees, etc. and manually select the information he or she wants to retrieve. On the other hand, a model can be queried by an application to automatically retrieve the data of interest; the programmer of such an application needs some means to programmatically inspect models in order to retrieve information from them.

The retrieval of information from a model may be performed visually when the size and complexity of such model is low, as was usual in the model-driven applications that first appeared. However, nowadays models can be very large and complex, so visual inspection has become too difficult to perform. To tackle this, some solutions such as model partitioning have been presented [GZL$^+$04] to build views that are easier to inspect by humans. Moreover, the programming of MDE applications has also become more difficult in those aspects related to the retrieval of information because the provided implementations of modeling frameworks usually rely on navigational semantics for this task; however, the growing complexity of the input models makes coding this navigation tedious and error prone, resulting in code that is neither readable or expressive enough to be maintainable.

Because of the need of methods to retrieve information from models in a readable, expressive and maintainable way, dedicated query languages and APIs (Application Programming Interfaces) have been developed. The OMG defined the Object Constraint Language (OCL) [OCL06a] as the standard language for defining constraints (e.g. invariants and preconditions) and queries for MOF models. The Eclipse Modeling Framework also provides an API for querying models named EMF Query [EMF12]. Other query languages such as XPath [XPa99] offer navigational semantics to query models in a more declarative way.

Therefore, the main goal of this thesis is to provide the MDE community with a means to manage large models with a focus on scalability and tool integration. Such management can be performed by two tools:

- A *model persistence* solution that handles large models and provides client applications with scalable access and transparent integration and
- a *model querying* solution that allows a client to retrieve model elements from the model persistence solution by specifying the constraints that must be satisfied by the results of a query in a usable, safe way. Such a solution must also

be efficient in memory and time to contribute to the scalability of the model persistence solution.

1.2 Contributions

In this thesis we present a model repository for scalable access that provides a model querying language. Next, we present the three main contributions achieved as a result of our research work.

Firstly, we have developed *Morsa*, a model persistence approach 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). We have dealt with the problem of transparent tool integration by implementing the EMF interface for accessing persisted models and by designing our load and store algorithms so that no (meta)model pre or post-processing is required. We contribute a prototype implementation for EMF [Mor13] that uses a MongoDB [Ban11] NoSQL backend and integrates transparently with client tools such as model transformation languages.

Secondly, in order to give model querying support to Morsa, we have defined the *Morsa Query Language (MorsaQL)*, a query language integrated with our model repository that has been implemented as an internal DSL [Fow10]. MorsaQL has been designed to achieve **usability**, **safeness** and **efficiency** in the definition and execution of queries against the Morsa model repository.

Finally, a survey and comparison of model persistence and model querying approaches have been made to analyze the state of the art in both areas in order to gain the necessary knowledge for the development of the two solutions. These survey and comparison are also a valuable contribution of this thesis, since it can be useful for the MDE community to decide what solutions to use, depending on their needs.

1.3 Development

The need of a model persistence solution first showed up in the context of a project aimed at developing a model-driven migration from Oracle Forms to Java. In this project, medium-sized to large models were generated, requiring an efficient persistent storage in order to persist and access them; moreover, complex metamodels were defined collaboratively within a small research team, so versioning issues began to appear, as Subversion [CSFP04] support was proven non-optimal.

In order to tackle the problems of model persistence and model versioning, we created a repository called *Metarep* [EPGM10] and presented its first prototype at the ICWMP workshop in Málaga (Spain), 2010, where it received positive feedback. However, several members of the MDE community expressed that they were more interested in a scalable repository for large models rather than in a model versioning solution, so we decided to discard Metarep and start a new development, choosing MongoDB as its database backend, instead of MySQL, which was the one for Metarep. The choice of MongoDB was motivated by its support to scalability, its lightweight client-server communication and because of the better fitness of the document-based database paradigm for the representation of models.

Although we discarded Metarep, its development taught us several lessons on model persistence and versioning that we considered useful to extend, doing more research on the model persistence area. As a result of this research, a survey was produced, guiding the elicitation of the goals and requirements of the model persistence solution that we wanted to create.

Once we had a deeper knowledge of the area of model persistence, we started developing *Morsa*, achieving a first prototype that was presented at the MoDELS conference in Wellington (New Zealand) in 2011 [EPSGM11], where it was very well received, becoming one of the best five contributions of the conference. Morsa was then extended to fully support the different operations on model persistence: store, load, update, delete and query and used in the context of the Oracle Forms to Java project, for which the candidate received a grant from the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, CARM).

The problem of model querying arose when Morsa had to be used in the context of the migration from Oracle Forms to Java to retrieve elements that satisfied conditions, evidencing that plain EMF code and the current model querying approaches available to EMF where not effective, usable and efficient enough. The decision of developing a query language integrated in Morsa was then made, and a preliminary step of analyzing the state of the art was needed. With the knowledge achieved with such analysis, we developed the *Morsa Query Language (MorsaQL)*, an internal DSL implemented in Java that is integrated with Morsa. This language was proven efficient, usable and safe in the context of the Oracle Forms to Java project and using a set of benchmarks derived from the Grabats 2009 contest [JS09].

1.4 Outline of this thesis

The structure of the rest of this document is as follows:

- **Chapter 2** introduces the background needed for the better understanding of this thesis, comprising concepts of MDE, the Eclipse Modeling Framework, model persistence, Domain- Specific Languages and the NoSQL movement.

- **Chapter 3** analyzes the state of the art in model persistence, defining a set of dimensions on model persistence that are broke down into features. These dimensions are used to characterize and compare a selected set of model persistence approaches.
- **Chapter 4** presents the running example that will be used to illustrate the design and operation of Morsa and to evaluate both Morsa and MorsaQL. This running example is based on the Grabats 2009 contest [JS09].
- **Chapter 5** describes Morsa in terms of its architectural and data design, its performance of the basic model persistence operations and its implementation and integration with the Eclipse Modeling Framework.
- **Chapter 6** evaluates Morsa and compares it with XMI and CDO, the most widely used model persistence approaches for EMF. A set of benchmarks is defined and executed against the models provided by the running example presented in Chapter 4.
- **Chapter 7** presents the problem of model querying and introduces the state of the art for this research area. A set of dimensions on model querying is presented in order to characterize the currently available model querying approaches and MorsaQL in the following chapters.
- **Chapter 8** describes MorsaQL in terms of the dimensions presented in the previous chapter. The abstract syntax, concrete syntax and semantics of MorsaQL are explained in detail.
- **Chapter 9** evaluates MorsaQL and compares it against the model querying approaches presented in Chapter 7. The running example from Chapter 4 is used to evaluate the dimensions defined in Chapter 7.
- **Chapter 10** presents our conclusions and further work. To do so, a categorization of Morsa using the dimensions and features defined in Chapter 3 is made; then, Morsa is compared to the selected model persistence approaches. Some research guidelines are outlined for model persistence and finally, the publications, tools, projects and grants related to this thesis are commented.

2

Background

Expert texpert
chocking smokers
don't you think the joker
laughs to you?

This chapter introduces the background needed for the better understanding of this thesis, which consists of: a brief explanation of the basic concepts of MDE, such as metamodeling and model transformations; the Eclipse Modeling Framework (EMF), presenting its metamodel and tooling; the representation of models as graphs and the terminology that is used to describe them; the persistence of models and its related operations; some concepts on Domain- Specific Languages (DSLs) and an approach to the NoSQL movement, describing its main paradigms and benefits.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is the term that is commonly used to refer to the area of Software Engineering that involves a set of development paradigms based on four basic principles [BCW12]: (i) models are used to represent aspects of a software system at some abstraction level; (ii) they are expressed using DSLs (a.k.a. modeling languages) (iii) that are built by applying the meta-modeling technique and (iv) model transformations provide automation in the software development process.

2.1.1 Metamodeling

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 [SBPM08] or

MOF [MOF06]. A metamodeling language is in turn 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* (aggregations or references) between pairs of classes to represent 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 2.1 shows a metamodel called OOMetamodel that represents a simple object-oriented programming language with concepts such as module, class, feature, field, method and parameter, which have been modeled as metaclasses. This metamodel will be used for explaining the concepts of metamodeling and the representation of a model as a graph. A special OOModel metaclass has been introduced to aggregate all modules. All these metaclasses have a name attribute which they inherit from a NamedElement class that is not shown for the sake of readability. The example also shows several association relationships among the metaclasses (modules, classes, returnType, features, type) and a generalization from Method and Field to Feature. Figure 2.2 is a UML object diagram that represents a model of an object-oriented program consisting of two modules, two classes, one method, one field and two method parameters.

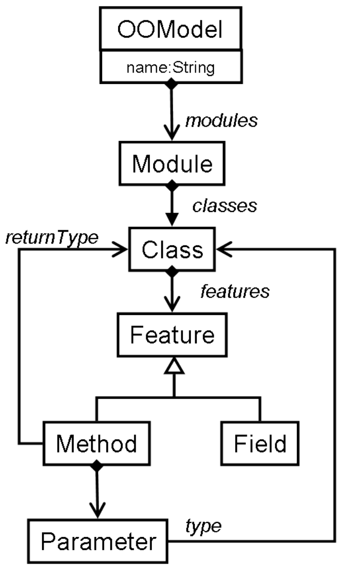


Figure 2.1 Metamodel for a simple object-oriented programming language

In MDE, the four-level metamodeling architecture [CESW08][BCW12] is normally used to explain the relationships between models, metamodels and meta-metamodels. An *instance-of* (or conformance) relationship is given between a model

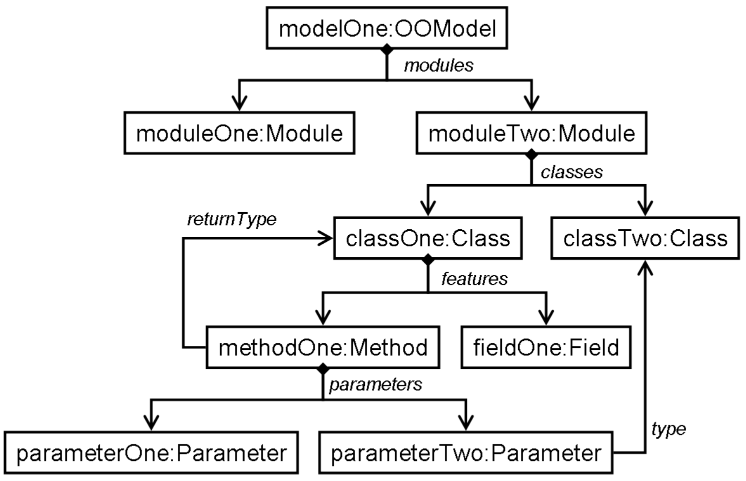


Figure 2.2 Example model representing an instance of a simple object-oriented programming language

and its metamodel as well as between a metamodel and its meta-metamodel. The elements of a (meta)model are instances of (they conform to) the metaclass of its (meta)metamodel. For example, the object modelOne in Figure 2.2 is an instance of the metaclass OOModel in Figure 2.1, which is in turn an instance of the element from the meta-metamodel that represents metaclasses (e.g. EClass in the Ecore meta-metamodel).

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 2.1 is a reference; this relationship is also shown between instances classOne and methodOne in Figure 2.2. 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 a 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 of 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 2.1 is a containment; this relationship is also shown between instances moduleTwo and classOne in Figure 2.2. Although containment relationships are not compulsory in all metamodeling languages, we assume their existence because it is the way EMF (which is the modeling framework used in this thesis) is designed and also because they can be semantically emulated by regular relationships.

2.1.2 Model Transformations

Model transformations allow automating the conversion of models between different levels of abstraction. An MDE solution usually consists of a model transformation chain that generates the desired software artefacts from the source models. Two kinds of model transformations are commonly used: *model-to-model* transformations (M2M) generate a target model from a source model by establishing mappings between their metamodels and *model-to-code* transformations (M2C or *model-to-text*, M2T) generate textual information (e.g. source code) from an input model. A survey on model transformation languages can be found in [CH06]. Some examples of M2M model transformation languages are QVT [QVT08], ATL [JABK08], Epsilon [Eps12] and RubyTL [SCGM07]; MOF2Text [MOF08] and XPand [Eff06] are some of the most widely used M2C or M2T model transformation languages.

M2M transformations are used in a transformation chain as intermediate stages that reduce the semantic gap, while M2C transformations produce the target artefacts at the last stage of the chain. A model transformation language requires accessing to persistence solutions in order to read the input models and, for M2M transformations, to write the generated ones. The process of writing model transformations can be made easier using a declarative model querying approach. It is worth noting that a model transformation language may be described by means of a metamodel [TJF^{+}09], so a transformation could be also persisted and queried as a model.

2.2 Eclipse Modeling Framework

This thesis uses the *Eclipse Modeling Framework* (EMF) [SBPM08] as its reference modeling framework, as the majority of the approaches commented and evaluated in Chapters 3 and 7 have been developed for this framework. EMF is a modeling framework defined for the Eclipse platform and inspired on the MOF modeling framework. [MOF06]. Although it was initially designed for code generation, the MDE community has adopted it as the foundation for most of its model-based developments, being the most widely used modeling framework at the moment. EMF is composed of a meta-metamodel called Ecore and the tooling needed for the creation and manipulation of Ecore (meta)models, which is usually referred also as EMF.

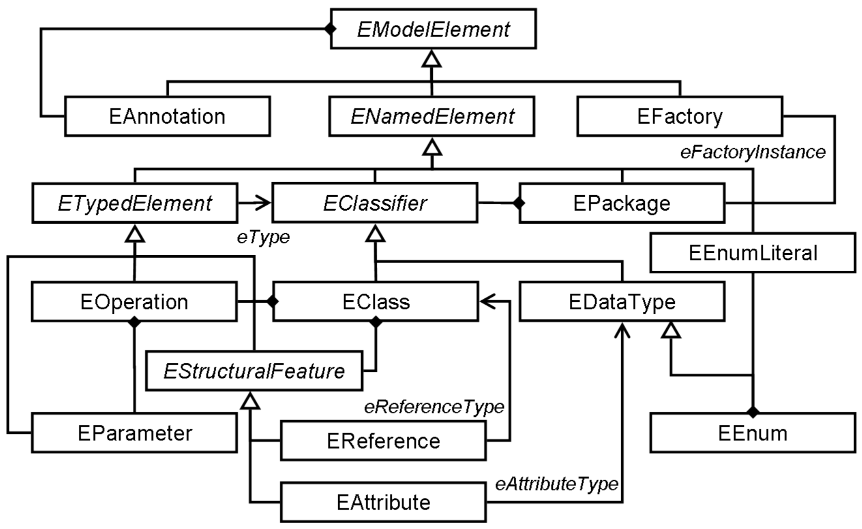


Figure 2.3 The Ecore metamodel

The *Ecore meta-metamodel* supports all the constructs defined in the previous section for metamodeling languages and adds some others that are focused on code generation or integration with Java and the Eclipse platform (e.g. EFactory, which is a factory class that creates model elements). Figure 2.3 shows the main concepts and relationships of the Ecore metamodel (the attributes and the least relevant references have been ommitted for the sake of clarity). As can be seen, metaclasses are represented as EClasses and structural features are represented as EStructuralFeatures, which may be attributes (EAttributes) or relationships (EReferences). The type of an structural feature may be an EClass if it is a reference or an EDataType if it is an attribute. The same meta-metaclass is used for representing both containment relationships and references (a containment boolean attribute is used to discern this, not shown for the sake of readability). A metamodel is represented in Ecore as an EPackage, which may include subpackages (i.e. composite metamodels).

The two basic interfaces of the *EMF tooling* are EObject and Resource: they represent the root class of every model element and the medium in which model elements are stored, respectively, and are essential for any application that uses the framework. EMF supports two ways of manipulating models: code generation and dynamic models. Given a metamodel, EMF can generate a set of classes and interfaces which represent the metaclasses of the metamodel and can be instantiated to create models. This is called *generated EMF*. Instead of generating code, EMF also allows a model to be dynamically generated by creating instances of a generic model object interface called DynamicEObject as the model elements are loaded from a resource. This is called *dynamic EMF*, and offers higher genericity and lower memory consumption, although its performance may be lower.

2.3 Models as graphs

A (meta)model can be represented as a directed labelled graph whose nodes are instances of (meta)metaclasses and whose arcs are determined by association relationships, being the labels on those arcs the names of the associations (for models) or their kinds (for metamodels) Given this, some terminology defined for graphs that will be used throughout this thesis 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 with the same parent;
- its *breadth* is its position in 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 2.2, which shows the containment relationships between model elements. In this figure we can observe the following:

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

2.4 Model persistence

Model persistence is a service provided by modeling frameworks in order to serialize in-memory models into a persistent storage such as a file set or a database. A *persistence solution* is a tooling that provides model persistence to client applications, whether they manage models programatically or through direct manipulation by final users; a model that is serialized and stored in a persistence solution is called a *persisted model.* Model persistence is therefore a key aspect of MDE, supporting most of the MDE techniques, such as metamodeling and model transformation.

There are five basic operations in model persistence that involve moving models between a client application and a persistent storage:

- *Load*: a model or a *model partition* (i.e. a part of a model) is transferred from the persistent storage to the client's memory. It involves rebuilding a set of model elements from their serialized counterparts. If the whole model is rebuilt, it is *fully loaded*; otherwise, the model is *partially loaded.*
- *Store*: a model or a model partition is transferred from the client's memory to the persistent storage. It involves representing an in-memory model in the format used by the persistence solution (e.g. relational tuples). If the whole model is stored at once, it is *fully stored*; otherwise, 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 persistent storage. It involves modifying the already persisted model elements 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 persistent storage. Deletion may be performed automatically when a model or model partition is updated and some model elements have been removed from it.
- *Query*: a set of model elements that satisfy a given condition is transferred from the persistent storage to the client.

These operations are needed when client applications access models and traverse them for different purposes. For example: an M2M transformation may look for a particular element that satisfies a given condition and then traverse all its descendants in order to generate a new target model element; an M2C transformation may simply traverse a whole model, processing each element once or twice. Both applications require loading a source model, traversing it and, for the former, building target model elements in memory; because such source model may be very large, a persistence solution should provide means to perform partial load, achieving scalability. Incremental store is also very important because it provides means to discard already generated model elements after they are stored.

In frameworks such as EMF, models are persisted as XML files, which become unscalable as models grow in size. There is a need for persistence solutions that dynamically manage memory usage, performing partial load and incremental store of model elements; such a persistence solution is called a *repository.* Some popular model repositories are CDO [CDO12], EMFStore [KH10] and ModelBus [HRW09], which will be described in detail in Chapter 3.

2.5 Domain-Specific Languages

A Domain-Specific Language (DSL) is a language that is defined for a specific domain and that is usually created by applying metamodeling, since it is represented by a metamodel and its instances are models. A DSL normally consists of three basic elements: abstract syntax, concrete syntax and semantics. The *abstract syntax* describes the set of language concepts and their relationships, along with the rules to combine them. Meta-metamodeling provides a good foundation for this component, but other formalisms such as grammars have also been used over the years. The *concrete syntax* defines the notation of the DSL, which can be textual or graphical (or a combination of both). The *semantics* defines the behavior of the DSL; there are several approaches for defining it [Kle08], but it is typically provided by building a translator (i.e. a compiler) to another language that already has a well-defined semantics (e.g. a programming language) or an interpreter.

Several techniques have been proposed for the implementation of both textual DSLs [Fow10][MHS05][EPMGM08] and graphical DSLs [KT08][CJKC07]. In this work we focus on textual DSLs, and particularly consider two kinds of styles according to the implementation technique used: external DSLs and internal DSLs. An *external DSL* is typically built by creating a parser that recognizes the language's concrete syntax and then developing an execution infrastructure, if necessary. Nowadays, language workbenches that are based on metamodeling are gaining acceptance as they facilitate the creation of external textual and graphical DSLs; such tools are considered a third technique for building DSLs in [Fow10].

An *internal DSL* or *fluent API*, however, is implemented on top of a general purpose language (the host language) and reuses its infrastructure (e.g. concrete syntax, type system and run-time system), which is extended with domain-specific constructs. The DSL is therefore defined using the abstractions provided by the host language itself. For instance, in an object-oriented language, method calls can be used to represent keywords of the language. Languages with a non-intrusive syntax (e.g. LISP, Smalltalk and Ruby) are well suited for use as host languages [SGM09].

According to [Fow10], an internal DSL can be typically implemented in host languages that have an intrusive syntax or static typing (e.g. Java, C#) using the method chaining, function sequence or nested function patterns, or a combination of them. These patterns use the *expression builder* pattern to build a *semantic model* (i.e. an instance of the class model that represents the concepts of the DSL) and ensure the correction of the instances of the language, acting like a parser [Fow10]. The expression builder is a class that is usually statically imported in languages such as Java to avoid its explicit reference. Code indentation is crucial for the legibility of the DSL using every pattern.

The *method chaining* pattern builds a semantic model by using a sequence of method calls where each call acts on the result of the previous calls; it has the advantage of not needing an explicit variable to hold the context of the part of the semantic model that has been built so far (i.e. a *context variable*); however, it is cumbersome for building hierarchical or nested structures. Listing 2.1 shows an example of how to build an excerpt of the model shown in Figure 2.2 using method chaining in Java: the OOModelBuilder class implements the expression builder pattern, providing a method to create the root model element (ooModel); methods are defined in the different classes to instantiate each metaclass (e.g. module, method) and feature (e.g. name, returnType). Note that special methods such as modules and endModules (lines 4 and 22, respectively) must be used to indicate when an object or collection of objects has been completely built; these methods act as language keywords or delimiters.

```
 1 OOModel modelOne = OOModelBuilder
 2   .ooModel()
 3     .name("modelOne")
 4     .modules()
 5       .module()
 6         .name("moduleOne")
 7       .endModule()
 8       .module()
 9         .name("moduleTwo")
10         . classes ()
11           . class ()
12             .name("classOne")
13             . features ()
14               .method()
15                 .name("methodOne")
16                 .returnType("classOne")
17               .endMethod()
18             .endFeatures()
19           .endClass()
20         .endClasses()
21       .endModule()
22     .endModules()
23   .endOOModel();
```

Listing 2.1 Example of an internal DSL implemented using the method chaining pattern

The *function sequence* pattern builds a model through a sequence of independent function calls. It is more flexible than the method chaining pattern, because each function is called in a separate statement, so non-DSL code can be put in between statements to perform complex operations that are not supported by the DSL. However, it requires a context variable that is implicitly referenced. Listing 2.2 shows how to build an excerpt of the model shown in Figure 2.2 using a function sequence in Java. Note that only static methods from the OOModelBuilder class (i.e. the expression builder) are used; methods that act as keywords or delimiters (e.g. modules and endModule at lines 3 and 21, respectively) are also used. These methods would use a context variable held by the expression builder to keep track of the semantic model built at a certain point.

```
OOModel modelOne = OOModelBuilder.ooModel();
  name("modelOne");
  modules();
    module();
      name("moduleOne");
    endModule();
    module();
      name("moduleTwo");
      classes();
        class();
          name("classOne");
          features();
            method();
              name("methodOne");
              returnType("classOne");
            endMethod();
          endFeatures();
        endClass();
      endClasses();
    endModule();
  endModules();
```

Listing 2.2 Example of an internal DSL implemented using the function sequence pattern

The *nested function* pattern builds a model through a sequence of function calls where the result of a function is used as a parameter for another one, unlike the function sequence pattern, where the function calls are independent. It is visually clearer than the method chaining pattern and it does not need context variables, but can be cumbersome to parse since the order of evaluation is reversed. Listing 2.3 shows how to build an excerpt of the model shown in Figure 2.2 using nested functions. Note that the ending keyword-like functions such as endModule are no longer needed and that the features of an object are no longer identified by method names but by their position as parameters of the nested functions.

```
OOModel modelOne = OOModelBuilder
  .ooModel("modelOne",
    modules(
      module(
        "moduleOne"
      ),
      module(
        "moduleTwo",
        classes(
          class(
            "classOne",
            features(
              method(
                "methodOne",
                "classOne"
              )
            )
          )
        )
      )
    )
);
```

Listing 2.3 Example of an internal DSL implemented using the nested function pattern

The expression builder pattern is also used in these patterns to resolve the references between model elements, which are specified by name. For example, the reference to the classOne object that is specified by name by the returnType reference of the methodOne object in lines 16, 15 and 15 of Listings 2.1, 2.2 and 2.3, respectively, is resolved internally by the OOModelBuilder class.

2.6 The NoSQL movement

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 [SM95].

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.). 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 [Sto10].

The NoSQL [Str11] 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) [SF12].

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

[DHJ+07] favor high scalability over consistency and omit rich querying and analytics features. *Document databases* such as MongoDB [Ban11] and CouchDB [Cou] 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. Documents can form graphs by establishing references among them and containment relationships (i.e., the value associated with a key is a document), so they are a good choice for representing object models; hence, the data model in a document database is composed of several graphs represented by documents and, in some cases as MongoDB, grouped in collections. Finally, *column-oriented databases* such as Bigtable [CDG+06] store and process data by columns 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:

1. *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 [Str11][Cat10].
2. *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 [CDO12].
3. *Accessible*: many NoSQL databases offer their data as JSON objects [JSO] 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 [CDT12].

3

Model Persistence

Mister city
policeman sitting
pretty little
policemen in a row

This chapter presents the current state of the art in model persistence. In order to characterize and evaluate the currently available model persistence approaches, a set of dimensions on model persistence is defined. Then, a selection of model persistence approaches is presented, evaluated and compared. Finally, some conclusions about the state of the art in model persistence are drawn.

3.1 Dimensions on model persistence

In order to provide the operations defined in Section 2.4, several design choices are involved in the creation of a model persistence solution, which can be grouped into dimensions. In this section we identify and define these dimensions, which will also be used to describe the selected set of persistence solutions in Section 3.2. We have identified these seven dimensions on model persistence:

1. The **Storage medium** dimension specifies the features that describe the medium that actually holds the serialized models.
2. The **Architecture** dimension describes how a persistence solution is implemented at a high abstraction level, that is, in terms of components and their relationships.
3. The **Access** dimension defines the features that characterize the implementation of the four basic operations of model persistence related to the manipulation of models (see Section 2.4): load, store, update and delete.

4. The **Query** dimension specifies the features that describe the query operation of model persistence (see Section 2.4).
5. The **Transparency** dimension is concerned about the amount of persistence-specific information that must be considered by a client in order to persist a model and how the solution respects the original semantics of the models.
6. The **Version control** dimension defines the features that are concerned about model versioning.
7. The **Client interface** dimension specifies the features of the communication between a client (human or application) and the persistence solution.

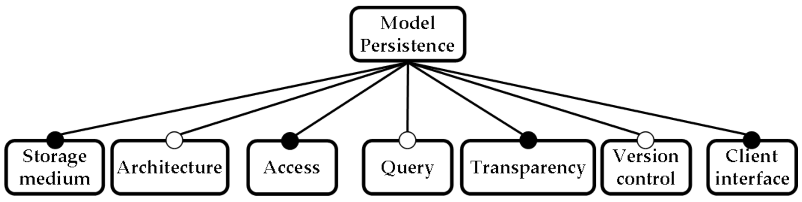


Figure 3.1 The dimensions on model persistence

These dimensions are shown in Figure 3.1 as a *feature diagram*, i.e. a visual representation of a feature model, which characterizes products (in this case, persistence solutions) in terms of the features that they provide [BSRC10]. A feature can be exploded into subfeatures; the relationship between a feature and a subfeature can be mandatory, meaning that the subfeature is required to achieve the feature, or optional. Subfeatures can be alternative or exclusive, meaning that at least one or only one of the subfeatures must be selected, respectively. The definition of these dimensions is the result of studying a variety of persistence approaches, reviewing the existing literature and, most of all, designing and implementing our own solutions [EPGM10]. Each dimension is explained in detail in the following subsections.

3.1.1 Storage medium

The *Storage medium* is the dimension that describes how a model is actually serialized. This dimension is very relevant, since it has a deep influence on the capabilities provided by a persistence solution and also on its scalability and architecture. Figure 3.2 shows the features of this dimension. A storage medium is characterized by its *Physical model* and its *Identification schema*.

3.1.1.1 Physical model

The physical model specifies the technology that implements the persistent storage of a solution; it can be a file set or a database.

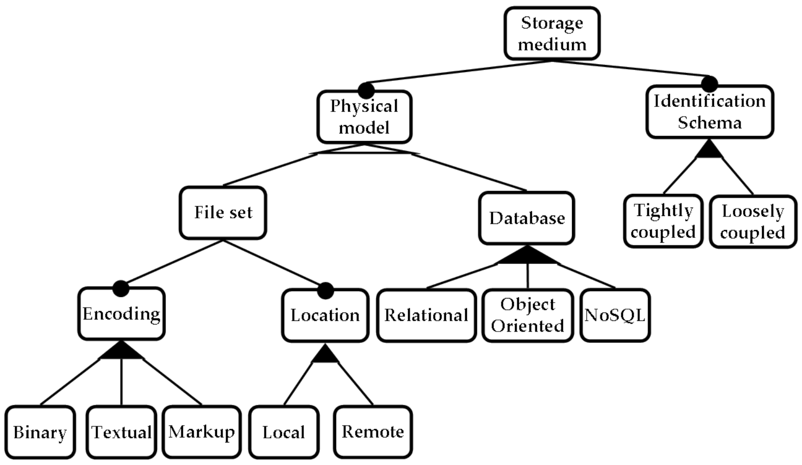


Figure 3.2 The *Storage medium* dimension

A *file set* is a collection of files that represent a model. When a file set is used for model serialization, the persistence solution is usually *file-based*; however, a repository may also use a file set for its implementation. The main difference between a file set and a collection of files used by a database to serialize its data is that a database file usually groups objects based on efficiency criteria, while a file set groups semantically related objects (e.g. from the same model) following criteria such as proximity. A file set is defined by two features: encoding and location.

The *encoding* of a file may be binary, textual or use a markup language such as XML. *Binary* files [JS09] are usually smaller and human clients cannot read them but applications such as transformations can; on the other hand, a *textual* file [Emf] may be easily read by a human client but an application would require a grammar in order to recognize it; finally, a *markup* language [XMI11] is an intermediate solution since it is human-readable (although it may be too verbose) and applications rely on standard parsers to recognize it.

The physical *location* of a file set can be local or remote. A *local* file is stored in the same machine that runs the client application or that is used by the human client, so its access is fast; however, it can only be accessed by clients at the same machine. A *remote* file is stored in a machine that is not the one used by the clients (e.g. file server), so they must communicate with it through a network in order to access the model. This solution is usually slower but allows for distributed access, version control and other capabilities related to the use of remote servers. Both local and remote files can be mixed in a single file set.

A *database* is a storage medium that serializes data as remote files or in the memory of a server. Databases are managed by dedicated server applications that perform services such as access control, transaction support and consistency check. We have considered the three most extended database paradigms: relational, object-oriented and NoSQL, as seen in Figure 3.2.

A *relational* database stores data as rows (i.e. tuples) inside tables (i.e. tuple sets). Since a relational database is not adequate for representing graphs (e.g. object models), an object-relational mapping (ORM) is normally used [SM95]. An ORM is a tool that provides the client an objectual representation of a relational database, simplyfing the management of CRUD (Create, Read, Update, Delete) operations and transactions. A *transaction* is a set of operations and relational databases manage them ensuring the ACID properties: Atomicity (a transaction must be executed completely or not at all), Consistency (the result of a transaction is a valid state of the database), Isolation (transactions do not interfere among them) and Durability (a commited transaction is permanent). A persistence solution that is based on a relational database via an ORM may benefit itself from these properties. However, its representation of models is rather unnatural and may lead to poor performance [SZFK12]: a metamodel is usually represented as a database schema with tables representing the different metaclasses and model elements are represented as tuples inside those tables; relationships are represented as foreing keys. This representation is very rigid since a change in a metamodel cannot be easily propagated to a change in a database schema.

An *object-oriented* database stores data as objects that have references among them. The whole database conforms to an object-oriented data model where objects are instances of classes that can be related through aggregation and inheritance hierarchies. Object-oriented databases are usually built as persistent object-oriented programming languages in object-oriented programming languages [Cat91]. An object-oriented database is well suited for representing models as both them and the object-oriented data model can be represented as object graphs. They appeared in the early nineties as a way to manage complex data that could not be represented efficiently in relational databases (e.g. software artefacts, electronic devices, documents); however, their querying capabilities are limited and they show a low performance on transaction processing [SM95].

The *NoSQL* databases were already introduced in Section 2.6. As a reminder, note that NoSQL databases have some features that may be beneficial to model persistence: they are scalable, schemaless and nearly all of them support JSON for management and object retrieval, providing means to put the models in the Cloud [CDT12].

3.1.1.2 Identification schema

The identification schema specifies how a model element is uniquely identified by a persistence solution. A model element may be identified following a tightly coupled identification schema or a loosely coupled one.

A *tightly coupled* identification schema is one where the identity of a model element depends on other model elements or on the physical implementation of the storage medium. This is a disadvantage, since moving an element from its current position to a new one would change its identity and all the references that point to it would have to be updated. An example of this schema is a URI composed of the path elements from the root element of the model to the identified one (e.g. an EMF URI) or a physical location (e.g. disk or file system address). A tighly coupled database schema is one where the references between model elements are restricted by constraints that do not allow moving an element (e.g. foreign keys in relational databases prevent model elements from moving between tables).

A *loosely coupled* identification schema is one where the representation of each model element is independent from the rest and from the physical implementation of the storage medium. This is usually achieved through unique identifiers (e.g. UUIDs) and mappings between identifiers and actual locations, such as file registers (for file sets) or indexes (for databases). Finally, a mix of loose and tight coupling is possible: a file may contain both URIs and UUIDs [XMI11] and a database may combine UUIDs and foreign keys [EPGM10].

3.1.2 Architecture

The *Architecture* dimension describes the software components that form a persistence solution and their relationships. This dimension has a big impact on the scalability of the solution, as explained below. An architecture for a model persistence solution is usually either a *Client-only* one or a *Client-server* one, as shown in Figure 3.3. The choice between the different architectures is usually a trade-off between scalability (for both client and server), fault tolerance and concurrency.

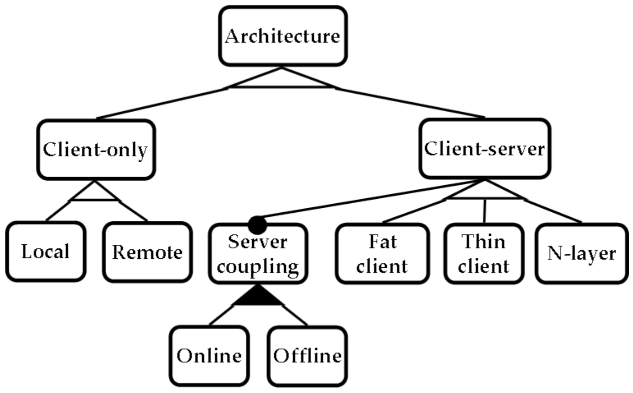


Figure 3.3 The *Architecture* dimension

3.1.2.1 Client-only

A client-only architecture is composed of a *driver* that is used by the client (user or client application) to interact with a passive storage medium such as a file set. All the logic involved in the serialization of a model and the management of the storage medium is dealt by the driver. The medium may be *local* or *remote*; for the latter, a remote server may be managing the medium, but it is not part of the architecture since it ignores its role in the persistence solution. This is the most client-heavy architecture and may lead to memory issues (e.g. memory overload when loading large models).

3.1.2.2 Client-server

A client-server architecture is composed of a driver and a remote server (although the server may also be local). Its main feature is the *server coupling*, which indicates the dependency between the client and the server. If the architecture supports only *online* coupling, then the client must always be connected to the server in order to manipulate persisted models, even if it has already loaded them; on the other hand, the *offline* coupling provides the client means to unbind the loaded objects from the server. We have identified three different client-server architectures for model persistence, depending on the number of components involved, their functionality and the awareness that the server has about the semantics of the stored data: fat client, thin client and n-layer.

A *fat client* is a client-server architecture where the server stores data in a black-box manner, that is, without knowing their actual semantics. In a persistence solution designed in this way, the client uses a driver that encodes a model in the format that is supported by the server (e.g. tuples, documents, etc.) and then sends it. There is no server support for communication or synchronization between clients, so the consistency of the persisted models depends on the drivers, hence the degree of concurrency achieved is low. On the other hand, there is little network communication, which enchances performance.

A *thin client* is a client-server architecture where the server stores data in a white-box manner, that is, understanding their semantics and hence being able to process them. In a persistence solution designed in this way, the driver is used by the client simply to send model elements to the server, which encodes and stores them. The server may support communication and synchronization between clients, providing more concurrency than the fat client, but at the cost of more network communication (e.g. changes in objects must be reported to the server).

An *n-layer* architecture is a thin client where the server role is divided into several logical servers, which can be in the same machine or in different ones. The most common n-layer architecture is the one that hosts the application server and the database engine in different machines. This allows for multiple servers, promoting scalability and fault tolerance; however, more network communication is needed.

3.1.3 Access

The *Access* dimension covers four of the five basic operations related to model persistence defined in Section 2.4: *store*, *load*, *update* and *delete*. It also covers how the local copy is stored in the machine and the support for transactions. Figure 3.4 shows the features of this dimension.

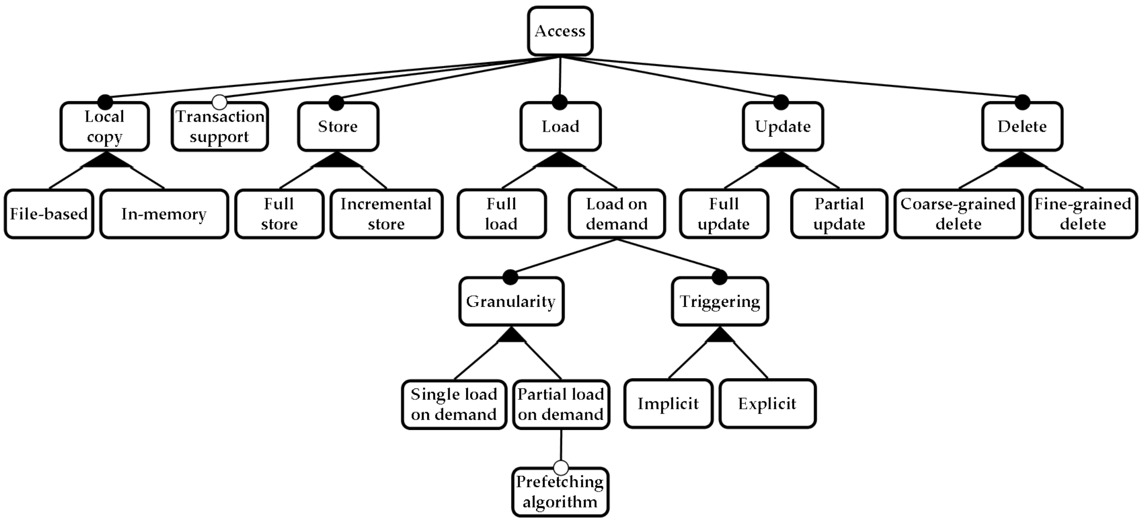


Figure 3.4 The *Access* dimension

3.1.3.1 Local copy

When a client retrieves a model from a persistence solution, a local copy is created in the client machine. This local copy is intended to save communications between the persistence solution and the client application. There are two kinds of local copies: file based and in-memory.

A *file-based* local copy creates temporary or permanent files in the client's file system; these files usually represent entire models or large model partitions rather than individual objects. This is how file-based client-server solutions work and has the advantage of avoiding remote communications; some repositories work in this way too. Its main disadvantages are its absence of synchronization for changes in the original model and the efficiency issues that may be caused by big models being communicated and serialized as files in the client machine.

An *in-memory* local copy is a set of objects in the client's memory. It is more efficient than the file-based one, which would anyway require a memory representation when loaded, and allows for change synchronization with the persistence solution.

3.1.3.2 Transaction support

A transaction, as introduced in Section 3.1.1.1, is a set of operations that must be all successfully executed or else discarded. A persistence solution provides transaction support when the client can, at least, explicitly define the beginning and end of a transaction. Transaction support provides stability by preventing inconsistencies on the stored models.

3.1.3.3 Store

A store operation transfers a model from the client's memory to the persistent storage. Depending on whether the transfer is done in one or several steps, there can be two scenarios: full store and incremental store.

The *full store* scenario is the simplest and is usually the default one. In this scenario, a model is made persistent in a single operation: a client holds the whole model in its memory and then transfers it to the persistence solution; however, this transfer may be done in several steps by the underlying persistence driver to minimize the communication latency, but this is transparent to the client. The main advantages of the full store are its simplicity and its savings in communication latency, since few transfers are done to the persistence solution. On the other hand, its main disadvantage is its memory usage: the biggest model that can be made persistent is the one that fits in the client's memory; moreover, since the storing process usually requires more memory than the actual model for persistence-specific data, even a model that can be kept in memory may not be persisted.

The *incremental store* scenario deals with the main disadvantage of the full store scenario. Suppose a model that is too big to be kept in the client's memory or to be efficiently managed (e.g. a model representing the source code of a software system, comprising millions of elements); this model could have been generated, for instance, by the client either manually or automatically (e.g. by means of a model transformation) or by a remote application that transfers it to the client in a streaming fashion. Storing such a model would be impossible in a single operation (i.e. full store), so the client may partition it in several pieces and store them separately in different store operations. This behaviour is called incremental store: each store operation adds a new model partition to the ones already stored; however, this is not considered a series of update operations since the model is not committed until the last partition is stored, while updates always work between committed states.

3.1.3.4 Load

A load operation transfers a model from the persistent storage to the client's memory. Depending on the number of steps that are required to load a model, there can be two scenarios: full load and load on demand.

The *full load* scenario simply fetches all the objects of a requested model from a persistent storage and transfers them to the client's memory. Again, this transfer may be done by the underlying persistence driver in several steps in order to optimize the communication between the persistence solution and the client. The main advantages of the full load are its simplicity and its smaller communication latency. Its main disadvantage is that the biggest model that can be loaded from the persistence repository is at most the biggest one that can be kept in the client's memory.

The *load on demand* scenario deals with the disadvantage of the full load scenario by defining a way to load and keep in memory only the objects that the client needs in a certain moment. For instance, suppose a model transformation that must traverse only a few elements of a model. In a full load scenario, the whole model would be transferred to the client and then traversed until the elements of interest are located; if the model is too big for the client's memory, it would either overload it or not being loaded at all. However, in a load on demand scenario, the client would request only the elements to be traversed, preventing overloads. Depending on its *granularity*, there can be two load on demand scenarios: single and partial.

A *single load on demand* scenario is the transfer of a single model element from the persistent storage to the client's memory. This is the opposite to the full load and uses the least memory possible. However, if more than one element is actually needed, multiple transfers from the persistence solution to the client may be inneficient, as communication overheads could be avoided by doing just one transfer.

A *partial load on demand* scenario transfers a cluster of model elements (i.e. a model partition) in a single step. This is an intermediate scenario between the full load and the single load on demand: it uses less memory from the client than the former and the communication between the client and the persistent storage requires less transfers than the latter, hence being more efficient.

Depending on the action that *triggers* a load on demand scenario, both single and partial load on demand may work implicitly or explicitly. An *implicit* load on demand is triggered when the client requests one or more model elements that have not been loaded, but without knowing so: the persistence driver transparently manages the load on demand of the requested elements. An *explicit* load on demand is triggered when a client performs a query to the persistence solution. When a partial load on demand is triggered implicitly, a *prefetching algorithm* is executed in order to calculate which elements must be fetched from the persistence solution. Usually, the prefetching algorithm fetches elements that are either physically (i.e. related in the model) or logically (i.e. consecutively requested by the client) close to the ones that have been requested, in order to make them available when they are needed, thus working in a cache-like fashion and saving communication overhead. An implicit triggering is usually preferred to make the access to the persistence solution simpler and more transparent.

3.1.3.5 Update

An update operation transfers the modifications done to previously loaded objects from the client's memory to the persistent storage. An update operation goes from one consistent state (i.e. a state where all the constraints of the storage medium are satisfied) of the persistence solution to another. Note that this does not mean that the state of the persisted model conforms to its metamodel; if this was to be enforced, the flexibility and usability of the persistence solution would be reduced, since partial models [SBP07] could not be stored. The update of a model may be a expensive operation because it implies the modification of the metadata that support the representation of a model in the storage medium. Depending on the granularity of the update, it can be full or partial.

A *full update* is one where a whole model is sent to the persistent storage in order to update some parts of it. The persistence solution may internally update only the modified elements or else delete the whole model and fully store it again. This is not preferred since it requires the transfer of entire models.

A *partial update* is one where only the modified elements are sent to the persistent storage in order to update them. A persistence driver may implement a full update by doing a partial update of the modified elements transparently to the client. This is preferred because a client can request a full update and a partial update with different operations that are performed in the same efficient way.

3.1.3.6 Delete

A delete operation removes one or more elements from a persisted model. If some elements are deleted from a model, but not all, the operation is usually performed as an update. If the whole model is deleted, a dedicated operation should be provided by the persistence solution. Depending on the implementation of such operation, the storage medium and the internal representation of models in that medium, the deletion may be coarse-grained or fine-grained.

A *coarse-grained* deletion removes a model as a top-level entity from the persistent storage (e.g. a table or schema drop in a relational database or as a file removal in a file system). Its main advantages are its simplicity and atomicity.

A *fine-grained* deletion removes a model by removing all its elements from the persistent storage; if the solution is internally designed in such a way that the representation of different models is tangled, a fine-grained delete operation is needed to guarantee the consistency of the storage medium. This behavior is not preferred since it is much slower than the coarse-grained deletion and may also lead to inconsistency issues if the operation is interrupted.

3.1.4 Query

The *Query* dimension covers the basic model persistence operation of query defined in Section 2.4, which searches the elements from a model that satisfy a given condition. Those elements may have already been transferred (i.e. loaded) to the client's memory; otherwise, they are loaded from the persistent storage. It is worth noting that a query must be executed both against the in-memory model and the persisted one because there can be elements in the former which have been modified but not updated int the latter, hence the query must consider inconsistent data. Figure 3.5 shows the features of this dimension. A persistence solution must provide a standard or dedicated *query language* in order to support querying.

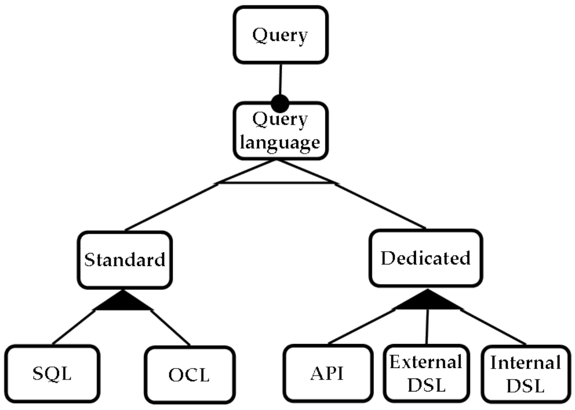


Figure 3.5 The *Query* dimension

A *standard* query language is a language that is well-known and formally defined by an organization. Because of this, its usage is usually supported by third-party libraries or modules that are accessed by the persistence solutions. The most common for persistence solutions are SQL (Structured Query Language) and OCL (Object Constraint Language). *SQL* is the standard query language for relational databases. It is widely spread and its declarative nature makes it easy to use for relational data (i.e. tables and rows); however, models are graphs, and mapping them to relational data is not trivial, making it difficult to use SQL for model querying. On the other hand, *OCL* [OCL06a][WK03] is a declarative language defined for completing UML models, whose capabilities for navigating models are very appropriate for querying models; however, it is not supported by almost any storage medium, so a translation step is usually needed, reducing its performance.

A *dedicated* language is an ad-hoc language defined for the persistence solution; it may be based on a formalism or not. A dedicated language has the advantage of being designed specifically for the architecture of the persistence solution, so it is usually more efficient and usable than a standard language. The actual implementation of the language may be an application programming interface (API), an internal DSL or an external DSL [Fow10]. An Application Programming Interface (*API*) is a set of software artefacts (e.g. methods, data structures, etc.) that provide some functionality to a client application such as querying against a persistence solution. The main advantage of a query API is that it can be combined with the client application's code for simplifying queries that involve loops and alternatives; however, complex queries that test multiple conditions over many attributes and relationships between elements may become too verbose and hence not readable or maintenable. An *external DSL* is a domain-specific language supported by a tooling (e.g. parsers and editors) that is independent from the client application's language. An external DSL should be conceived to simplify complex queries, although the definition of loops and alternatives may be cumbersome, depending on the design of its constructs. An intermediate solution is an *internal DSL* (or fluent API) [Fow10], which is a programming interface whose syntax resembles a DSL; it is more readable than a regular API and it can also be combined with regular constructs of the host language for handling loops and alternatives; greater detail on internal DSLs is given in Section 2.5.

3.1.5 Transparency

The *Transparency* dimension defines what persistence-specific code or actions must be included in the client code or performed on the models, respectively, in order to store and access models in and from a persistence solution. It is also concerned about the degree in which the persistence solution respects the semantics of the metamodeling language and the persisted models. This dimension is relevant because it gives a glimpse on the flexibility of a solution; the ideal solution would not imply any persistence-specific code or operations, using only the persistence interface defined by the modeling framework. We have identified three main features: integration, metamodel management and semantic transparency, as seen in Figure 3.6.

3.1.5.1 Integration

The *integration* between a persistence solution and a modeling framework is done through an API. Its level of transparency depends on which API is used and it can be transparent, opaque or customized.

A persistence solution shows *transparent* integration with a modeling framework when its driver implements the persistence interface of the framework, that is, a

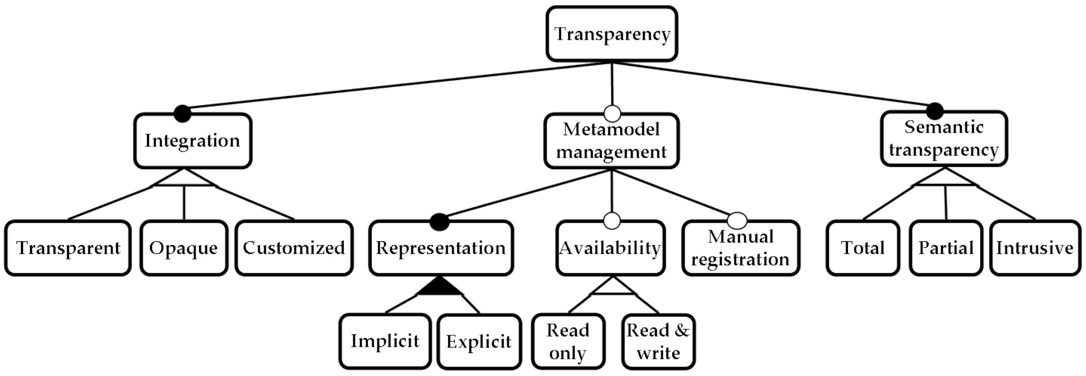


Figure 3.6 The *Transparency* dimension

client code does not require any persistence-specific statement to communicate with the persistence support. This is very convenient, since a user does not need to learn a new persistence interface; however, it hinders the flexibility of the persistence solution because any operation that is not supported by the modeling framework's persistence interface could not be requested to the solution, even if it provides it.

On the other hand, an *opaque* integration uses an API specifically developed for the persistence solution. It may support any kind of operation that is provided by the solution, but any client application that interacts with it must combine this API with the one of the modeling framework, resulting in a code that is not portable between different persistence solutions.

Finally, a *customized* integration schema is transparent for the client application since it implements the persistence interface of the modeling framework, but also adds methods to support the special capabilities provided by the persitence solution. Using a customized integration, a client application may work with the persistence solution without any persistence-specific statements in its souce code and it may also use capabilities that are not supported by the framework. It is worth noting that the persistence solution should use any extension mechanism provided by the modeling framework in order to achieve compatibility with other persistence solutions.

3.1.5.2 Metamodel management

The *metamodel management* feature describes how a metamodel is represented in a persistence solution, its availability to the client and the need of registering it in the solution. This feature is optional because a persistence solution may use a local copy of a metamodel instead of storing it, using the identifier of the metamodel (e.g. an URI) in the storage medium.

The *representation* of a metamodel in a persistence solution can be implicit or explicit. An *implicit* representation is one where the metamodel structure is obscured in the persistence solution, that is, it is not represented as a regular artefact of the storage medium. For instance, when a metamodel is represented as a database schema instead of as a set of rows stored in one or more tables. Although an implicit representation is not inherently bad, it is usually bound to a low level of availability (see below) and the management of metamodels by the storage medium is hindered. Its main advantage is that the metamodel may be accessed by the user quickier than a regular model; for instance, it is faster to access a metamodel represented as a compressed file inside a database than to rebuild it from tables and rows. An *explicit* metamodel representation stores the metamodels as regular artefacts of the storage medium, so they can be manipulated. This is desirable because it is more flexible, as explained below.

The *availability* of a metamodel represents the operations that a client can perform over a metamodel stored in a persistence solution and it can be none, read-only and read & write. The availability is *none* when a metamodel is totally transparent to the client, i.e. it cannot access it in any way. This is a drawback because it makes the solution less flexible; for instance, a change in a metamodel would require rebuilding its representation, and models would need to be migrated to be consistent with the new metamodel. As commented above, an implicit representation leads to a low availability, since it is transparent to the client as the metamodel is not represented as a regular artefact of the storage medium. *Read-only* and *read & write* availabilities allow clients to load and update metamodels, respectively. The most desirable availability is the latter because it allows the client to modify metamodels, providing more flexibility.

Finally, a persistence solution may require the *manual registration* of metamodels in order to use them. This is usually imposed by solutions that use implicit representation and it is a negative feature since it makes the client aware of the actual persistence solution being used.

3.1.5.3 Semantic transparency

The degree of persistence-related metadata added to metamodels and models determines the level of *semantic transparency* provided by a persistence solution, which may be total, partial or intrusive.

Total semantic transparency does not modify models or metamodels and handles persistence metadata in separated artefacts. This prevents issues caused by conflicts between models and persistence metadata (e.g. name conflicts in features) and makes the client unaware of the specific persistence solution that is accessed.

Partial semantic transparency is achieved when models or metamodels are modified in order to store persistence metadata but this modification is transparent to the client. For instance, EMF EAnnotations are usually transparent to client applications such as model transformations. Partial semantic transparency is less desirable than total transparency but is better than intrusive transparency.

The lowest semantic transparency is the *intrusive* one, which is caused by severe modifications on models and metamodels in order to store persistence metadata; such modifications include changes in attribute values, additions of new metaclasses, etc. Not only the semantics of the models are modified, but also the changes may lead to validation issues (e.g. a new metaclass has the same name as an existing one) and other problems. This is obviously the least desirable transparency.

3.1.6 Version control

The *Version control* dimension exposes the features of a version control system for models and metamodels. The development of a software artefact usually goes through different stages where different developers introduce changes on the artefact; those changes need to be commited and merged (combinated) and may conflict with each other. A version is a software artefact that has changed, and a revision is a version that has been validated and commited. A Version Control System (VCS) [ASW09] is a tool that manages the versioning of software artefacts, i.e. the validation, commiting, merging and conflict resolution of versions. The features of this dimension are: collaboration schema, VCS architecture, merge, branching and model comparison (see Figure 3.7).

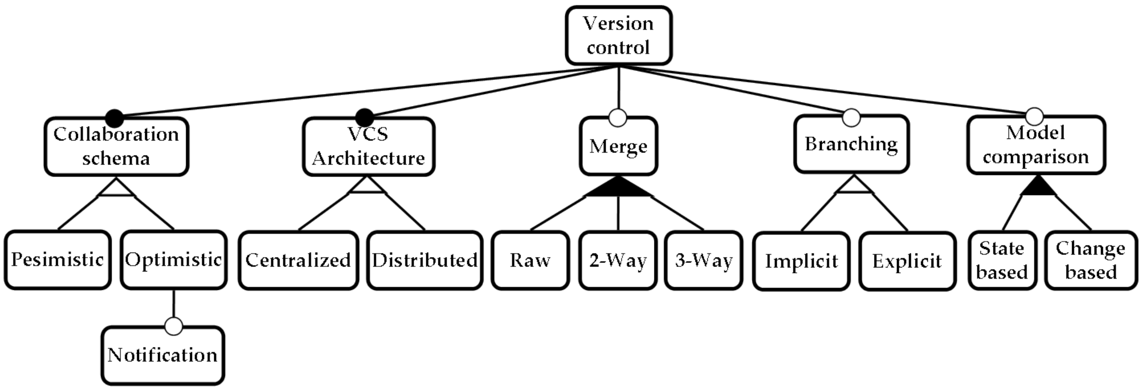


Figure 3.7 The *Version control* dimension

3.1.6.1 Collaboration schema

Different developers may share models under distributed development using either a pesimistic or optimistic collaboration schema.

A *pesimistic* collaboration schema provides mutual exclusion by following a lock-modify-unlock paradigm, i.e. only one user may modify a model or model element (which is locked by that user) at a time while the others wait. This is a coarse-grained solution that severely perjudices concurrency, although it is safest one.

An *optimistic* collaboration schema provides a totally distributed solution since it allows users to keep a copy of a model (a local copy), modify it and store it back in the persistence solution, whose VCS uses a comparison algorithm to determine changes and conflicts. Such a schema may support change *notification*, which involves notifying clients whenever a change is made on a shared element. This is useful for collaborative design and the degree of relevance of remote changes may be customizable from just notification to local copy invalidation.

3.1.6.2 VCS Architecture

The architecture of a VCS describes not only its implementation in terms of distribution but also the way clients interact with the version control. The architecture of the version control system may differ from the one of the persistence solution, and it can be centralized or distributed.

A *centralized* version control implements a client-server model with a master repository that is accessed by clients [CSFP04]. The master repository handles model comparison, conflict detection, and versioning history.

A *distributed* VCS uses local servers deployed on each client machine, so the client does not need a network connection anymore to commit versions, access versioning history, etc. [Cha09]. Servers may connect among them and share all changes performed on the persisted models to build a unique global revision. This is far more complex than a centralized VCS, but more flexible.

3.1.6.3 Merge

A merge is the combination and resolution of conflicting changes performed on the same model by different clients. Although merging is an operation provided by optimistic schemas, pesimistic schemas may also provide it for combining branches. There are three kinds of merge: raw, two-way and three-way.

A *raw* merge simply performs all the changes in a certain order (e.g. temporal order) on the original persisted model. This is the simplest merge but it does not resolve conflicts, so it may lead to undesired resolutions.

A *two-way* merge performs a model comparison among the different versions and detects the conflicts among them, but since it does not consider the original version of the model, it cannot detect changes like creation or deletion of elements.

A *three-way* merge performs model comparison among the different versions and the original version of the model, detecting all kinds of changes. It is the de-facto standard for current VCSs such as Subversion [CSFP04].

3.1.6.4 Branching

A branch consists of a set of revisions that are conceptually or semantically related among them and represent a separated development line from the rest of revisions. Branches are useful to provide different versions of the same model for different development scenarios such as maintenance, design, production, etc. Branching support in VCSs may be done either implicitly or explicitly.

Implicit branching usually replicates the original (root) model and stores each revision (branch) in a different location. A branch is like any other model, but the VCS engine is responsible of recognizing its branch nature.

Explicit branching stores branch-related metadata natively in the storage medium. This may be done through mechanisms such as labels, tags or separated folders. The granularity of such metadata may vary: for instance, a single representation of a model may contain elements that belong to different branches.

3.1.6.5 Model comparison

Model comparison is a key feature for version control because it provides the information for identifying differences and conflicts among models. A model comparison algorithm determines whether two models are equal or not, and the deltas (i.e. differences) between them. There are two strategies for model comparison: state-based and change-based. The main difference between them is how they recognize the correspondences between elements.

State-based comparison algorithms use heuristic-driven matching algorithms to recognize which elements in the new revisions correspond to each element in the original one, and then calculate changes between them [BP08]. They are usually slow since they traverse the whole models and their accuracy is not perfect; however, they require few or none additional information on the models.

Change-based comparison algorithms keep trace of loaded elements in order to record their changes in the different versions [BHH$^+$12]. Once these elements are stored back, the comparison algorithm inspects only their changes, instead of the whole models. These algorithms are preferred because they are faster than the state-based ones and more reliable; on the other hand, they require additional information on model elements (e.g. change descriptors) and the unique identification of each model element is a must in order to compare different versions.

3.1.7 Client interface

The *Client interface* dimension describes the interaction between a client and a persistence solution. There are two kinds of interfaces: User Interfaces (UIs), which are accessed by human users, and Application Programming Interfaces (APIs), which are acessed by applications.

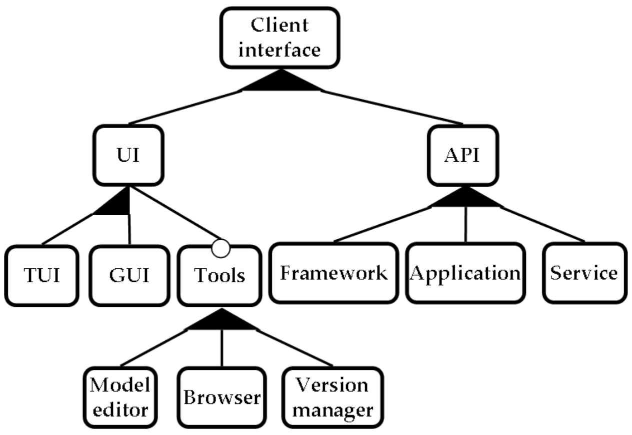


Figure 3.8 The *Client interface* dimension

3.1.7.1 UI

Two kinds of UIs can be used by human users to access the persistence services: textual (TUI) and graphical (GUI). Moreover, a user interface may provide different tools for managing the persistence solution.

A Textual User Interface (*TUI*) is a console-based program where a human user types commands in order to manage the persistence solution. A TUI is the simplest kind of user interface and the most limited, since it cannot display complex visualizations, so its functionality usually consists of simple operations such as downloading models to local files, deleting them or managing user permissions.

A Graphical User Interface (*GUI*) is a part of a window-based application which shows information on the state of the persistence solution (e.g. lists of stored models and their versions), visualizes persisted models and offers support for operations (e.g. wizards for queries). Being a GUI more complex than a TUI, it normally supports the tools for repository management described below.

The user interface of a persistence solution may provide a set of *tools* in order to facilitate the management of the persisted models and the solution itself. Such tools may include editors, browsers and version managers. A *model editor* provides manipulation of persisted models, usually in a graphical interface (e.g. a tree or a graph) with property sheets and other capabilities such as querying. A persistence *browser* shows all the persisted models to the user in order to facilitate their access (rather than having to remember their identifiers) and query for whole models. A *version manager* allows users to access any revision of a model, create and manipulate branches, merge them and other VCS functions.

3.1.7.2 API

An Application Programming Interface (API) is defined to give access to the features provided by a persistence solution via method invocation. APIs provide integration with artefacts such as model transformations. Depending on its design, an API may be a framework, application and/or a service one.

A *framework* API is designed to be integrated with a modeling framework such as EMF or Epsilon, normally at a very low level of abstraction, which is beneficial for application integration (e.g. with model transformations) as commented in 3.1.5. Such an API usually supports fine-grained access such as load on demand.

An *application* API is designed to be integrated with an application such as an Integrated Development Environment (IDE) e.g. Eclipse [1] or Visual Studio [2]. Persistence solutions designed in this way usually offer a higher level of abstraction, delegating model access to the methods of the hosting application.

A *service* API is designed to be exposed as a remotely available service such as a web service or a CORBA server [COR12]. Its level of abstraction may vary, but it is usually restricted to downloading local copies of models and manipulating revisions. However, using low-level remote services such as CORBA may provide a more detailed access to the persisted artefacts.

3.2 Persistence solutions

We have selected six persistence solutions that are representative either for their popularity or for their capabilities, including XMI, ModelBus, EMFStore, CDO, MongoEMF and OOMEGA. We have considered XMI because it is the default one for frameworks such as EMF; ModelBus and EMFStore have been included because of their rich user interface and integration with Eclipse; CDO is the most widely used model repository and also an Eclipse project; MongoEMF takes a different approach to model persistence by using a NoSQL database and finally OOMEGA has been included because of its integrated all-in-one design.

Each solution is evaluated in three steps: (i) the solution is presented; (ii) its main features are explained: architecture, storage medium, model and metamodel representations, client access (including querying capabilities) and version control and finally (iii) a brief description of the solution in terms of the seven dimensions on model persistence is given.

[1]The Eclipse Platform: http://www.eclipse.org
[2]Microsoft Visual Studio: http://www.microsoft.com/visualstudio

3.2.1 XMI

The XML Metadata Interchange (XMI) [XMI11] file format is an OMG standard for exchanging metadata information via XML. XMI is the standard way to serialize models and metamodels in MOF and EMF. In this chapter we analyze XMI as used by the EMF modeling framework.

The architecture of an XMI-based solution involves an XML parser and a set of one or more files that represent models and metamodels. A model is represented as one or more files that reference each other through URIs. Metamodels are represented as models, but in separate files. A client accesses models by fully parsing the XMI file(s). Although there is no support for querying or version control, EMF provides tools for such operations (e.g. EMFQuery [EMF12] for querying and EMFCompare [BP08] for comparison). The seven dimensions on model persistence are covered by XMI as follows:

1. **Storage medium**: consists of a *local* or *remote file set* with *markup* encoding using either a *loosely* or *tightly* coupled identification schema.
2. **Architecture**: a *local* or *remote client-only.*
3. **Access**: a *file-based* local copy without transaction support can be accessed for *full store*, *full load*, *full update* and *coarse-grained* delete, since the SAX parser that is normally used does not support load on demand.
4. **Query**: no query support is provided.
5. **Transparency**: being XMI the default persistence solution for EMF and MOF, it provides *transparent* integration with *explicit*, *read & write* metamodel representation without any registration and *total* semantic transparency.
6. **Version control**: no version control is provided.
7. **Client interface**: no user interface is provided, being a *framework* API the only way to communicate with the persistence solution.

3.2.2 ModelBus

ModelBus [3] [HRW09] is a framework that provides integration of MDE tools for EMF inside the ModelPlex [4] project; it is implemented as an Eclipse application. It is based on the idea of modeling service as an operation having models as inputs and outputs. Different modeling services may be connected through CORBA and web service-based middleware.

The architecture of a ModelBus repository is shown in Figure 3.9. It consists of a ModelBus server, which is a standalone Eclipse application, and a Subversion server,

[3]ModelBus Model-Driven Integration Framework: http://www.modelbus.org
[4]ModelPlex Modeling Solution for Complex Software Systems: http://www.modelplex.org

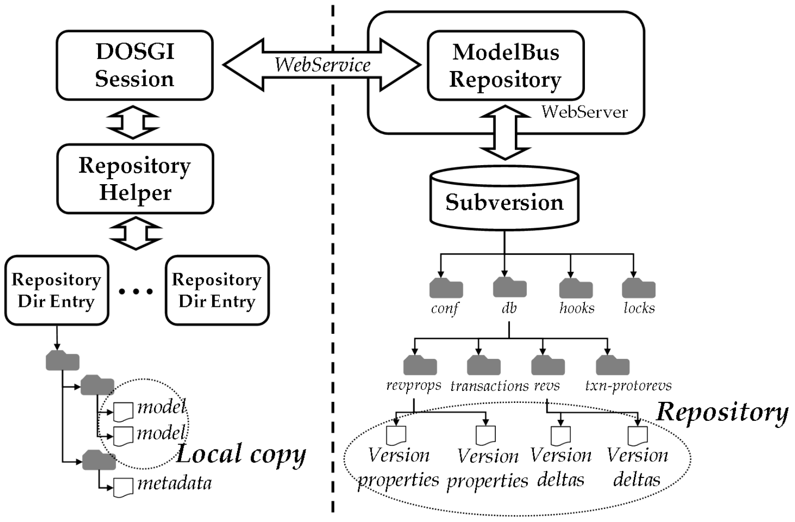


Figure 3.9 ModelBus architecture

both shown at the right side. The ModelBus client (left side) uses an Eclipse instance that holds local copies of the accessed models, which are managed by a repository helper that communicates with the server via a DOSGI session [OSG03]. A model is represented as a set of one or more XMI files held by a Subversion repository, i.e. a folder in the Subversion server; metamodels are also represented as XMI files. A client accesses models through web services that provide methods to fully download them as local copies and to search the repository; once a model is downloaded, the ModelBus client builds a user interface for its manipulation. The integration of a Subversion server provides version control capabilities; for each model, its repository contains subfolders that hold revision properties, binary-formatted deltas and artefact locks. Model comparison is provided by EMFCompare. The seven dimensions on model persistence are covered by ModelBus as follows:

1. **Storage medium**: consists of a *remote file set* with *markup* encoding (XMI files) using either a *loosely* or *tightly* coupled identification schema. Versioning information uses *binary* encoding.
2. **Architecture**: a *thin client* client-server architecture with *offline* server coupling is provided thanks to the use of local copies downloaded from Subversion.
3. **Access**: a *file-based* local copy can be accessed for *full store*, *full load*, *full update* and *coarse-grained* delete, since the local copy is an XMI file. *Transaction support* is provided by Subversion.
4. **Query**: no query support is provided by the repository.
5. **Transparency**: ModelBus shows *opaque* integration because of the manual download of models. Metamodels are managed in the same way as XMI (see Section 3.2.1) and *partial* semantic transparency is achieved due to Subversion.

6. **Version control**: ModelBus' embedded Subversion engine provides *optimistic* collaboration schema in a *centralized* VCS architecture supporting *implicit* branching and *3-way* merge with *state-based* comparison.
7. **Client interface**: ModelBus relies on the Subversion Eclipse plugin to show a rich *GUI* with all kinds of tools. Its API is both an *application* and a *service* one, because it depends on Eclipse and support web services.

3.2.3 EMFStore

EMFStore [5] [KH10] is an EMF-based repository designed for usability. It is part of the UNICASE [BCHK08] client, an Eclipse application for model integration that supports traceability, bug tracking and project managing, among others. Validation, migration, edition and navigation of models are provided by a set of tools called the EMF Client Platform, which is bundled with EMFStore.

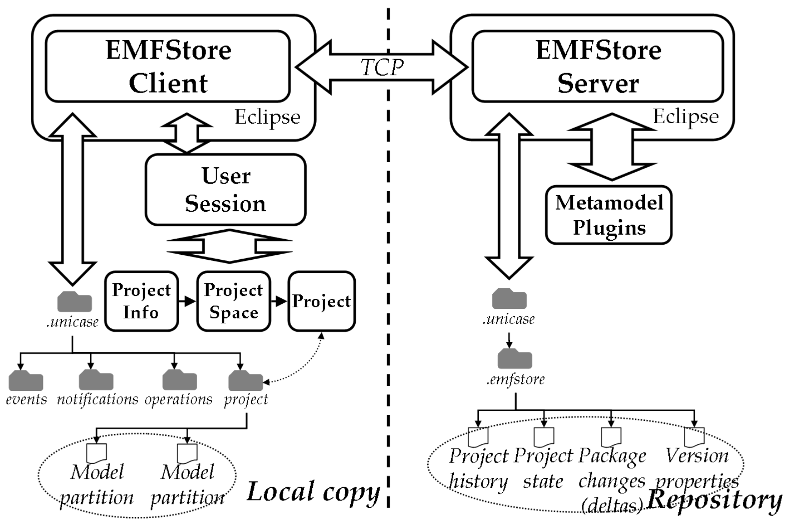


Figure 3.10 EMFStore architecture

The architecture of an EMFStore repository is shown in Figure 3.10; it is similar to the one of a Subversion repository. The EMFStore server and the clients run into separate Eclipse instances. The local copy of a model is represented in the client side as a project, i.e. a set of folders, containing one or more XMI files that hold model partitions; model elements are identified by identifiers that are locally unique for that model. A user session manages project (model) repository metadata. In the server side, a model is represented resembling a Subversion project (see Section 3.2.2). A metamodel is represented as an Eclipse plugin that has to be manually

[5] EMFStore, a model repository for EMF: http://eclipse.org/emfstore/

registered in the EMFStore server; such plugin contains EMF code generated from the metamodel, which has to be modified to add a special metaclass at the top of its hierarchy and to remove non-supported features. A client accesses models through an API or a GUI for browsing projects (models) and fully downloads them as XMI files with UNICASE metadata for visualization and version control. EMFStore provides version control capabilities similar to the ones of Subversion, but without branching. Each project stores a stable model and revision deltas (i.e. change descriptions) that are used to build a certain revision from the stable model and that can also be merged to rebuild the model at a customizable revision-count pace.

1. **Storage medium**: consists of a *remote file set* with *markup* encoding (XMI files) using either a *loosely* or *tightly* coupled identification schema.
2. **Architecture**: EMFStore is implemented as a *thin client* client-server architecture using Eclipse. Local copies provide *offline* server coupling.
3. **Access**: a *file-based* local copy can be accessed for *full store*, *full load*, *full update* and *coarse-grained* delete, since the the local copy is an XMI file. *Transaction support* is provided by the repository.
4. **Query**: no query support is provided.
5. **Transparency**: EMFStore shows *opaque* integration and *intrusive* semantic transparency because of the modifications it imposes on metamodels, which require *manual registration* and are represented in an *implicit* way.
6. **Version control**: like Subversion, EMFStore provides *optimistic* collaboration schema in a *centralized* VCS architecture with *3-way* merge, but using a *change-based* comparison. Branching is not supported.
7. **Client interface**: EMFStore provides a *GUI* with all kinds of Subversion-inspired tools, but simpler and less functional. Its API is an *application* one that depends on Eclipse and UNICASE.

3.2.4 CDO

Connected Data Objects (CDO) [CDO12] is an EMF model repository designed for collaborative distributed development of models. CDO is the most widely used model repository mostly because of its Eclipse project status and its inclusion in the Eclipse Modeling bundle; however, it has been shown that it lacks performance for large models [SZFK12]. It provides features like replication and conflict resolution with a rich user interface for repository management and model edition. It is implemented as both an Eclipse plugin and a standalone application.

The architecture of CDO is shown in Figure 3.11. It consists of a client, a database and a server containing several managers for tasks such as session management, querying, transaction management and version control; these managers use a set of

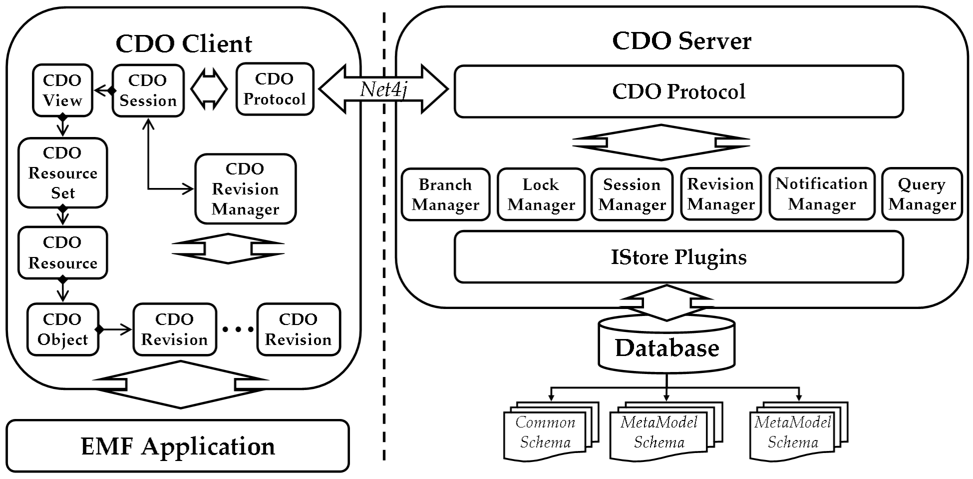


Figure 3.11 CDO Architecture

plugins to communicate with one or more databases. The client side of CDO provides a GUI for model edition and simple repository management. On the server side, different database paradigms (relational, NoSQL and object-oriented) are supported, although the database architecture is based on an object-relational mapping (ORM) designed for relational databases, which is emulated in the rest, hence not taking advantage of the particular benefits of each one. A model is represented in CDO as a set of rows in the database schema that represents its metamodel; such a schema is usually composed of a table for each concrete metaclass or for each metaclass (abstract or concrete); metamodels are also stored as compressed files in a dedicated table. A client accesses models using an implementation of the standard EMF persistence interface (Resource) called CDOResource that handles CDOObjects, which implement the EMF model object interface (EObject). Clients use CDOResources to manage transactions, views and communication with the CDO server. In order to access a model, a connection to the CDO repository must be done, which can be a transaction (read & write access with explicit locking, commit and rollback), an auditory (a read-only view of the state of a model at a certain timestamp) or a plain read-only view of the current version. CDOObjects representing model elements may fall into two different kinds: native and legacy. Native objects support load on demand and versioning, but CDO-specific code generation and manual registration of their metamodels is needed, while legacy objects do not support those features, although their metamodels also need to be registered.

A client may query the repository using SQL, OCL and a variation of the former called HQL if Hibernate [BK04] is used as the ORM. SQL queries are very cumbersome since the actual database schema needs to be known, which is not a trivial task since an ORM is used. OCL queries require the server to load all the objects of the queried model. Furthermore, the logical model of CDO does not allow querying elements of a single model (all persisted models are queried simultaneously). Optimistic version control is provided by CDO with change notification and invalidation when working online; it also supports automatic merging (conflicts cannot be resolved manually by the client) and branching. However, clients must rely on timestamps to access revisions. The seven dimensions on model persistence are covered by CDO as follows:

1. **Storage medium**: CDO stores models in a *database*, which may be of any *kind*, but with a *tightly coupled* identification schema.
2. **Architecture**: an *n-layer* architecture with a client, a server and a database is used by CDO. Although older versions supported only online server coupling, the current one (4.0) supports *offline* coupling by storing changes in the client.
3. **Access**: CDO uses an *in-memory* local copy and supports *full store*, *full* and *partial* updates and *coarse-grained delete*. *Full load* and both *single* and *partial load on demand* are supported with *implicit* triggering. Partial load is managed by a configurable *prefetching algorithm*. Moreover, *transactions* are supported.
4. **Query**: supported query languages are *SQL* and *OCL*.
5. **Transparency**: CDO implements a *customized* integration. Metamodels are represented in an *implicit* and *read-only* way with *manual registration*. Native objects allow for *partial* semantic transparency.
6. **Version control**: CDO provides *optimistic* collaboration schema with change *notification*. Its VCS is *centralized* and supports *3-way merge* using a *change-based* comparison algorithm. *Explicit* branching is also provided.
7. **Client interface**: the *GUI* of CDO includes a collaborative concurrent *model editor*. A *framework* and an Eclipse-dependant *application* APIs are used to connect to the repository, manage it and access to persisted models.

3.2.5 MongoEMF

MongoEMF [Hun13] is a model persistence solution based on MongoDB and designed to provide a transparent persistence layer, supporting queries and integration on the Eclipse platform through OSGI. It has emerged from the EMF community as a personal research, attracting the attention of several developers for testing.

The architecture of MongoEMF is shown in Figure 3.12, and is composed of a client side and a server side which is a MongoDB database. The client side of MongoEMF implements the URIHandler EMF interface, which is used by the standard EMF Resource; this handler relies on two builders for converting DBObjects (i.e. MongoDB documents) into EMF EObjects and viceversa, using a converter for (un)marshalling datatypes. All the communications between the MongoDB database

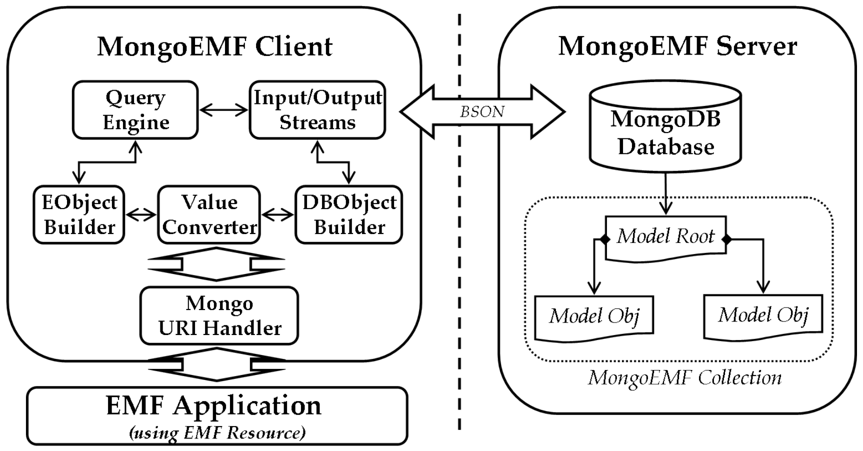


Figure 3.12 MongoEMF architecture

and the client are done in BSON [BSO10] through input and output streams. A model is represented in MongoEMF as a document for the root element and nested documents for all its contained objects; a single collection holds all stored models. Metamodels are not stored in the database, so they need to be manually registered in the Eclipse package registry; references between persisted model elements and their metaclasses are done through relative logical URIs.

Like CDO, a MongoEMF client accesses models using the standard EMF Resource interface, but the identification schema of MongoEMF hinders load on demand because it represents containment relationships using document nesting (a form of tight coupling); however, simple partial load on demand and incremental store can be achieved through a model partition mechanism that uses the EMF feature of cross-document referencing, handling remote references that are represented in the database as special proxy documents. MongoEMF provides simple queries through a dedicated API. Although a client can query for a single element, single load on demand is only possible when such element does not have any containment relationship, due to document nesting. The seven dimensions on model persistence are covered by MongoEMF as follows:

1. **Storage medium**: MongoEMF uses a *NoSQL* MongoDB database with a *tighly coupled* identification schema.
2. **Architecture**: the client-server architecture of MongoEMF is conformed by a *fat client* with *offline* server coupling and a database.
3. **Access**: MongoEMF supports *full load*, *full store*, both partial and full *update* and *coarse-grained* delete. Simple *incremental store* and *partial load on demand* are provided by the model partition described above. The triggering is either *implicit* or *explicit* without a prefetching algorithm. No transaction support is given and the local copy is *memory-based*.

4. **Query**: simple queries can be defined using a dedicated *API*.
5. **Transparency**: integration between MongoEMF and client applications is *customized*. Metamodels are represented in an *implicit* way that is not accessible by the client and *manual registration* is required. MongoEMF provides *total* semantic transparency.
6. **Version control**: MongoEMF does not support any version control feature.
7. **Client interface**: an EMF *framework* API, an Eclipse OSGI *application* interface and a *service* one are provided.

3.2.6 OOMEGA

OOMEGA [OOM12] is an all-in-one Eclipse-based platform that provides its own modeling framework with model transformations and various model persistence solutions: binary files, XML files and an object-oriented database repository; only the latter will be discussed in this chapter, since the other two do not add any special capability over XMI (see Section 3.2.1). Some EMF tools such as ATL [JABK08] have been adapted to interact with OOMEGA's own modeling framework.

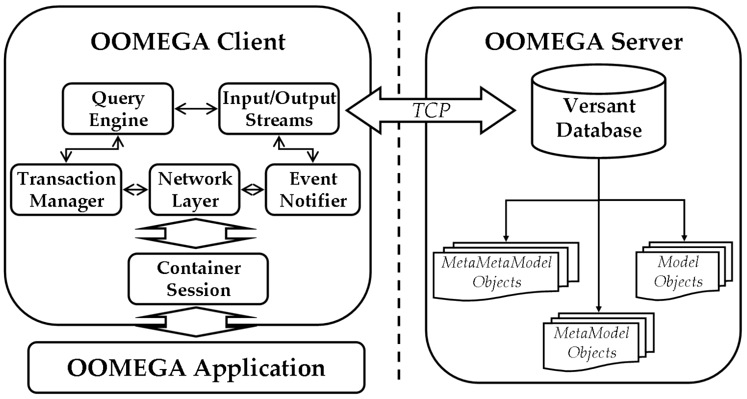


Figure 3.13 OOMEGA architecture

The architecture of OOMEGA is shown in Figure 3.13. An OOMEGA client application interacts with a session managed by a network layer that performs all the communications to and from a Versant [6] object database through input and output object streams and manages memory, object caching and prefetching. The network layer component can be omitted, being the client in charge of its duties in that case. A model is represented in OOMEGA as a set of objects in the Versant database with references to the metamodel, which is also represented in the database in the same way; even the meta-metamodel of OOMEGA is stored in the database homogeneously. However, manual metamodel registration is needed. References between

[6]Versant Object Database: http://www.versant.com

model elements are established as OIDs (i.e. object identifiers), which can be used to perform single or partial load on demand. OOMEGA also supports file-based model persistence.

A client accesses models using OOMEGA's model persistent interface, which supports all kinds of load, store, delete and update; since the modeling framework and the persistence solution have been designed together, the former has all the methods and data structures required by the latter, so their integration is transparent. Complex queries can be performed using a Java internal DSL whose syntax is similar to SQL, providing operations such as grouping and ordering. Although no version control is provided, an event notifier informs of changes done to the persisted objects by other clients. The seven dimensions on model persistence are covered by OOMEGA as follows:

1. **Storage medium**: OOMEGA uses an *object-oriented* database using a *loosely coupled* identification schema, although an ORM may be used over a *relational* one using a *tightly coupled* identification schema.
2. **Architecture**: a *fat client* with *online* server coupling and a database conform the client-server architecture of OOMEGA.
3. **Access**: All kinds of access are supported by OOMEGA with any kind of local copy. Load on demand is triggered in an *implicit* or *explicit* way.
4. **Query**: a Java *internal DSL* provides querying capabilities.
5. **Transparency**: the integration is *transparent* and the semantic transparency is *total*. Metamodels are represented in an *explicit* way and can be accessed in a *read & write* fashion, although *manual registration* is required.
6. **Version control**: OOMEGA does not provide any version control, but changes performed to persisted objects are notified to all clients.
7. **Client interface**: again, being the framework and the solution designed together, a *framework* API is used.

3.3 Comparison

Figures 3.14 to 3.20 show the comparison between the different model persistence solutions for each of the seven dimensions on model persistence. Greyed cells indicate the ocurrence of a feature. Each dimension is commented below.

The comparison on the *Storage medium* dimension is shown in Figure 3.14. On the one hand, XMI, ModelBus and EMFStore use file sets as their storage media; not surprisingly, they show the poorest capabilities in model access, mainly because their unit of work is a whole model or a model partition. On the other hand, CDO and OOMEGA may use multiple kinds of databases, but they are optimized to relationals and object-oriented ones, respectively; MongoEMF uses a MongoDB database with

DIMENSION	FEATURES				XMI	Model Bus	EMF Store	CDO	Mongo EMF	OOMEGA
Storage medium	Physical Model	File set	Encoding	Binary		▨				
				Textual						
				Markup	▨	▨	▨			
			Location	Local	▨					
				Remote	▨	▨	▨			
		Database		Relational				▨		▨
				NoSQL				▨	▨	
				Object-oriented				▨		▨
	Identification Schema			Loosely coupled	▨	▨	▨			▨
				Tightly coupled	▨	▨	▨	▨	▨	▨

Figure 3.14 Comparison between solutions for the *Storage medium* dimension

DIMENSION	FEATURES			XMI	Model Bus	EMF Store	CDO	Mongo EMF	OOMEGA
Architecture	No-server		Local	▨					
			Remote						
	Client-server	Server coupling	Offline	▨	▨	▨	▨	▨	
			Online						▨
			Fat client					▨	▨
			Thin client		▨	▨			
			N-layer				▨		

Figure 3.15 Comparison between solutions for the *Architecture* dimension

limited access capabilities. Figure 3.15 shows the comparison on the *Architecture* dimension. ModelBus and EMFStore are both thin clients, because they are based on and inspired by Subversion, respectively. CDO is the only n-layer architecture, which allows it to provide multiple synchronized repositories. Finally, MongoEMF and OOMEGA are fat clients, communicating with a raw database server. It is worth noting that only OOMEGA requires online server coupling, mainly because of its change notification mechanism.

DIMENSION	FEATURES				XMI	Model Bus	EMF Store	CDO	Mongo EMF	OOMEGA
Access	Local copy			File-based	▨	▨	▨			▨
				In-memory				▨	▨	▨
				Transaction support		▨	▨	▨		▨
	Store			Full store	▨	▨	▨	▨	▨	▨
				Incremental store					▨	▨
	Load			Full load	▨	▨	▨	▨	▨	▨
		Partial load	Granularity	Single load on demand				▨		▨
				Partial load on demand				▨	▨	▨
				Prefetching algorithm				▨		▨
			Triggering	Implicit				▨	▨	▨
				Explicit					▨	▨
	Update			Full update	▨	▨	▨	▨	▨	▨
				Partial update				▨	▨	▨
	Delete			Coarse-grained delete	▨	▨	▨		▨	▨
				Fine-grained delete				▨		

Figure 3.16 Comparison between solutions for the *Access* dimension

The *Access* dimension shows the big difference between file sets and databases:

DIMENSION	FEATURES			XMI	Model Bus	EMF Store	CDO	Mongo EMF	OOMEGA
Query	Query language	Standard	SQL				▨		
			OCL				▨		
		Dedicated	API					▨	
			Internal DSL						▨
			External DSL						

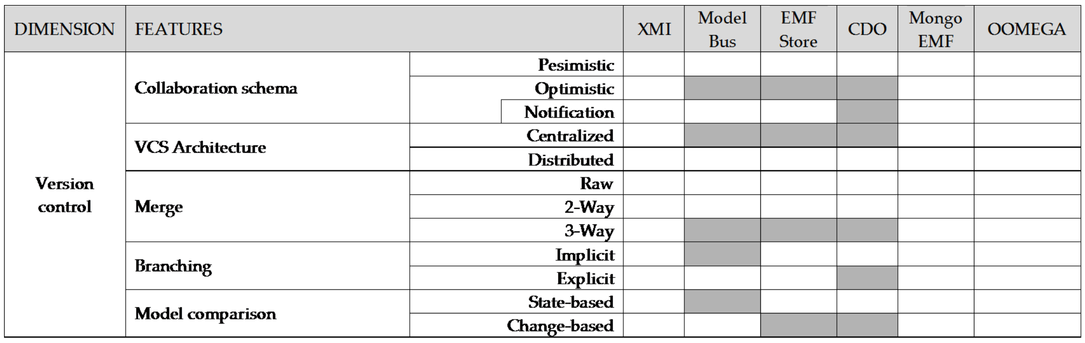
Figure 3.17 Comparison between solutions for the *Query* dimension

XMI, ModelBus and EMFStore only offer full store, load, update and delete, while the others support almost all operations. However, not all of the database repositories offer the same degree of compliance for each operation: for instance, the partial load on demand of MongoEMF is very limited, and CDO shows poor performance when managing large models [SZFK12]. The commented difference is even bigger in the *Query* dimension, being XMI, ModelBus and EMFStore incapable of any querying (see Figure 3.17). The *Transparency* dimension shows uneven results (see Figure 3.18): possibly the most relevant aspect to note is that the best results are achieved by OOMEGA, showing that a proper persistence support should be part of the design of a modeling framework. On the other side, the worst results are achieved by EMFStore, which may not even be capable of persisting some models because of its intrusive semantic transparency.

DIMENSION	FEATURES			XMI	Model Bus	EMF Store	CDO	Mongo EMF	OOMEGA
Transparency	Integration		Transparent	▨					▨
			Opaque		▨	▨			
			Customized				▨	▨	
	Metamodel management	Representation	Implicit			▨	▨	▨	
			Explicit	▨	▨				▨
		Availability	Read-only				▨		
			Read & write	▨	▨				▨
			Manual registration			▨	▨	▨	▨
	Semantic transparency		Total	▨				▨	▨
			Parallel		▨		▨		
			Intrusive			▨			

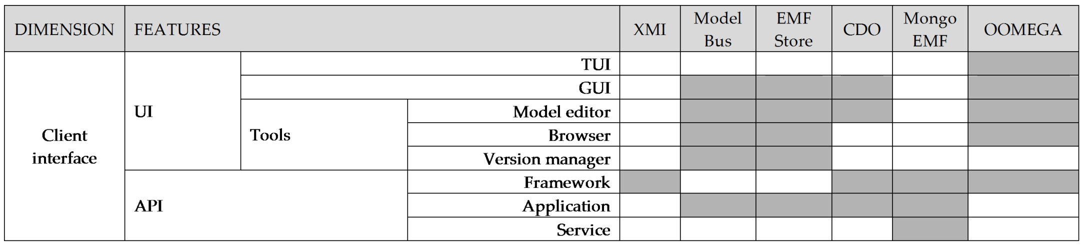
Figure 3.18 Comparison between solutions for the *Transparency* dimension

The comparison on the *Version control* dimension is shown in Figure 3.19. Due to their Subversion support or inspiration, ModelBus and EMFStore show good versioning capabilities, but CDO is unarguably the best. MongoEMF does not provide any version control, mainly because it is still a research prototype, while the others are stable commercial releases. It is worth noting that the OOMEGA database persistence does not provide version control; however, its file-based persistence may be combined with Subversion. The *Client interface* capabilities of each solution are related to their version control ones: ModelBus, EMFStore and CDO offer rich user interfaces (although the APIs of the former two are very poor), while MongoEMF offers a rich API, but no user interface. Finally, OOMEGA shows a rich set of user



DIMENSION	FEATURES		XMI	Model Bus	EMF Store	CDO	Mongo EMF	OOMEGA
Version control	**Collaboration schema**	**Pesimistic**						
		Optimistic		▨	▨	▨		
		Notification				▨		
	VCS Architecture	**Centralized**		▨	▨	▨		
		Distributed						
	Merge	**Raw**						
		2-Way						
		3-Way		▨	▨	▨		
	Branching	**Implicit**		▨				
		Explicit				▨		
	Model comparison	**State-based**		▨				
		Change-based			▨	▨		

Figure 3.19 Comparison between solutions for the *Version control* dimension



DIMENSION	FEATURES			XMI	Model Bus	EMF Store	CDO	Mongo EMF	OOMEGA
Client interface	UI		TUI						▨
			GUI		▨	▨	▨		▨
		Tools	Model editor		▨	▨	▨		▨
			Browser		▨	▨			▨
			Version manager		▨	▨			
	API		Framework	▨			▨	▨	▨
			Application		▨	▨	▨	▨	
			Service					▨	

Figure 3.20 Comparison between solutions for the *Client interface* dimension

interface tools and a complex API thanks to its framework-integrated design.

3.3.1 Categorization

As a result of the evaluation and comparison of the different persistence solutions, we have categorized them in four categories, depending on their purpose: default, user-oriented, application-oriented and integrated.

A *default* persistence solution is the one that is specified by the modeling framework; XMI falls into this category. Default solutions usually provide simple capabilities with no aim on advanced features such as scalability or versioning. However, they are the most transparent and normally all the other solutions must fit their interfaces to the default ones. OOMEGA may also fall into this category, but we preferred to define a special one for it.

A *user-oriented* persistence solution is designed to be used by a human client. User-oriented solutions offer rich user interfaces and versioning capabilities, although they usually consider models as atomic artefacts, so scalability and transparent integration are almost impossible to achieve. ModelBus and EMFStore clearly fall into this category; CDO is mainly user-oriented, but it also provides a framework API for application integration.

An *application-oriented* persistence solution is designed to be integrated with applications. It provides poor or no user interface, but a rich API instead. Thanks to their low-level persistence support, application-oriented solutions may perform better than the user-oriented ones, being faster and more scalable. MongoEMF is an application-oriented solution.

Finally, an *integrated* persistence solution is one that has been designed together with a modeling framework with special focus in advanced persistence solution capabilities such as version control and UIs; OOMEGA falls into this category. Integrated solutions have the potential to be both user-friendly and application-oriented, providing rich user interfaces and scalability; their actual capabilities depend completely on the decisions of their developers (e.g. OOMEGA does not support version control when using database persistence).

4 Running Example

See how they smile
like pigs in a sty,
see how they snied

In order to illustrate the design and operation of the model repository and model language presented in this thesis, as well as to perform their evaluation, a running example will be used in the following chapters. This chapter presents (i) the test metamodel, (ii) the test models and (iii) the test query for the running example.

4.1 Test metamodel

The running example is based on the reverse engineering case study of the Grabats 2009 contest [JS09]. This case study consisted in executing a particular query on five very large test models that represent 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, etc.; finally, the PrimitiveTypes package includes metaclasses that represent Java primitive types such as String or Integer.

4.2 Test models

We have considered the models proposed in the Grabats 2009 contest. They conform to the JDTAST metamodel. There are five models, from Set0 to Set4, each one containing its predecessor. Table 4.1 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.

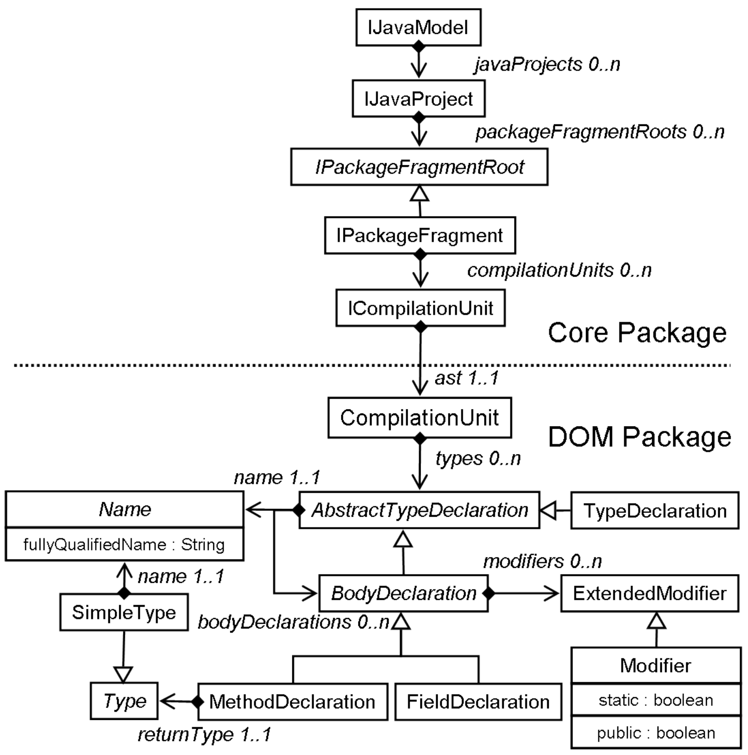


Figure 4.1 Grabats 2009 contest JDTAST metamodel simplification

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 4.1 Test models

4.3 Test query

The query proposed in the case study consists in obtaining every class (i.e. instance of TypeDeclaration) that declares a static, public method whose returning type is that same class. Figure 4.1 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 element) to the declaration of that type (TypeDeclaration element); the matching between both objects must be done by their name. Listing 4.1 shows the query for the test case using OCL. The query basically consists in the following steps for each TypeDeclaration element of the model:

1. Get the fullyQualifiedName of the Name element referenced by its name relationship.
2. Find at least one MethodDeclaration element referenced by its bodyDeclarations relationship that:
   a) has a Type element referenced by the returnType relationship and
   b) that Type element is a SimpleType and has a Name element referenced by the name relationship and
   c) that Name element has a fullyQualifiedName attribute that matches the fullyQualifiedName obtained in step 1 and
   d) that has two Modifier elements referenced by its modifiers relationship, one of them with a value of true for its static attribute and the other with a value of true for its public attribute.

```
TypeDeclaration. allInstances () ->
  select (td : TypeDeclaration | td.bodyDeclarations ->
    exists (bd : BodyDeclaration |
      bd.oclAsType(MethodDeclaration)
        .returnType.oclAsType(SimpleType).name
          .fullyQualifiedName = td.name.fullyQualifiedName
      and bd.modifiers ->
        exists (em : ExtendedModifier | em.oclAsType(Modifier). _static )
      and bd.modifiers ->
        exists (em : ExtendedModifier | em.oclAsType(Modifier). public )))
```

Listing 4.1 OCL query for the test case

We have chosen this test case for four reasons: (i) the test models are very large and capable of overloading the memory of a client application; (ii) the complexity of the metamodel requires type checking and casting due to its various type hierarchies, so the resulting query will be syntactically and semantically complex; (iii) these models have been extracted from the source code of real applications and (iv) the test query is a realistic example of the kind of access done by client applications such as model transformations.

5 Morsa

I am the eggman,
they are the eggmen

This chapter describes Morsa, our approach for model persistence. Morsa is a model repository that supports all the model persistence operations defined in Section 2.4; this chapter covers four of the five operations, since querying is covered in Chapter 8, as its support has been implemented as a dedicated DSL called MorsaQL.

The chapter is organized as follows: first, the design of Morsa is explained; then, the store, load, update and delete operations are covered; finally, the implementation of Morsa and its integration with EMF is described.

5.1 Design

As commented in the Introduction, 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 5.1.1.

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 5.1.2. Moreover, the architectural design is also involved in the goal of scalability since its components support the data design.

5.1.1 Architectural design

Morsa is based on a *fat client* client-server architecture, whose components are shown in Figure 5.1. 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. 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.

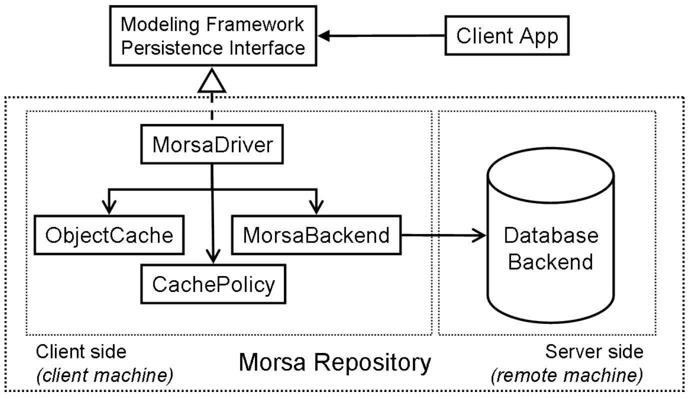

Modeling Framework Persistence Interface
Client App
MorsaDriver
ObjectCache
MorsaBackend
CachePolicy
Database Backend
Client side (client machine)
Morsa Repository
Server side (remote machine)

Figure 5.1 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, 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 a database 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.

5.1.2 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 2.3, 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 URI 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 5.2 shows an example of the representation of a model called javaModel in the Morsa repository. On the left side, an instance of the *JDTAST* metamodel (see Figure 4.1) 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.

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 2.3).
iv. *Ancestors*: a list of MorsaReferences to the MorsaObjects that represent the ancestors of the model element (see Section 2.3).
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*

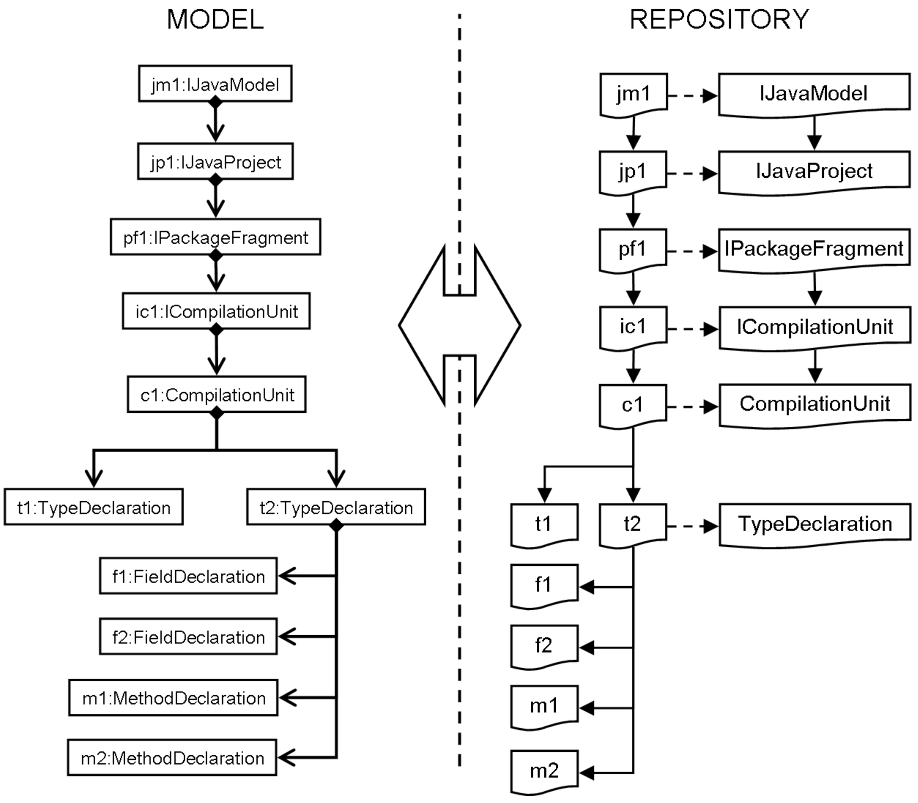


Figure 5.2 Example of repository persistency for the running example

features represent the structure of the object graph and are used for partial loading as explained later in Section 5.3.1.2.

A MorsaReference is a smart reference (i.e. an usage of the *Proxy* pattern [GHJV95]) that is composed of, at least, the *MorsaID* of the referenced element, its *Metatype* and the MorsaCollection that holds it, i.e. its containing model; depending on the implementation of MorsaBackend being used, it may contain additional information.

Figure 5.3 shows the internal structure of the MorsaObjects that represent the elements t2 and TypeDeclaration of the model and metamodel, respectively, shown in Figure 5.2. 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

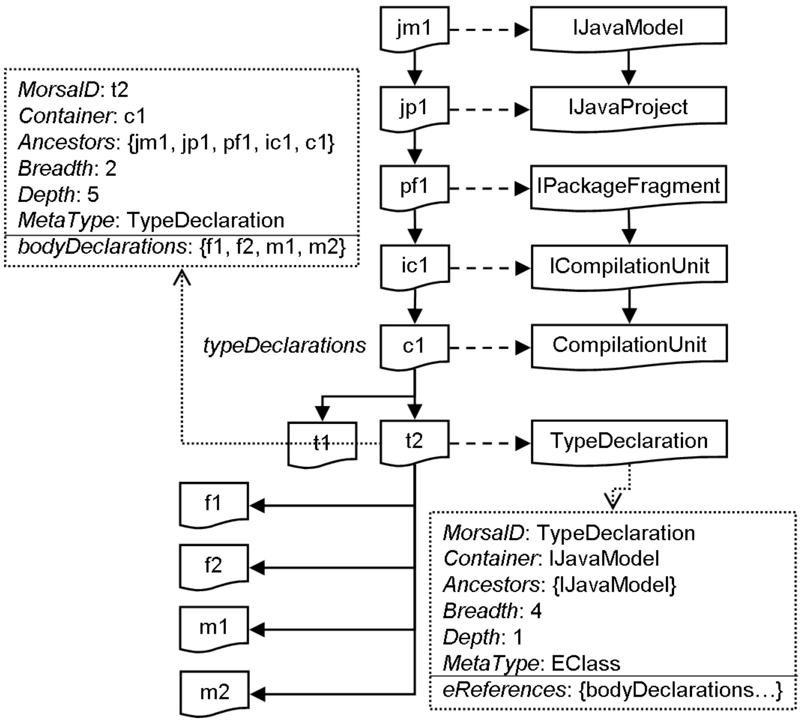


Figure 5.3 Internal structure of two MorsaObjects

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 5.4 shows how the index object references the root objects of the three packages (Core, DOM and PrimitiveTypes) defined in the metamodel of Figure 4.1. Note that the index object points to the jm1 model element, which is the root of the javaModel1 model, described in Figure 5.2.

MorsaObjects provide the client-side a way to transfer model elements from/to a Morsa repository that is backend-independent. Figure 5.5 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 sends a request (r) to the repository for 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.

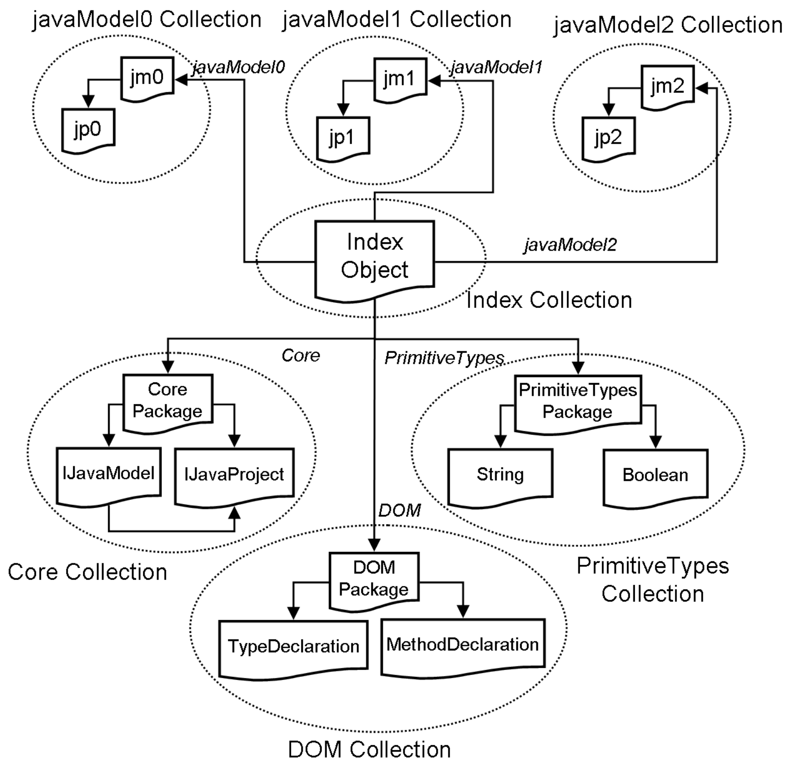


Figure 5.4 Collections contained by the repository for the *JDTAST* metamodel packages and three sample models

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 5.2 and 5.3, respectively. The following sections describe the algorithms for the store, load and update operations on models and model partitions, as defined in Section 2.3. 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.

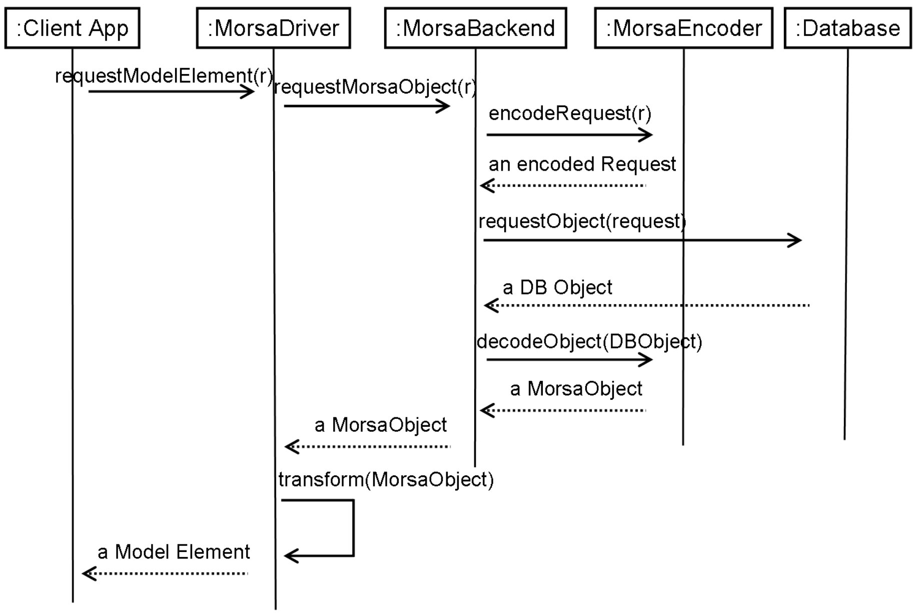


Figure 5.5 Simplification of the interaction between client-side components for loading a model element

5.2 Model store

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.

5.2.1 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 stored yet 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 5.1.2) 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.

5.2.2 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 5.3. 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 repositor: on the one hand, if the driver has been connected to the repository over all the incremental store process and still is, it calculates the difference between the save cache (i.e. the model elements that have a *MorsaID* assigned to them) and the ones 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 travesing 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).

5.3 Model load

This section is dedicated to the load operation on models, as described in Section 2.4. 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.

5.3.1 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 from loading them again. The selection and configuration of each scenario is done by the client application by parametrization of the MorsaDriver.

5.3.1.1 Full load

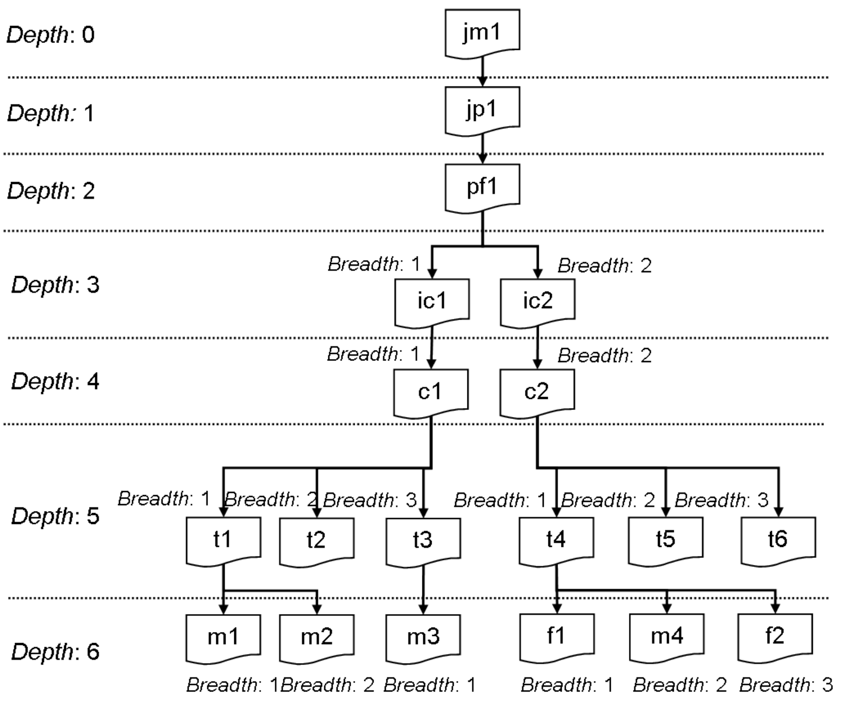
Consider a small or medium-sized model that can be kept in memory by a client application. If the whole model is going to be traversed, it would be a good idea to load it once, hence saving communication time with the persistence backend. We call this scenario *full load* and this is the way EMF works 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.

5.3.1.2 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 at 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 5.6(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 5.6 (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.



(a)

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)

Figure 5.6 Partial load on demand in the running example: a) model b) object cache before partial load c) object cache after partial load

5.3.2 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.

5.3.3 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) preventing the driver from loading objects that have already been 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 5.6 (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 5.6 (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 5.6 (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 avoid 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 was 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 5.4.

5.3.4 Cache replacement policies

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

- An 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.
- An 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 5.6, an LIFO policy would unload the objects t1...t6, while an FIFO policy would unload jm1, jp1, pf1, ic1, ic2, and c1.
- An 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. An LRU policy would be equivalent to a FIFO one for depth-first and breadth-first traversals.
- An 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).

5.4 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 2.4, 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 5.2.1 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 [SBPM08] and MOF [MOF06], which are the most widely used metamodels.

Figure 5.7 shows en example of an update; modifications done on the source model (left side) are: (i) TypeDeclaration t2 is moved from CompilationUnit c1 to CompilationUnit c2 and (ii) 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 5.7 (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 5.2.2. When some model elements are deleted rather than the entire model, a model update is performed instead.

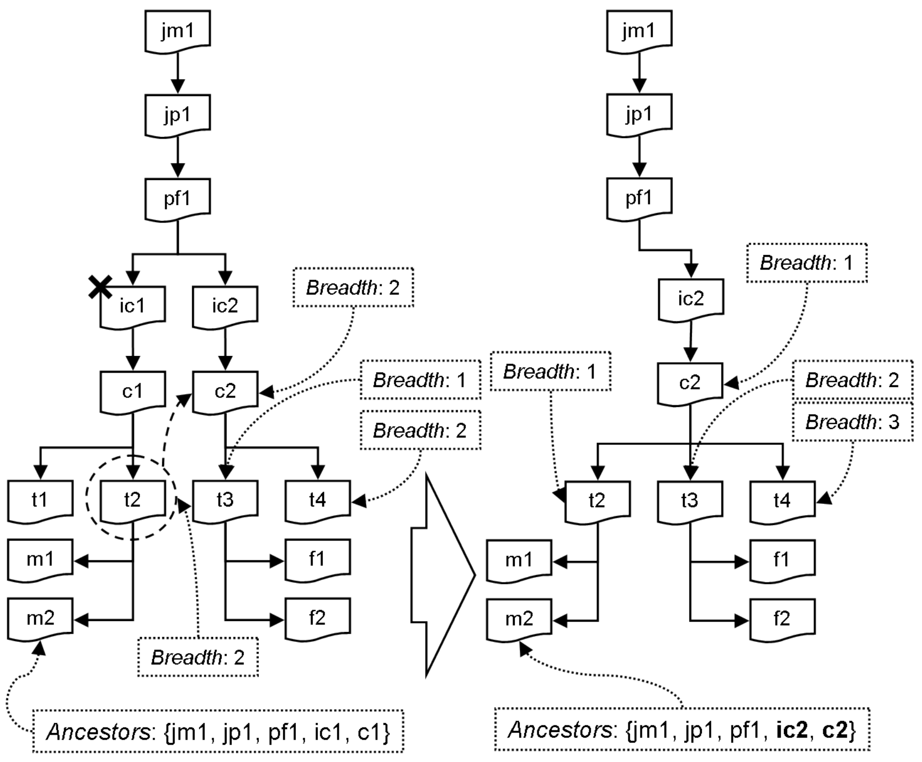


Figure 5.7 Modifications and deletions over the running example

5.5 Integration and implementation

Morsa is intended to be integrated with modeling frameworks and their applications (e.g. model transformation engines). Our current prototype [Mor13] is integrated with EMF. As commented in Section 5.1.1, a transparent way to achieve integration is to design the MorsaDriver as an implementation of the persistence interface of the modeling framework (Resource in EMF); this is done in our prototype using a MorsaResource class that is connected to the MorsaDriver, which delegates the implementation of all the Resource methods to the driver. 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 [CDO12][KH10][HRW09]. 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 [CDO12] 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 5.3.1.2), 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 5.1.2, 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 [Cou] and MongoDB [Ban11], since they are the most relevant document-based NoSQL databases. On the one hand, CouchDB consists of a flat address space of JSON [JSO] documents that can be selected and aggregated using JavaScript in a Map/Reduce [DG04] 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 [BSO10] 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.

6
Evaluation of Morsa

See how they run
like pigs from a gun,
see how they fly

This chapter evaluates our approach in terms of memory and time consumption and compares it with CDO, the most widely used model repository (see Section 3.2.4) for EMF and XMI (see Section 3.2.1), the default persistence support for EMF. The evaluation has been made using the running example presented in Chapter 4; the evaluation of the query test case is covered in Chapter 9.

The organization of this chapter is as follows: first, the test benchmarks are introduced; then, the results for every test benchmark are presented and commented; finally, an overall assessment is made, explaining the main differences between the performance results of each persistence solution.

6.1 Test benchmarks

We have built a different benchmark for the operations of store, load, update and delete, as defined in 2.4 (the query operation is discussed in Chapter 7 and evaluated in Chapter 9):

1. 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.
2. 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.

3. The *model update* benchmark consists in executing the Grabats 2009 test query described in Chapter 4 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.
4. 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.

6.2 Results

Each benchmark has been executed using the EMF XMI loading facility, a CDO repository in legacy and native mode (see Section 3.2.4) 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 empyrical 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.

6.2.1 Model store

Table 6.1 shows the results of the model store benchmark. The incremental store scenario has been tested with incremental store as described in Section 5.2.2 (*Inc* mode) and also as described in Section 5.4 (*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.

6.2.2 Model load

Table 6.2 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.

6.2.3 Model update

Table 6.3 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.

6.2.4 Model delete

Table 6.4 shows the results of the model delete benchmark. Since the current Morsa prototype uses MongoDB, the deletion of a model consists in 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.

6.3 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 test case. 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 peroperly is a must.

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 this assymetrical conditions instead of storing the XMI files on a remote server because the latter is not a realistic scenario and 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 scope of this thesis, but its is worth noting them because their implementation may had have 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.

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 6.1 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 6.2 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 6.3 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 6.4 Performance results of the model delete benchmark

7 Model Querying

Corporation teeshirt,
stupid bloody tuesday

This chapter presents the problem of model querying and how it is addressed by a selected set of model querying approaches for EMF. The organization of this chapter is as follows: first, the motivation of querying models is explained; then, the OCL standard is briefly introduced; a set of dimensions that will describe and evaluate both the selected model querying approaches and our own one (which is described in Chapter 8) are defined; an example query is presented afterwards in order to help visualize the different approches; finally, each model querying approach of the selected set is presented and described in terms of the example query.

7.1 Motivation

Model querying is the process of applying the query operation defined in Section 2.4 to search for a set of model elements that satisfy a given condition inside a (meta)model. The simplest way to query a model is to visually inspect it using a visualization application like a tree or graph editor; this can be done by human users. A further step would be using code that navigates the model while checking the conditions on the navigated elements. Finally, a navigational approach such as XPath [XPa99] can be used by both human users and client applications to make the traversal of models easier.

As the acceptance of MDE grows, the former techniques have to be applied to tackle more challenging problems that require managing complex and large models like those that represent source code [JS09], software architectures [ADM07] or communication networks [SZFK12], which make visual inspection almost unfeasible; moreover, the complexity of the search patterns used by applications such as model transformations is now beyond simple navigation rules and raw code querying seems too verbose and hard to maintain to be considered.

Model querying has arisen as a relevant research topic in the area of MDE, as there is a substantial amount of work done on such purpose (Section 7.5 describes some approaches). Model querying is related to other MDE research areas as it can be a foundation where techniques for model visualization, model comparison and model versioning rely to simplify their implementations and make them more efficient.

7.2 The OCL standard

OCL [OCL06a][WK03] is a declarative language defined by the OMG as an add-on to UML in order to complete UML models with information such as constraints, queries and derived attributes; moreover, OCL can be applied to any MOF-compliant model. The current version of OCL (2.3.1.) has been formally published as an ISO standard (ISO/IEC 19507). In the context of MDE, OCL can be used as both a navigation language for M2M and M2T transformations and a constraint language for defining well-formedness rules for metamodels. Several implementations of OCL (or OCL-like languages) have been built to be used in MDE applications (e.g. OCLinEcore for Ecore models [OCL06b]). However, OCL is not supported by almost any storage medium for models, so a translation step is usually needed; given the complexity of OCL, this reduces its performance.

7.3 Dimensions on model querying

We have identified four key dimensions that a querying approach should address: effectiveness, usability, safeness and efficiency. These dimensions summarize some widely used criteria for evaluating and comparing programming languages. They have also helped us evaluating the different model querying approaches (see Section 7.5 and Chapter 9).

We define the **effectiveness** of a querying approach as its ability to perform complex queries with the least actual queries executed against the persistent storage. This dimension summarizes the following evaluation criteria for programming languages:

- *Expresiveness*: a querying approach should support every needed feature.
- *Preciseness*: a querying approach should be able to handle slight variations in a uniform and simple way. For instance, object-oriented features such as polymorphism.

We measure the effectiveness of a querying approach as the amount of separated queries that must be performed in order to retrieve the desired model elements from a persistence solution: the less queries that are needed, the more effective the language is. Each one of these queries (or subqueries) provides the other queries intermediate results that, when combined, produce the desired result; each one of this subqueries must be executed separetely. Some features that provide greater effectiveness are nested subqueries and named variables (i.e. variables that hold the result of an operation and can be accessed by later operations).

The **usability** of a querying approach is related to making reading and writing a query easier. This dimension summarizes the following evaluation criteria for programming languages:

- *Readability*: a query should be easily understood by anyone just by knowing the involved metamodel.
- *Writability*: queries should be written in a clear, concise, quick and correct manner.
- *Consistency*: a querying approach should use a syntax that is similar to the one of a well-established query language that users are familiar with.

We measure the usability of a querying approach as (i) the amount of declarative (the less,the better) and imperative (the more, the worse) statements, (ii) the absence of explicit type castings and checks and (iii) non-query statements that may appear mixed with the query ones (e.g. Java constructors) and (iv) the resemblance to a well-established query language.

The **safeness** of a querying approach is related to the support for compilation or interpretation given by the approach and its integration capabilities on existing or new applications. This dimension summarizes the following evaluation criteria for programming languages:

- *Reliability*: a querying approach should enforce its syntactic and semantic rules prior to the execution of queries so no unexpected errors occur.
- *Provability*: the process of a query should be formally verifiable through the analysis of that query.
- *Portability*: a querying approach should be usable in different applications without requiring major changes in them.

We measure the safeness of a querying approach as (i) the enforcement of syntactic and (ii) semantic rules of the language in design time, (iii) the independence from any external tool rather than linked code to create, edit, execute or manipulate queries in any way and (iv) the existence of a formal basis supporting the querying approach.

The **efficiency** of a querying approach is related to the amount of resources that it consumes when executing a query. This dimension represents basically the *efficiency* evaluation criterion for programming languages. We measure the efficiency of a querying approach as the (i) time and (ii) memory needed for executing a query. This is possibly the hardest dimension to address, since it strongly depends on the underlying persistence solution and usually the more effective and usable a language is, the less efficient it may be (as it has to deal with complex querying structures).

Our main goal is to develop a querying approach for Morsa that is focused on efficient and usable querying which can be easily integrated with client applications aciheving the most possible effectiveness. Section 7.5 describes some already existing model querying approaches, while Chapter 8 shows the query language that we have developed for Morsa.

7.4 Example query

An example query will be used on the following sections to illustrate how the different approaches that have been analyzed in this thesis cover the four dimensions identified in Section 7.3. We have chosen a different example than the one presented in Chapter 4 because its simplicity is better suited for describing each querying approach, using short and readable examples. The runnnig example presented in Chapter 4 will be used in the evaluation of MorsaQL and the selected querying approaches in Chapter 9.

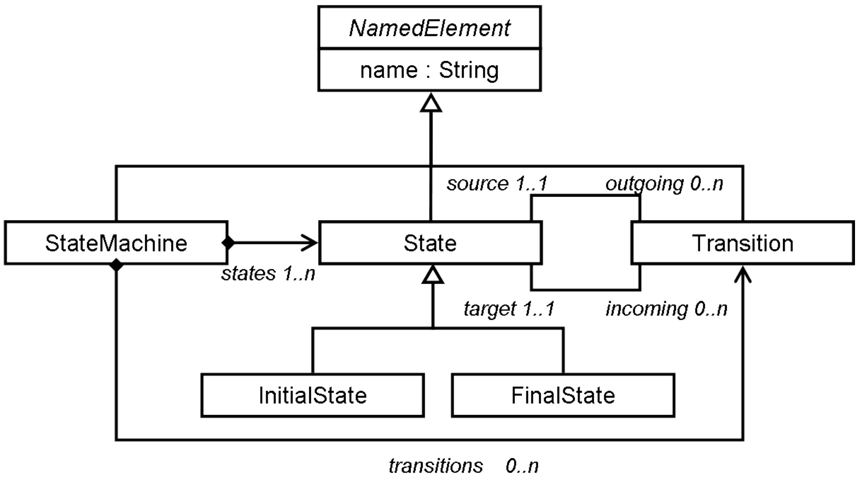


Figure 7.1 State machine metamodel

The example query consists of a language for the definition of simple state machines, which is a canonical example that is frequently used in the literature. Figure

7.1 shows the abstract syntax of this language. A StateMachine holds states and transitions. States can be regular (State), initial (InitialState) or final (FinalState). The Transitions go from a source state to a target state; the opposite relationships to source and target are outgoing and incoming, respectively. All the metaclasses have a name attribute. An example of a model that conforms to this metamodel is shown in Figure 7.2 using the UML notation for state machines; the example describes the basic operation of a simple compiler that reads source code, checks its syntax and semantics and finally generates binary code. Note that the final state can be reached from three different states following three transitions: two errors and one successful step.

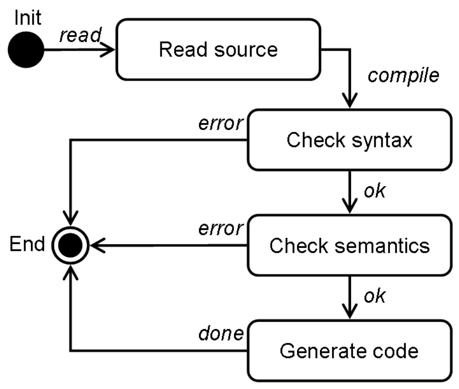


Figure 7.2 State machine model for a simple compiler

Along with the metamodel and the model, we have defined a query to help us describe the syntax of each of the querying approaches we have analyzed. It consists in obtaining every state that could yield to an error, that is, every State that is the source of a Transition whose target is a FinalState and whose name is *error*. For the example model, this query would find the Check syntax and Check semantics states. Note that an error transition could have also been represented as a child class of Transition, but we have not used this design in order to show queries that handle both attributes and references. The OCL specification of this query would be (being self the StateMachine object):

```
self . states—>select(state | state .outgoing —> exist(
  transition | transition .oclIsKindOf( FinalState ) and ( transtion .name = "error")
  )
)
```

Listing 7.1 OCL specification for the example query

7.5 Existing querying approaches for EMF

Several approaches have been defined for querying EMF models. In this section, some of them will be presented and described. We have chosen five approaches that include plain code, two implementations of OCL, an API and an external DSL (see Section 2.5); this choice has been made following the concerns of integration in client applications, efficiency and use by the modeling community.

The following subsections describe each approach in terms of their syntax (both abstract and concrete) and their semantics (i.e. the way they calculate queries) using the example query to illustrate them. Since APIs are not strictly languages, we will consider that their classes and relationships are their abstract syntax and that their concrete syntax are the way their instances are created and interact through method parameterization and invocation. The characterization of each approach using the dimensions defined in Section 7.3 will be done in Chapter 9.

7.5.1 Plain EMF

The simplest approach to model querying in EMF is to use plain EMF Java code. Both generated and dynamic EMF classes provide methods to navigate models following the relationships between their elements: the former classes provide dedicated, strongly-typed methods while the latter make use of the reflective API of EMF. Virtually any query can be performed using this approach (and usually other approaches rely on it under the scenes) and packed inside a method or a class. However, even using methods or classes to organize and modularize the code, a query could become too complex to be easily written; moreover, type checking and casting are mandatory when using dynamic EMF, and even with generated models they are necessary when dealing with type hierarchies, making code even more difficult to write, read and maintain. On the other side, this approach benefits itself from all the operations and control structures that Java provides, e.g. loops, conditionals, recursion, etc.

Listing 7.2 shows the Java code for implementing the example query in plain EMF using both dynamic and generated EMF. The generated EMF code is briefer, clearer and safer since the strong typing of the objects prevents the use of reflective methods and type checks. As can be seen, a query in plain EMF is just a matter of navigating through relationships and checking constraints using plain Java code.

```
//With dynamic EMF
Collection <EObject> states = new LinkedList<EObject>();
EObject stateMachine = (EObject)resource.getContents().get(0);
List <EObject> transitions = stateMachine.eGet(
  stateMachine.eClass(). getEstructuralFeature (" transitions "));

for (EObject transition : transitions ) {
  EObject target = (EObject) content.eGet(
 9      content.eClass().getEStructuralFeature("target"));
10    if (target.eClass().getName().equals("FinalState")) {
11      String name = (String)content.eGet(
12        content.eClass().getEstructuralFeature("name"));
13      if (name.equals("error")) {
14        EObject source = (EObject)content.eGet(
15          content.eClass().getEstructuralFeature("source"));
16      states.add(source);
17      }
18    }
19 }
20
21 //With generated EMF
22 Collection<State> states = new LinkedList<State>();
23 StateMachine stateMachine = (StateMachine)resource.getContents().get(0);
24
25 for (Transition transition : stateMachine.getTransitions()) {
26   State target = transition.getTarget();
27   if (target instanceof FinalState) {
28     if (transition.getName().equals("error")) {
29       states.add(transition.getSource());
30     }
31   }
32 }
```

Listing 7.2 Example query with plain EMF

The above listed code does the following:

1. An empty list is created to accomodate the results of the query. Note that for dynamic EMF, the list is of kind EObject, while for generated EMF, the list is of kind State (lines 2 and 18).
2. The root of the model (i.e. the StateMachine object) is obtained from the resource variable, which is an instance of the Resource EMF interface used to represent a model storage (lines 3 and 19).
3. A list containing all the Transitions is obtained using a reflective method (eGet) or a strongly-typed one (getTransitions) for dynamic and generated EMF, respectively (lines 4 and 21).
4. The list of Transitions is iterated (lines 6 and 21), obtaining their target States (lines 7 and 22) and checking that they are of instances of FinalState (lines 8 and 23).
5. The name attribute of the Transition is obtained (lines 9 and 24) and, if it is equal to *error* (lines 10 and 24), the source state of the Transition is obtained (lines 11 and 25) and stored in the list (lines 12 and 25).

The architecture of a querying approach based on plain EMF is basically the one of the EMF framework, because there are no additional features built on top of it, although it can use modularization mechanisms such as method and class encapsulation.

7.5.2 EMF Query

EMF Query [EMF12] is an EMF component for model querying released in 2005 as an Eclipse Project. It is designed as an API that somehow resembles an internal DSL (see Section 2.5), since it has classes named after SQL keywords (SELECT, FROM and WHERE) but it cannot be considered as one, because there is too much *syntactic noise* (i.e. presence of syntactic elements from the host language, such as initializer methods) on its concrete syntax. Figure 7.3 shows the abstract syntax of EMF Query, which involves the following classes:

- The SELECT class represents query statements and is the entry point from which queries are built. An instance of this class is formed by a FROM and a WHERE objects and stores the maximum number of results to be found.
- The FROM class is used to specify the *search scope* of a query, i.e. the root object from which the query starts to search for results. It is internally represented as an IEObjectSource. Such a search scope may be a single EObject or a collection of them.
- The WHERE class is used as the start point of the conditions that must be satisfied by the results of the query.
- The Condition class represents the conditions that the query must check. It is the root of the hierarchy of conditions. Additional conditions can be created and linked using the AND and OR methods.
- The EObjectCondition class represents the conditions that a model element must satisfy. Its subclasses EObjectAttributeValueCondition, EObjectReference-ValueCondition and EObjectTypeRelationCondition check attribute and reference values and type conformance, respectively. An EObjectAttribuveValueCondition uses a DataTypeCondition object for the actual value check.
- The DataTypeCondition represents conditions applied to primitive values such as strings or integers. It has subclasses for the different EMF primitive types and their operations (e.g. greater than, equals, etc.); these are not shown for the sake of clarity.
- The IQueryResult interface represents the results from a query as an EObject collection.

Listing 7.3 shows the source code that specifies the query from the example query using the concrete syntax of EMF Query. There are four things worth noting in it:

- A query is created as a SELECT object that is passed its components objects (FROM and WHERE) as parameters of its initialization method, as seen in lines 12 to 18 and 25 to 33.

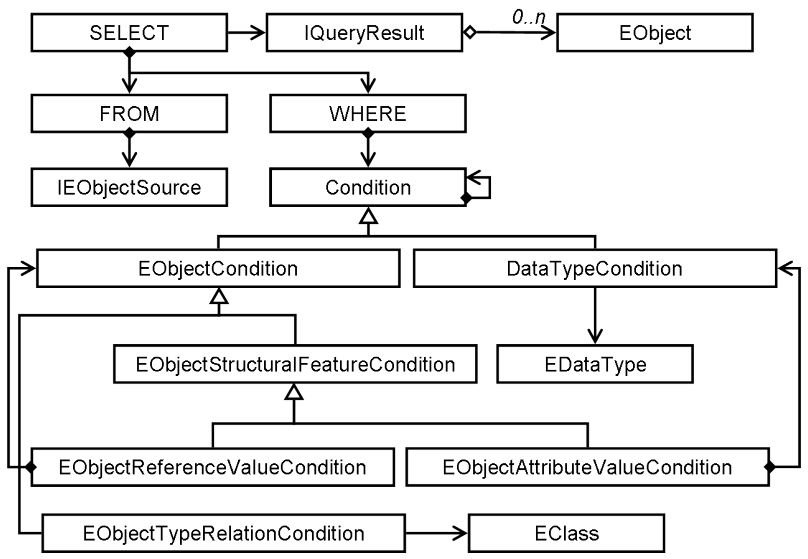

SELECT
IQueryResult
0..n
EObject
FROM
WHERE
IEObjectSource
Condition
EObjectCondition
DataTypeCondition
EObjectStructuralFeatureCondition
EDataType
EObjectReferenceValueCondition
EObjectAttributeValueCondition
EObjectTypeRelationCondition
EClass

Figure 7.3 Abstract syntax of EMF Query

- EObjectReferenceValueCondition objects are used to navigate through the relationships between model elements by specifying first the relationship to navigate and then the condition to filter navigated elements, as seen in lines 14 and 15.
- There is little difference in the size of the query code between dynamic EMF and generated EMF because of the use of EReference and EAttribute objects to specify the structural features that are being checked, as seen in lines 27 and 32.
- Using EReference and EAttribute objects imply the need of obtaining the EClass objects for states, transitions and final states on advance when using dynamic EMF, as seen in lines 7 to 10. This makes the code longer.

```
 1 //With dynamic EMF
 2 EObject stateMachine = (EObject)resource.getContents().get(0);
 3 List<EObject> states =
 4   (List<EObject>)stateMachine.eGet(stateMachineClass.eClass()
 5     .getEstructuralFeature("states"));
 6
 7 EPackage stateMachinePackage = stateMachine.eClass().getEPackage();
 8 EClass stateClass = (EClass)stateMachinePackage.getEClassifier("State");
 9 EClass  transitionClass  = (EClass)stateMachinePackage.getEClassifier("Transition");
10 EClass  finalStateClass  = (EClass)stateMachinePackage.getEClassifier("FinalState");
11
12 SELECT select = new SELECT(Integer.MAX_VALUE,
13   new FROM(states),
14   new WHERE(new EObjectReferenceValueCondition ((EReference)stateClass.getEstructuralFeature("outgoing"),
15         new EObjectReferenceValueCondition( (EReference) transitionClass.getEstructuralFeature("target"),
16           new EObjectTypeRelationCondition(finalStateClass)).AND(
17             new EObjectAttributeValueCondition((EAttribute) transitionClass.
                getEstructuralFeature ("name"), new StringValue("error"))))));

IQueryResult  result  = select.execute();

//With generated EMF
StateMachine stateMachine = (StateMachine)resource.getContents().get(0);

SELECT select = new SELECT(Integer.MAX_VALUE,
  new FROM(stateMachine.getStates()),
  new WHERE(new EObjectReferenceValueCondition(StateMachinePackage
          .getState_Outgoing(),
        new EObjectReferenceValueCondition(StateMachinePackage.
          getTransition_Target (),
          new EObjectTypeRelationCondition( finalStateClass )).AND(
            new EObjectAttributeValueCondition(StateMachinePackage.
              getNamedElement_Name(), new StringValue("error"))))));

IQueryResult  result  = select.execute();
```

Listing 7.3 Example query with EMF Query

The query engine of EMF Query is rather simple: the SELECT object gets the search scope from the FROM object and then iterates over all its descendants, checking the conditions specified under the WHERE object for each one of them, which is more or less what plain EMF does, so EMF Query can be considered as a facade to simplify the coding of queries by presenting the client a more declarative interface.

7.5.3 MDT OCL

MDT OCL [OCL12] (a.k.a. Eclipse OCL) is an Eclipse project dedicated to the implementation of the OCL standard language (see Section 7.2) for the EMF modeling framework, both for model querying and for specifying restrictions on metamodels. MDT OCL provides an external DSL and an API named OCLInEcore and OCLInterpreter, respectively, to define OCL queries and constraints. OCLInEcore provides a text editor built with XText [EB10] that can be used within the metamodel editor of Eclipse to define and evaluate OCL constraints and queries. In this thesis we will consider OCLInterpreter, which is the API that can be used by client applications to define OCL queries at runtime.

The specification for the abstract and concrete syntax details of OCL can be found in [OCL06a]. The abstract and concrete syntax of the OCLInterpreter API are described below. Figure 7.4 shows the abstract syntax of the OCLInterpreter API, involving the following classes:

- The OCL class represents the MDT OCL engine, acting as a facade for the OCL parser and evaluator. It has to be initialized with an EnvironmentFactory that provides its Environment.
- The Environment class represents a metamodeling framework, as MDT OCL may also work with UML models. EcoreEnvironment is the environment for

EMF and is provided by the EcoreEnvironmentFactory, which is a subclass of EnvironmentFactory. It also stores the context of a query, i.e. the type of its self OCL variable.

- The OCLHelper class encapsulates the OCL parser and is obtained from an OCL object configured for a certain environment. When supplied an EClassifier as its context, it can parse text queries, generating OCLExpression objects.
- The OCLExpression class represents parsed OCL expressions, which may be constraints or queries.
- The Query class represents the OCL evaluator, which evaluates expressions represented as OCLExpressions against a given Environment, producing a collection of objects as a result, which may be model elements or plain Java objects. In order to evaluate the query, it must be passed an EObject that acts as the self variable of OCL.

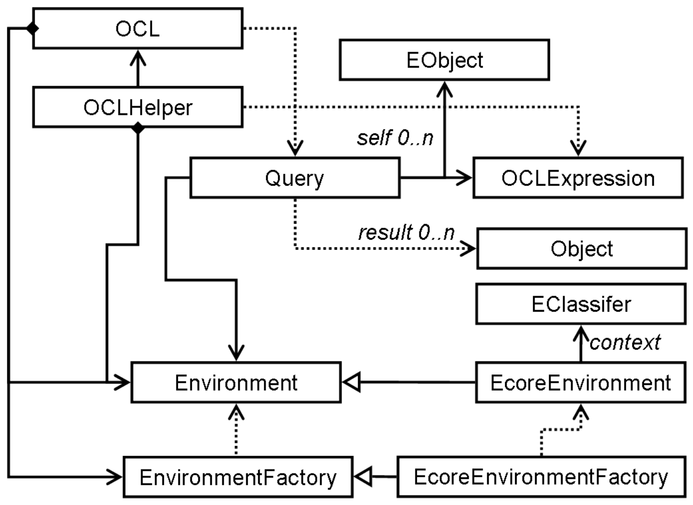


Figure 7.4 Abstract syntax of the OCLInterpreter MDT OCL API

Listing 7.4 shows the implementation of the running example query using the MDT OCL approach. Note that the only differences between dynamic and generated EMF are the first and the last lines (lines 2 and 16 for dynamic EMF and lines 19 and 33 for generated EMF), because the parsing done by OCLHelper automatically resolves the class names for dynamic EMF.

```
1 //With dynamic EMF
2 EObject stateMachine = (EObject)resource.getContents().get(0);
3 EClass stateMachineClass = (EClass)stateMachinePackage.getEClassifier("StateMachine");
4
5 OCL<?, EClassifier, ?, ?, ?, ?, ?, ?, ?, Constraint, EClass, EObject> ocl =
      OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);
6
```

```
 7 OCLHelper<EClassifier, ?, ?, Constraint> helper = ocl.createOCLHelper();
 8 helper.setContext(stateMachineClass);
 9
10 OCLExpression<EClassifier> expression = helper.createQuery(
11   " self . states -> "
12     + "select(st : State | st.outgoing -> "
13       + "exists(t : Transition | t.target.oclIsKindOf( FinalState ) "
14         + "and t.name = 'error'))"
15   );
16
17 Query<EClassifier, EClass, EObject> query = ocl.createQuery(expression);
18 Collection <EObject> result = (Collection<Object>)query.evaluate(stateMachine);
19
20 //With generated EMF
21 StateMachine stateMachine = (StateMachine)resource.getContents().get(0);
22
23 OCL<?, EClassifier, ?, ?, ?, ?, ?, ?, ?, Constraint, EClass, EObject> ocl =
       OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);
24
25 OCLHelper<EClassifier, ?, ?, Constraint> helper = ocl.createOCLHelper();
26 helper.setContext(stateMachine.eClass());
27
28 OCLExpression<EClassifier> expression = helper.createQuery(
29   " self . states -> "
30     + "select(st : State | st.outgoing -> "
31       + "exists(t : Transition | t.target.oclIsKindOf( FinalState ) "
32         + "and t.name = 'error'))"
33   );
34
35 Query<EClassifier, EClass, EObject> query = ocl.createQuery(expression);
36 Collection <State> result = (Collection<State>)query.evaluate(stateMachine);
```

Listing 7.4 Example query with MDT OCL

The above code listing does the following:

1. An OCL instance is created passing it an instance of the EcoreEnvironmentFactory class (lines 5 and 23).
2. An OCLHelper instance is created (lines 7 and 8) and its context is set to the EClass that represents the StateMachine metaclass (lines 25 and 26).
3. An OCLExpression instance is created, passing in the string that specifies the OCL query (lines 10 to 15 and 28 to 33).
4. A Query instance is created using the OCLExpression object (lines 17 and 18) and the result is obtained by invoking its evaluate method (lines 35 and 36).

The MDT OCL engine parses OCL expressions and builds their abstract syntax trees. Then, using a visitor pattern, it evaluates the expressions against the model elements. MDT OCL uses knowledge from the metamodel to narrow the search space and avoid traversing entire model partitions as plain EMF and EMF Query do.

7.5.4 IncQuery

IncQuery (or EMF-IncQuery) [BHH$^+$12] is a framework for defining declarative queries against EMF models. Its query language is implemented as an external DSL generated with XText. This DSL allows queries to be represented as graph patterns that are evaluated using a variation of the Rete algorithm [For82], which computes models as graphs. The novel feature of IncQuery is that it provides an *incremental querying* mechanism that tracks changes in model elements and then adapts its representation of the already executed queries so that they do not have to be fully evaluated again. IncQuery generates Eclipse plugins for each query that have to be registered in the Eclipse Platform; although this thesis is focused on approaches that can be seemlessly integrated within the code of client applications, we have selected this approach because of its focus on query optimization.

The abstract syntax of IncQuery is shown in Figure 7.5, including the following classes:

- A Package is a collection of Patterns defined for a given set of EPackages which it must Import. This is the root of IncQuery's abstract syntax.
- A Pattern is a collection of conditions that must be satisfied by a set of parameters (i.e. model elements), which are represented as Variables typed by EClasses. However, Variables may be defined inside Constraints using dynamic typing (i.e. without declaring their types).
- A Constraint is a condition that must be satisfied by a Variable. This is the root of a hierarchy of conditions that perform specialized tasks such as checking type conformance (EClassConstraint for classes and EDataTypeConstraint for primitive types) and equality or unequality between variables (MatchingConstraint).
- A RelationConstraint checks that the value of a structural feature is equal to a primitive value (EAttributeConstraint) or a variable (EReferenceConstraint). If the variable has not been already declared, the EReferenceConstraint navigates the reference and stores its value on that variable. For more complex attribute conditions, CheckConstraints allow defining imperative blocks with expressions.
- A PatternConstraint invokes a pattern and stores its results in a local variable.

Listing 7.5 shows the implementation of the running example query using IncQuery. The meaning of each statement on the test pattern is:

1. Line 7: get all the outgoing Transitions from S and store them in T. Note that the type of variable T is dynamically inferred at runtime.
2. Line 8: get the target from each Transition T and store it in FS. Note that even the multiplicity of a variable changes dynamically.
3. Line 9: check that FS is an instance of FinalState.

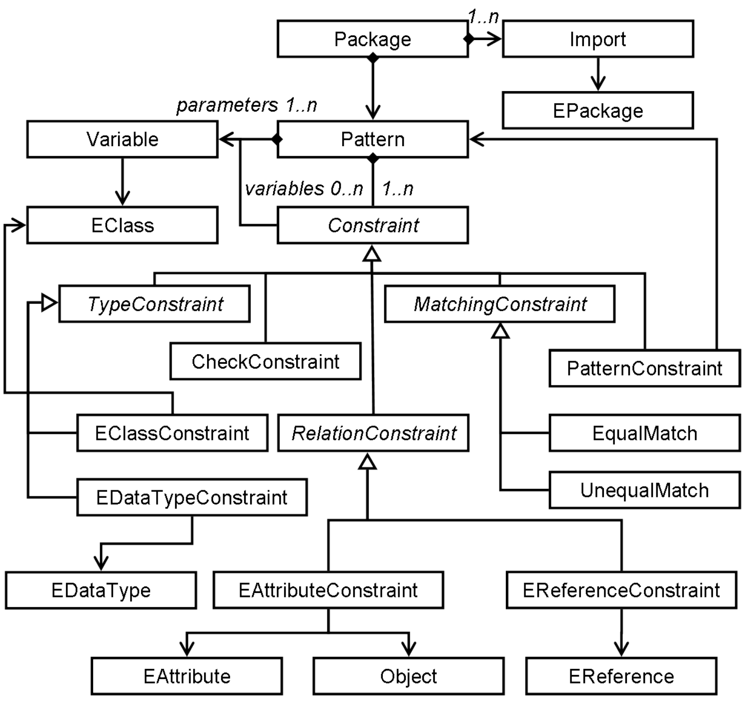


Figure 7.5 Abstract syntax of IncQuery

4. Line 10: check that the name of T is *erro*r.

After writing the pattern file, it is automatically compiled by XText, producing a Matcher and a MatcherFactory classes that need to be registered as Eclipse plugins. Then, a client application may request them to a registry in order to execute the query against an entire Resource. The results of the query are returned as IPattern-Match objects, which are maps of objects indexed by the names of the parameters defined in the pattern (in this case, the only parameter is S). Note that only the source code for generated EMF is shown, since IncQuery cannot work with dynamic EMF, as it requires the generation of the EMF code for the metamodels and their registration as Eclipse plugins.

```
 1  //IncQuery file
 2  package StateMachine
 3
 4  import "es.modelum.statemachine"
 5
 6  pattern test(S : State) = {
 7    State.outgoing(S, T);
 8    Transition.target(T, FS);
 9    FinalState(FS);
10    Transition.name(T, "error");
11  }
12
13  //Invocation from client application
```

```
List <State> result = new LinkedList<State>();
IncQueryMatcher matcher = MatcherFactoryRegistry
  .getMatcherFactory("StateMachine.test")
    .getMatcher(resource);

Collection <IPatternMatch> matches = matcher.getAllMatches();

for (IPatternMatch match : matches) {
  result .add(match.get("S"));
}
```

Listing 7.5 Example query with IncQuery

IncQuery evaluates patterns by representing the source model as a VPM [VP03] graph and then executing the Rete algorithm against that graph. Once a pattern is calculated, its results are cached and if some model element is modified, every evaluated pattern is evaluated again for that model element, saving execution time.

7.5.5 CDO OCL

The recommended query language for CDO (see Section 3.2.4) is OCL. An MDT OCL engine runs on the server side of CDO and evaluates queries against the distributed objects. The main difference between CDO OCL and MDT OCL is the interface, which in the former is much simpler. The only classes involved in a CDO OCL query are:

- CDOView or CDOTransaction for read-only access. These classes represent transactions, as explained in Section 3.2.4.
- CDOQuery for defining a query as plain text. This class is intended to generate queries for different languages, so the name of the language must be passed as a parameter along with the search scope and the context. The results of the query are returned as a generic type or a generic list, so there is no type casting required.

Listing 7.6 shows the implementation of the example query using CDO OCL. Note that only generated EMF has been used since CDO requires it. Also, note that there is no direct relationship between the Resource from where the StateMachine is loaded and the CDOView; this is due to the fact that the scope of a transaction in CDO is the whole repository, not just a model.

```
CDOSession session = getSession();
CDOView view = session.openView();

StateMachine stateMachine = (StateMachine)resource.getContents().get(0);
CDOQuery query = view.createQuery( "ocl", " self . states ->
    + "select(st : State | st.outgoing -> "
      + "exists(t : Transition | t.target.oclIsKindOf(FinalState) "
        + "and t.name = 'error'))", stateMachine);

```

```
List<State> result = query.getResult();
```

Listing 7.6 Example query with CDO OCL

The server side of CDO uses MDT OCL to parse and evaluate queries against the distributed objects held by the server. It is worth noting that a query is not evaluated directly against the database; instead, all the model elements that may be involved in that query have to be loaded from the database and held in the server which, as will be seen in Chapter 9 delivers poor performance.

8 MorsaQL

*If the Sun don't come,
you get a tan from
standing in the
English rain*

In this chapter we present our approach to model querying using Morsa, our own model repository, which was introduced in Chapter 5. As done with the other approaches in Section 7.5, the query language of Morsa will be described in terms of its abstract syntax, its concrete syntax and its semantics.

8.1 Design goals

As explained in Section 7.1, a dedicated querying solution is a must when having to perform complex queries against large models. After studying the approaches described in Section 7.5 and testing some of them with Morsa, we found their performance too low, mainly because of the way they traverse models to query them. For this reason, we decided to design and implement our own query language for Morsa, called *Morsa Query Language* (MorsaQL).

The main *design goals* of MorsaQL are its efficiency, usability and safeness, as defined in Section 7.3. **Efficiency** on MorsaQL is achieved through its semantics, which optimize the access to the database backend by minimizing the amount of data that is transferred between the client and the server; in this way, a client application can query the Morsa repository using very little memory.

The **usability** of MorsaQL is achieved through its implementation as an internal DSL, which provides a natural syntax with declarative statements and no type checking, resulting in a *readable* and *writable* language that is *consistent* with common notations as it mimics the syntax of SQL and EMF Query.

Finally, the **safeness** of MorsaQL also relies on its implementation as an internal DSL, making it *portable* as it is easy to integrate queries on source code without any external parser and *reliable*, thanks to its syntactic checking capabilities. We have chosen to leave **effectiveness** as a secondary goal, since we are focused on integrating our query language in client applications in a usable and efficient way, rather than to provide a highly expressive language; however, *preciseness* has been tackled, supporting polymorphism.

The design of MorsaQL is inspired on SQL as a query is built using a SELECT - FROM - WHERE schema, although it uses object navigation instead of relational algebra to define the constraints of a query. This choice has been made to make the language *consistent* with both SQL and EMF Query, which is the recommended query language for EMF; we have not chosen to be consistent with OCL since it is too complex to be implemented as an internal DSL. The SELECT operator declares the type of the resulting element; the FROM operator declares the *context* (a.k.a. search scope) of the query, that is, the ancestor to all the possible results; finally, the WHERE operator specifies the constraints that the results must satisfy using navigational statements that narrow the search scope by traversing the model through its relationships and conditions that check the actual contraints.

8.2 Abstract syntax

The abstract syntax of MorsaQL is shown in Figure 8.1. Although it is an internal DSL, the classes are not named after the keyword of the operation they represent as it is done in EMF Query (see Section 7.5.2): the QueryBuilder provides methods to build queries in a semantic-noise free way without imposing such constraints on the rest of the classes. The main classes provided by MorsaQL for building model queries are:

- A MorsaQuery is a query that can be passed to a MorsaResource for its parsing and execution. Such a query cannot be modified.
- A ParseableQuery is a query that can be parsed by QueryParser, normally under request of a MorsaResource or a MorsaDriver internally. The result of this parsing is a set of MorsaObjects as explained in Section 8.4. The difference between a ParseableQuery and a MorsaQuery is that the former can be modified to parameterize the following attributes of the query: the maximum depth (see Section 5.1.2) of the results (depth), the maximum number of them (limit), whether they are returned as proxies or internally resolved by the MorsaDriver (asProxies), whether to consider or not the subtypes of the involved metatypes (includeSubtypes) and to only return the number of results found (count).

- The QueryBuilder is the *expression builder* (see Section 2.5) that manages the building of queries so that the process may be performed as the instantiation of an internal DSL. It provides methods for instantiating the different classes in a natural way that does not involve the use of Java constructors, reducing the syntactic noise.
- A SelectStatement is the root of a MorsaQL query. It declares the expected metatype of the results and gives access to the rest of the elements in a query.
- A FromStatement specifies the context of a query, i.e. its search scope, in the form of an EObject or a collection of them. If no context is given, the QueryParser assumes that the query is performed against the roots of a MorsaResource. The initial context for the first condition is, by default, conformed by all the objects of the metatype declared in the SelectQuery that are descendants of the context; this saves navigation from the root of the model to the elements of interest.
- A WhereStatement is, like the WHERE class of EMF Query, the start point from where the constraints that must be satisfied by a query result are specified.
- An AbstractCondition is a *condition* (i.e. constraint) that must be satisfied by the results of the query. It checks a given feature of a given metatype and it is the root of the rest of the condition classes.
- A NavigateStatement represents a navigation over a reference of a partial result (i.e. a model element that satisfies the previous conditions). This is similar to the EObjectReferenceValueCondition of EMF Query. The elements on the other side of the navigated reference must satisfy the nested condition referenced by nextCondition, which uses the result of the navigation as its context.
- A TerminalCondition represents a condition whose results cannot be navigated because they are primitive values. As its name suggests, every nesting of conditions must end with a TerminalCondition. This class is the root class for BooleanCondition conditions such as *equals* or *not equals*, StringCondition conditions such as *equals* or *substring*, NumericalCondition conditions such as *greater* or *less than* and CollectionCondition conditions such as *includes* or *empty*; there is a class for representing each condition, although they have been ommitted from Figure 8.1 for the sake of readability.
- A TypeCondition checks the type of a feature value. The difference between a TypeCondition and a NavigationCondition is that the former is a terminal condition and it implies no navigation, so the context of the condition remains the same.
- An Operator is a boolean operator (i.e. *and* and *or*, not shown) that can relate the results of various conditions. The condition that owns the operator is its left side, while its rightSide reference points the condition that is the right side of the operator. The condition on the right side inherits the context from the

one on the left side, that is, the one declared on the FromStatement object or the result of the last NavigateCondition.

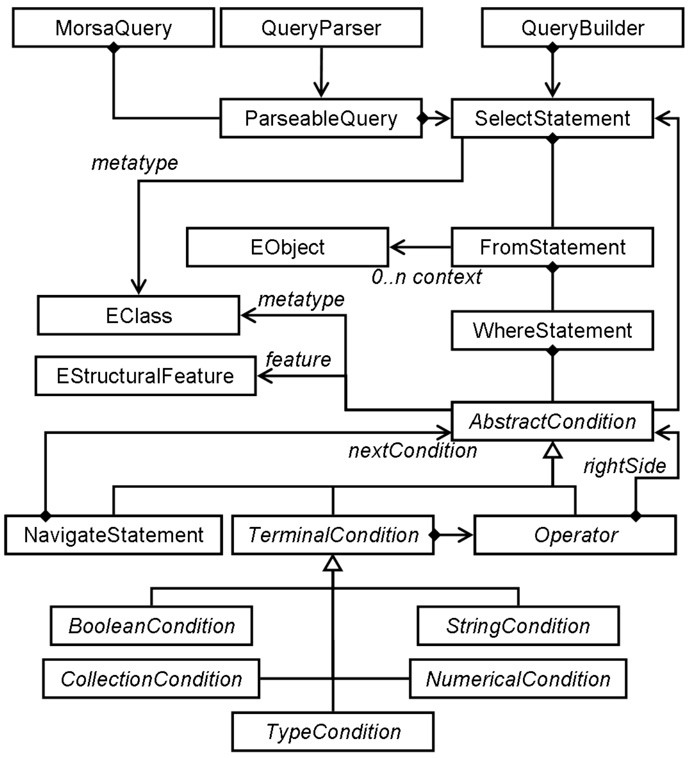


Figure 8.1 Abstract syntax of MorsaQL

8.3 Concrete syntax

As explained in Section 8.1, the concrete syntax of MorsaQL has been designed to make it *consistent* with the most used query language (SQL) and the current EMF querying approach (EMF Query); this means that a query is built using SELECT - FROM - WHERE block. The constraints (a.k.a. conditions) of the query are specified against the current context, which is narrowed using the navigate method that navigates the relationships of the model elements in the search scope; each constraint is specified using a keyword (i.e. a method), a structural feature (or its name) and a value. Once a query has been built, the done method makes it available as a MorsaQuery object that can be queried using a MorsaResource instance.

The concrete syntax of MorsaQL has been implemented as an internal DSL with the *usability* and *safeness* in mind. The QueryBuilder class is statically imported to allow a natural-looking use of the *method chaining* and *nested function* patterns (see Section 2.5) where the names of the methods are the keywords of the language and no Java constructors are used. This can be seen in Listing 8.1, which shows the code that must be written to perform the example query defined in Section 7.4.

```
 1 //With dynamic EMF
 2 EObject stateMachine = (EObject)resource.getContents().get(0);
 3
 4 MorsaQuery query = SELECT("es.modelum.statemachine/State")
 5    .FROM(stateMachine)
 6    .WHERE(navigate("outgoing")
 7       .asType("es.modelum.statemachine/Transition")
 8       .INSTANCEOF("target", "es.modelum.statemachine/FinalState")
 9       .AND(STREQ("name", "error"))).asProxies().done();
10
11  Collection <EObject> result = morsaResource.query(query);
12
13 //With generated EMF
14 StateMachine stateMachine = (StateMachine)resource.getContents().get(0);
15
16 MorsaQuery query = SELECT(StateMachinePackage.getState()
17    .FROM(stateMachine).
18    .WHERE(navigate(StateMachinePackage.getState_Outgoing())
19       .INSTANCEOF(StateMachinePackage.getTransition_Target(), StateMachinePackage.getFinalState())
20       .AND(STREQ(StateMachinePackage.getTransition_Name, "error"))).asProxies().done();
21
22  Collection <EObject> result = morsaResource.query(query);
23
24 //Using find
25 StateMachine stateMachine = (StateMachine)resource.getContents().get(0);
26
27  Collection <EObject> result = SELECT(StateMachinePackage.getState()
28    .FROM(stateMachine)
29    .WHERE(navigate("outgoing")
30       .INSTANCEOF("target", es.modelum.statemachine/FinalState")
31       .AND(STREQ("name", "error"))).asProxies().find();
```

Listing 8.1 Example query with MorsaQL

The above listed code does the following:

1. Line 4 (line 16 for generated EMF) creates a SelectStatement that specifies that the type of the result must be State.
2. Line 5 (line 17 for generated EMF) creates a FromStatement that specifies the root context of the query. The actual context for the first condition will be the set of States that are descendants of stateMachine.
3. Line 6 (line 18 for generated EMF) creates a WhereStatement that serves as the container for all the conditions. The navigate method instances a NavigationStatement that navigates the outgoing reference of each State, retrieving every Transition element. The Transition type of the navigated elements is ensured by the asType method (line 7), which is optional.
4. Line 8 (line 19 for generated EMF) creates an INSTANCEOFCondition condition (subclass of TypeCondition) that checks if the value pointed by the target

reference of each Transition obtained in Line 6 (line 18 for generated EMF) is an instance of the FinalState class.

5. Line 9 (line 20 for generated EMF) creates an ANDOperator (subclass of Operator) that joins the condition built in line 8 (line 19 for generated EMF) with a STREQCondition (subclass of StringCondition) that checks if the value of the name attribute of each Transition obtained in line 8 (line 19 for generated EMF) is *error*. The AND and OR operators are short-circuited in MorsaQL. The done method tells the QueryBuilder class to stop building the query, returning a MorsaQuery object that contains the query that must be parsed as a ParseableQuery object; its execution is then parameterized using the asProxies method, which tells the driver to not resolve the results and return them as proxies.

As can be seen, the concrete syntax of MorsaQL features high **usability** thanks to the extensive use of the QueryBuilder methods (the ones that do not define an explicit caller object, such as navigate), following a SQL-like syntax where no Java keywords appear and the parameters of the methods have been reduced to the ones that are strictly necessary for querying purposes.

For even less code usage, ParseableQuery provides a find method that automatically requests the execution of the query to the MorsaResource, as shown in lines 25-31, without having to explicitly use a MorsaQuery object. The **safeness** goal is achieved thanks to its implementation as an internal DSL, since the syntactic correctness of a query is checked by the Java compiler and the integration between a client application and a query is clean.

Note that in Listing 8.1, the main difference between using dynamic EMF and generated EMF is that metatypes and feature declarations may be done using Java objects in the latter, as it is done in EMF Query (see Section 7.5.2); moreover, they can be also declared as strings when using generated EMF. It is also worth noting that the type check done by the asType (line 7) method is not compulsory: if ommitted (line 18), the metatype of the navigated element is the one specified by the reference.

8.4 Semantics

We will now describe how MorsaQL performs a query. Its semantics has been devised to achieve good *efficiency* through low execution time and client memory usage. The process of performing a query consists in *generating* a different MorsaObject for each one of its conditions and then use them to *execute* database queries incrementally, fetching partial results that conform the search scope of the next conditions until the actual results are obtained; several MorsaObjects can be *packed* together so that less queries are effectively performed against the database.

A MorsaObject that represents a condition is built as a *prototype* of the results to be fetched, following a *query by example* approach. As explained in Section 5.1.2, a MorsaObject is composed of a descriptor and a content. In the case of such a prototype, the descriptor declares the expected type of the results (*MetaType* key), the model elements that conform the root context (*Ancestors* key) and the identifiers (*MorsaID* key) of the model elements that conform the current context (i.e. the already obtained partial results). The content of the MorsaObject declares the features that must be checked by the query and their expected values, which can be regular expressions, depending on their type.

The linking between conditions made by NavigateStatement and Operator objects is parsed as a tree of nested MorsaObjects. A MorsaObject parsed from a NavigateStatement contains a special key for the nextCondition reference in its descriptor, being its value a nested MorsaObject representing the next condition, and a special key called *Ref* in its content for the name of the reference that is navigated. A condition that is linked to another one by an Operator object declares a special key in its descriptor for its rightSide reference, being its value a nested MorsaObject representing the linked condition. Figure 8.2 shows how the example query is performed through its different stages, which are the following:

1. A generation stage parses each AbstractCondition into a MorsaObject that represents the condition as a query to the database; the MorsaEncoder encodes these objects into the database backend's native representation. For our MongoDB prototype, the MorsaObjects are transformed into DBObjects. For the example query, the following actions are performed:

   a) The NavigateStatement is parsed into a MorsaObject with the following decriptor keys:

      - *MetaType*: the type of the expected results is State, as it is the expected type of the results of the query.
      - *Ancestors*: the root context of the query is the stateMachine model element.
      - *nextCondition*: the nested MorsaObject representing the next condition (shown in the figure as an aggregation).

      Its content declares the *Ref* key with the name of the outgoing reference, which is the one that is navigated.

   b) The INSTANCEOFCondition is parsed into a MorsaObject with the following descriptor keys:

      - *MetaType*: the type of the expected result is Transition, since it is the metaclass that declares the target reference.

   - *Ancestors*: the root context of the query is the stateMachine model element.
   - *rightSide*: the nested MorsaObject representing the condition linked by the ANDOperator operator (shown in the figure as an aggregation).

   Its content declares the *target* key with a regular expression that checks that the type of the target reference value is an instance of FinalState.

   c) The STREQCondition is parsed into a MorsaObject with the following descriptor keys:

   - *MetaType*: the type of the expected result is Transition, since it is the metaclass that declares the error attribute.
   - *Ancestors*: the root context of the query is the stateMachine model element.

   Its content declares the *name* key that checks that the name attribute value is *error*.

2. A packing stage packs all the conditions that have been joined using Operators into a mininal set of MorsaObjects, since they all have the same context as they are TerminalConditions. This reduces the amount of queries that are sent to the database, increasing the *efficiency* of the whole querying process. In the example query, this packing joins the MorsaObjects parsed from the INSTANCEOFCondition and the STREQCondition into a single one with the following descriptor keys:

   - *MetaType*: the type of the expected result is Transition, which is the common type of the packed conditions.
   - *Ancestors*: the root context of the query is the stateMachine model element.

   Its content declares the *target* and *name* keys that check the values of the target and name structural features, respectively. The MorsaObject parsed from the NavigateStatement now contains the new packed MorsaObject in its *nextCondition* key.

3. Finally, an execution stage transforms the MorsaObjects into database queries and sends them to the database. The conditions that are nested in NavigateStatements are evaluated in first place to reduce the context of the latter. For the example query, this would be the following:

   a) The first query to be sent is the one contained by the NavigateStatement, which returns the *MorsaIDs* of its results: CheckSyntaxToEnd, which is the transition between the Check syntax and End states and CheckSemanticsToEnd, which is the transition between the Check semantics and End states (see Figure 7.2).

b) The query for the NavigateStatement is then executed using the *MorsaIDs* of the results of its referenced query (CheckSyntaxToEnd and CheckSemanticsToEnd, their actual *MorsaIDs* are not shown for the sake of readability) as its context, declaring them in its *MorsaID* descriptor key. This query returns the information needed to build the proxies that represent the results: the Check syntax and Check semantics states, shown in the figure using the notation from Figure 7.2.

c) The results are then returned to the client application as proxies, which would be resolved automatically by the MorsaDriver when they are accessed.

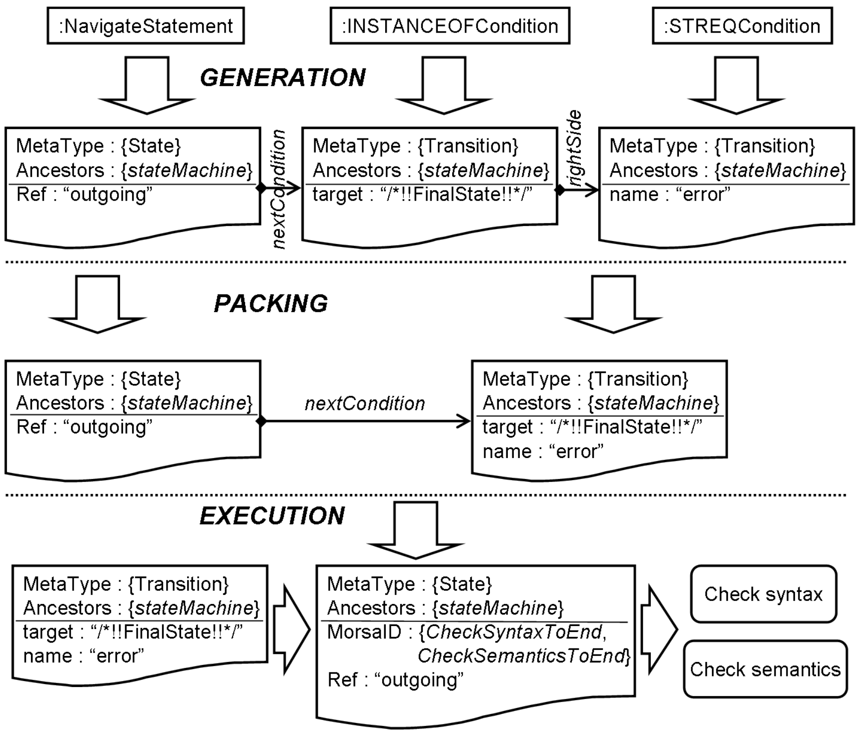


Figure 8.2 Semantics of the Morsa Query Language for the example query

For even better *efficiency*, Morsa uses a cache that stores the results of every query as MorsaReferences. If a cached query is executed again, its results are resolved from the ObjectCache if they are still in memory or otherwise fetched from the database without having to recalculate the query.

9
Evaluation of MorsaQL

See how they fly
like Lucy in the sky,
see how they run

In this chapter, the evaluation of the different querying approaches presented in Section 7.5 and the Morsa Query Language will be shown. The test case used for the evaluation is the query proposed in the running example introduced in Chapter 4, whose implementation in OCL is shown in Listing 4.1. The test models are the ones shown in Table 4.1.

This chapter is organized as follows: first, the **effectiveness**, **usability** and **safeness** dimensions of each approach (as defined in Section 7.3) will be tested by analyzing the code used to build the test query; the **efficiency** dimension will be evaluated by executing the different querying approaches over three persistence solutions: XMI (see Section 3.2.1), CDO (see Section 3.2.4) and Morsa (see Chapter 5); finally, the results of the evaluation will be commented.

9.1 Evaluated approaches

Not every combination of querying approaches and persistence solutions can be done: some of the former are only available for the persistence solutions they are designed to interact with. Table 9.1 explains the possible combinations between querying approaches and persistence solutions, showing also which are the default and recommended ones for each persistence solution. In the following, each combination will be named after its persistence solution and its querying approach (e.g. XMI+EMF stands for plain EMF over XMI).

We have not considered IncQuery over Morsa or CDO because it basically indexes and transforms a whole model before querying it, which is highly inefficient for a model repository as it is a case of full load (see Section 6.2).

	XMI	MORSA	CDO
EMF	Default	Default	Default
EMFQ	Recommended	Tested	Tested
OCL	Tested	Tested	Tested
INCQ	Tested	Not considered	Not considered
COCL	Not available	Not available	Recommended
MQL	Not available	Recommended	Not available

Table 9.1 Combinations between persistence solutions and querying approaches. EMF stands for plain EMF, EMFQ stands for EMF Query, OCL stands for MDT OCL, INCQ stands for Inc Query, COCL stands for CDO OCL and MQL stands for MorsaQL

9.2 Effectiveness, usability and safeness evaluation

The first three dimensions identified on Section 7.3 will be evaluated together in this section. For each querying approach, the source code of the test query will be shown and used to evaluate its **effectiveness**, **usability** and **safeness** using the metrics shown in Tables 9.2, 9.3 and 9.4, respectively. These metrics are aligned with the ones commented in Section 7.3.

Metric	Meaning	Unit
NQ	Number of queries against the persistent storage	Integer

Table 9.2 Evaluation metrics for the **effectiveness** dimension

Metric	Meaning	Unit
DC	Majority of code is declarative	Boolean
TC	Absence of type castings and checks	Boolean
RC	Absence of mixed non-query code	Boolean
CS	Consistency with established query languages	Boolean

Table 9.3 Evaluation metrics for the **usability** dimension

Metric	Meaning	Unit
SY	Syntactic checking prior to execution	Boolean
SE	Semantic checking prior to execution	Boolean
IN	Independence from external tools	Boolean
FB	Existence of a formal basis	Boolean

Table 9.4 Evaluation metrics for the **safeness** dimension

To make the reading of the evaluation of each dimension easier, the terms *low*, *medium* and *high* will be used. For the **efectiveness** dimension, we have estimated that a *high* value is achieved when an approach uses the same or less queries than the OCL standard, which uses 1 query (*NQ* metric), as shown in the Listing 4.1.

Given that, a *medium* value is achieved when 2 queries are used and a *low* value is achieved when 3 or more queries are needed.

For the **usability** and **safeness** dimensions, each metric with a value of True adds 1 point to the total score of the approach for that dimension. A *low* value is achieved with a score of 1 or less, a *medium* value is achieved with a score of 2 and a a *high* value is achieved with a score of 3 or above.

9.2.1 Plain EMF

The test query has been implemented in plain EMF in two steps: one that obtains every TypeDeclaration and another one that checks if a TypeDeclaration satisfies the constraints of the query. This separation has been done to refactor the second step into a single checkTypeDeclaration method, as it is common to all the studied persistence solutions, which have a different method for obtaining all the TypeDeclarations, optimized using their own features.

Listings 9.1 and 9.2 show the implementations of the checkTypeDeclaration method with dynamic and generated EMF, respectively; the main differences between both of them are the ones pointed in Section 7.5.1. Listings 9.3, 9.4 and 9.5 show the code to get all the TypeDeclarations from XMI, Morsa and CDO, respectively (Listings 9.4 and 9.5 will be used in the rest of the test cases for Morsa and CDO, respectively). Morsa and CDO use their advanced capabilities to access directly to all the TypeDeclarations of a model, which is impossible with plain EMF.

```
//With dynamic EMF
boolean checkTypeDeclaration(EObject typeDeclaration) {
  EPackage domPackage = typeDeclaration.eClass().getEPackage();
  EClass typeDeclarationClass = typeDeclaration.eClass();
  EClass methodDeclarationClass = (EClass)domPackage.getEClassifier("MethodDeclaration");
  EClass simpleTypeClass = (EClass)domPackage.getEClassifier("SimpleType");
  EClass nameClass = (EClass)domPackage.getEClassifier("Name");
  EClass mdifierClass = (EClass)domPackage.getEClassifier("Modifier");

  List<EObject> bodyDeclarations =
        (List<EObject>)typeDeclaration.eGet(typeDeclarationClass.getEStructuralFeature("bodyDeclarations"));
  for (EObject bodyDeclaration : bodyDeclarations) {
    if (found) break;
    if (bodyDeclaration.eClass().equals(methodDeclarationClass)) {
      EObject returnType = (EObject)bodyDeclaration.eGet(
            methodDeclarationClass.getEStructuralFeature("returnType"));
      if (returnType.eClass().equals(simpleTypeClass)) {
        EObject returnTypeName = (EObject)returnType.eGet( simpleTypeClass.getEstructuralFeature("name"));
        String returnTypeFQName = (String)returnTypeName.eGet(
              nameClass.getEStructuralFeature("fullyQualifiedName"));
        EObject typeDeclarationName = (EObject)typeDeclaration.eGet(
              typeDeclarationClass. getEstructuralFeature ("name"));
        String typeDeclarationFQName = (String)typeDeclarationName.eGet(
              nameClass.getEStructuralFeature("fullyQualifiedName"));
        if (returnTypeFQName.equals(typeDeclarationFQName)) {
          boolean foundStatic = false;
          boolean foundPublic = false;
```

```
            List <EObject> modifierList = (List<EObject>bodyDeclaration.eGet(
                MethodDeclarationClass.getEStructuralFeature("modifiers"));
            for (EObject modifier : modifierList ) {
              if ( modifier . eClass () . equals( modifierClass )) {
                if (foundStatic && foundPublic) break;
                if ((Boolean)modifier.eGet( modifierClass . getEStructuralFeature (" static "))) foundStatic = true;
                if ((Boolean)modifier.eGet( modifierClass . getEStructuralFeature (" public"))) foundPublic = true;
              }
            }
            if (foundStatic && foundPublic) result.add(typeDeclaration);
            return true;
          }
        }
      }
    }
    return false ;
}
```

Listing 9.1 TypeDeclaration check with dynamic plain EMF

```
//With generated EMF
boolean checkTypeDeclaration(TypeDeclaration typeDeclaration ) {
  for (BodyDeclaration bodyDeclaration : typeDeclaration .getBodyDeclarations()) {
    if (found) break;
    if (bodyDeclaration instanceof MethodDeclaration) {
      MethodDeclaration methodDeclaration = (MethodDeclaration)bodyDeclaration;
      if (methodDeclaration.getReturnType() instanceof SimpleType) {
        SimpleType returnType = methodDeclaration.getReturnType();
        String returnTypeFQName = returnType.getName().getFullyQualifiedName();
        String typeDeclarationFQName = typeDeclaration.getFullyQualifiedName();
        if (returnTypeFQName.equals(typeDeclarationFQName)) {
          boolean foundStatic = false;
          boolean foundPublic = false;
          for (ExtendedModifier modifier : methodDeclaration.getModifiers()) {
            if ( modifier instanceof Modifier) {
              if (foundStatic && foundPublic) break;
              if ( modifier . getStatic ) foundStatic = true;
              if ( modifier . getPublic) foundPublic = true;
            }
          }
          if (foundStatic && foundPublic) result.add(typeDeclaration);
          return true;
        }
      }
    }
  }
  return false ;
}
```

Listing 9.2 TypeDeclaration check with generated plain EMF

```
//With dynamic EMF
List <EObject> result = new LinkedList<EObject>();
Iterator <EObject> iterator = resource.getAllContents();

while ( iterator .hasNext()) {
  EObject eObject = iterator .next();
  if (eObject.eClass().getName().equals("TypeDeclaration") && eObject.eClass().getEPackage.getNsURI()
    .equals("org.amma.dsl.jdt.dom")) {
    if (checkTypeDeclaration(eObject))
```

```
        result .add(eObject);
  }
}

//With generated EMF
List<TypeDeclaration> result = new LinkedList<TypeDeclaration>();
Iterator <EObject> iterator = resource.getAllContents();

while ( iterator .hasNext()) {
  EObject eObject = iterator .next();
  if (eObject instanceof TypeDeclaration) {
     if (checkTypeDeclaration((TypeDeclaration)eObject))
        result .add((TypeDeclaration)eObject);
  }
}
```

Listing 9.3 Code for obtaining the TypeDeclarations from XMI with plain EMF

Table 9.5 summarizes the evaluation of the metrics for the plain EMF approach:

Metric	Meaning	Value
NQ	Number of queries against the persistent storage	Infinite
DC	Majority of code is declarative	False
TC	Absence of type castings and checks	False
RC	Absence of mixed non-query code	False
CS	Consistency with established query languages	False
SY	Syntactic checking prior to execution	True
SE	Semantic checking prior to execution	True
IN	Independence from external tools	True
FB	Existence of a formal basis	False

Table 9.5 Evaluation of the **effectiveness**, **usability** and **safeness** metrics for the plain EMF approach

The **effectiveness** of this approach is not easy to evaluate, since there is no explicit querying or separated access to the persistence solutions. We will consider it *low* because we gave the *NQ* metric a value of infinite, since depending on the persistence solution, this approach may issue a query every time a relationship is navigated.

The **usability** of this approach is obviously *low*: the code is almost completely imperative and, in the case of dynamic EMF, there are so many type castings and checks that it is hardly readable or writable at all. Moreover, there is no difference between the regular code and the one dedicated to querying. The code does not resemble any well-established query language either. On the other hand, no external tools are needed to execute the code, so it is portable.

The **safeness** of this approach is *high*: since it is basically plain EMF code, there is no need of an integration effort into client applications and the syntactic and semantic checking is done by the Java compiler; on the other hand, semantic checking is only possible when generated EMF is used. No formal basis relies underneath EMF.

```
//With dynamic EMF
List<EObject> result = new LinkedList<EObject>();
List<EObject> typeDeclarationList =
  SELECT("org.amma.dsl.jdt.dom/TypeDeclaration").FROM(resource);

for (EObject typeDeclaration : typeDeclarationList ) {
  if (checkTypeDeclaration(typeDeclaration))
    result .add(typeDeclaration);
}

//With generated EMF
List<TypeDeclaration> result = new LinkedList<TypeDeclaration>();
List<EObject> typeDeclarationList =
  SELECT("org.amma.dsl.jdt.dom/TypeDeclaration").FROM(resource);

for (EObject typeDeclaration : typeDeclarationList ) {
  if (checkTypeDeclaration((TypeDeclaration)typeDeclaration))
    result .add((TypeDeclaration)typeDeclaration);
}
```

Listing 9.4 Code for obtaining the TypeDeclarations from Morsa

```
//With generated EMF
CDOSession session = getSession();
CDOView view = session.openView();
List<TypeDeclaration> result = new LinkedList<TypeDeclaration>();

List<TypeDeclaration> typeDeclarationList =
  view.createQuery("ocl", "TypeDeclaration. allInstances ()",
    DOMPackage.getTypeDeclaration());

for (TypeDeclaration typeDeclaration : typeDeclarationList ) {
  if (checkTypeDeclaration(typeDeclaration))
    result .add(typeDeclaration);
}
```

Listing 9.5 Code for obtaining the TypeDeclarations from CDO

9.2.2 EMF Query

Listings 9.6 and 9.7 show the implementations of the checkTypeDeclaration method with EMF Query for dynamic and generated EMF, respectively, that are executed over every TypeDeclaration, which may have been obtained in different ways, depending on the persistence solution that is being used; Listings 9.4, 9.5 and 9.8 show the code to get all the TypeDeclarations from Morsa, CDO and XMI, respectively. The main differences between the dynamic and the generated EMF versions are the same as the ones pointed in Section 7.5.2. Table 9.6 shows the evaluation of the metrics for this approach.

```
//With dynamic EMF
boolean checkTypeDeclaration(EObject typeDeclaration) {
  EPackage domPackage = typeDeclaration.eClass().getEPackage();
  EClass typeDeclarationClass = typeDeclaration.eClass();
  EClass methodDeclarationClass = (EClass)domPackage.getEClassifier("MethodDeclaration");
  EClass simpleTypeClass = (EClass)domPackage.getEClassifier("SimpleType");
  EClass nameClass = (EClass)domPackage.getEClassifier("Name");
  EClass modifierClass = (EClass)domPackage.getEClassifier("Modifier");
  EObject typeDeclarationName = (EObject)typeDeclaration.eGet( simpleTypeClass. getEstructuralFeature ("name"));
  String typeDeclarationFQName = (String)typeDeclarationName.eGet(
       nameClass.getEStructuralFeature("fullyQualifiedName"));

  SELECT select = new SELECT(1,
    new FROM(typeDeclaration),
    new WHERE(new EObjectReferenceValueCondition(
      (EReference) typeDeclarationClass . getEStructuralFeature ("bodyDeclarations"),
      new ObjectTypeRelationCondition(methodDeclarationClass).AND(
        new EObjectReferenceValueCondition(
             (EReference)methodDeclarationClass.getEStructuralFeature("returnType"),
          new EObjectReferenceValueCondition( (EReference) simpleTypeClass. getEStructuralFeature ("name"),
            new EObjectAttributeValueCondition( (EAttribute)
                 nameClass.getEStructuralFeature(" fullyQualifiedName"),
            new StringValue(typeDeclarationFQName))))
        .AND(new EObjectReferenceValueCondition(
             (EReference)methodDeclarationClass.getEStructuralFeature("modifiers"),
          new EObjectAttributeValueCondition( (EAttribute)  modifierClass . getEStructuralFeature (" static "),
            new BooleanCondition(true)))
        .AND(new EObjectReferenceValueCondition(
             (EReference)methodDeclarationClass.getEStructuralFeature("modifiers"),
          new EObjectAttributeValueCondition( (EAttribute)  modifierClass . getEStructuralFeature (" public"),
            new BooleanCondition(true))))
        )))));
  IQueryResult  result  = select . execute();
  return  ! result . isEmpty();
}
```

Listing 9.6 checkTypeDeclaration check with dynamic EMF Query

```
//With generated EMF
boolean checkTypeDeclaration(EObject typeDeclaration) {
  SELECT select = new SELECT(1,
    new FROM(typeDeclaration),
    new WHERE(new EObjectReferenceValueCondition(
        DOMPackage.getTypeDeclaration_BodyDeclarations(),
      new ObjectTypeRelationCondition(
          DOMPackage.getMethodDeclaration()).AND(
        new EObjectReferenceValueCondition(
            DOMPackage.getMethodDeclaration_ReturnType(),
          new EObjectReferenceValueCondition(
              DOMPackage.getSimpleType_Name(),
            new EObjectAttributeValueCondition(
                DOMPackage.getName_FullyQualifiedName(),
            new StringValue(name))))
        .AND(new EObjectReferenceValueCondition(
            DOMPackage.getMethodDeclaration_ReturnType(),
          new EObjectAttributeValueCondition(
                DOMPackage.getModifier_Static,
            new BooleanCondition(true)))
        .AND(new EObjectReferenceValueCondition(
            DOMPackage.getBodyDeclaration_Modifiers(),
          new EObjectAttributeValueCondition(
              DOMPackage.getModifier_Public,
            new BooleanCondition(true))))
        )))));
  IQueryResult  result  = select . execute();
  return  ! result . isEmpty();
}
```

Listing 9.7 TypeDeclaration check with generated EMF Query

Metric	Meaning	Value
NQ	Number of queries against the persistent storage	2
DC	Majority of code is declarative	False
TC	Absence of type castings and checks	False
RC	Absence of mixed non-query code	False
CS	Consistency with established query languages	True
SY	Syntactic checking prior to execution	True
SE	Semantic checking prior to execution	False
IN	Independence from external tools	True
FB	Existence of a formal basis	False

Table 9.6 Evaluation of the **effectiveness**, **usability** and **safeness** metrics for the EMF Query approach

The **effectiveness** of this approach is *medium*: two separate queries are needed to obtain the results: one to get all the TypeDeclarations and another one to check each one of them. This is due to the fact that EMF Query does not provide named variables, which may have been used to reference the feature name of each TypeDeclaration from the rest of the query (see Listing 4.1 for an example of this).

The **usability** of this approach is *low*: although the code is somehow declarative and consistent since it mimics the syntax of SQL, there are lots of Java constructors and complicated class names all over the queries, and the fact that it requires the EStructuralFeature and EClass objects to define conditions makes it even worse, with many type castings for dynamic EMF.

The **safeness** of this approach is *medium*: the syntactic correctness of the statements is checked before execution but not the semantic one, as there is no connection between the results of one condition and the conditions that are applied to them; moreover, the approach is independent from any external tool.

```
//With dynamic EMF
List<EObject> result = new LinkedList<EObject>();
SELECT select =
  new SELECT(Integer.MAX_VALUE,
    new FROM(resource.getContents()),
      new WHERE(new EObjectTypeRelationCondition(typeDeclarationClass));

IQueryResult typeDeclarationList = select.execute();

for (EObject typeDeclaration : typeDeclarationList ) {
  if (checkTypeDeclaration(typeDeclaration))
    result.add(typeDeclaration);
}

//With generated EMF
List<TypeDeclaration> result = new LinkedList<TypeDeclaration>();
SELECT select =
  new SELECT(Integer.MAX_VALUE,
    new FROM(resource.getContents()),
      new WHERE(new EObjectTypeRelationCondition(typeDeclarationClass));

IQueryResult typeDeclarationList = select.execute();

for (EObject typeDeclaration : typeDeclarationList) {
  if (checkTypeDeclaration((TypeDeclaration)typeDeclaration))
    result.add((TypeDeclaration)typeDeclaration);
}
```

Listing 9.8 Code for obtaining the TypeDeclarations from XMI with EMF Query

9.2.3 MDT OCL

Listing 4.1 shows the OCL query for the test case as it would be for standard OCL. However, this query led to errors when executed in MDT OCL (e.g. elements that were identified as being of a certain type using oclIsKindOf failed when cast to that type using oclAsType), so after some debugging we were able to write the query with the fewest OCL queries, as shown in Listings 9.9 (for dynamic EMF) and 9.10 (for generated EMF). As can be seen, the checkTypeDeclaration method needs a TypeDeclaration argument: Listings 9.4, 9.5 and 9.11 show the code to get all the TypeDeclarations from Morsa, CDO and XMI, respectively. The main differences between the dynamic and generated EMF implementations are same as the ones pointed in Section 7.5.3. Table 9.7 shows the evaluation of the metrics for this approach.

```
//With dynamic EMF
boolean checkTypeDeclaration(EObject typeDeclaration) {
  EPackage domPackage = typeDeclaration.eClass().getEPackage();
  EClass typeDeclarationClass = typeDeclaration.eClass();
  EClass nameClass = (EClass)domPackage.getEClassifier("Name");
  EClass methodDeclarationClass = (EClass)domPackage.getEClassifier("MethodDeclaration");
  EObject typeDeclarationName =
        (EObject)typeDeclaration.eGet(typeDeclarationClass.getEstructuralFeature("name"));
  String typeDeclarationFQName = (String)typeDeclarationName.eGet(
        nameClass.getEStructuralFeature("fullyQualifiedName"));

  OCL<?, EClassifier, ?, ?, ?, ?, ?, ?, ?, Constraint, EClass, EObject> ocl =
        OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);
  OCLHelper<EClassifier, ?, ?, Constraint> helper = ocl.createOCLHelper();
  helper.setContext(typeDeclarationClass);

  OCLExpression<EClassifier> expression = helper.createQuery(
    "self.bodyDeclarations -> select(bd : BodyDeclaration bd.oclIsKindOf(MethodDeclaration))");

  Query<EClassifier, EClass, EObject> query = ocl.createQuery(expression);
  Collection <EObject> methodDeclarations = (Collection<EObject>)query.evaluate(typeDeclaration);
  for (EObject methodDeclaration : methodDeclarations) {
    OCLHelper<EClassifier, ?, ?, Constraint> helper2 = ocl  .createOCLHelper();
    helper2.setContext(methodDeclarationClass);
    OCLExpression<EClassifier> expression2 = helper2.createQuery(
      "self.returnType.oclAsType(SimpleType).name.fullyQualifiedName = '" + name +
        "' and self.modifiers -> exists(m : ExtendedModifier | m.oclAsType(Modifier)._static)" +
        " and self.modifiers -> exists(m : ExtendedModifier | m.oclAsType(Modifier).public)");
    Query<EClassifier, EClass, EObject> query2 = ocl.createQuery(expression2);
    Object oclResult2 = query2.evaluate(methodDeclaration);
    if (oclResult2 instanceof Boolean && ((Boolean) oclResult2))
      return true;
  }
  return false;
}
```

Listing 9.9 MDT OCL query for the TypeDeclaration check using dynamic EMF

```
//With generated EMF
boolean checkTypeDeclaration(TypeDeclaration typeDeclaration) {
  OCL<?, EClassifier, ?, ?, ?, ?, ?, ?, ?, Constraint, EClass, EObject> ocl =
        OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);
  OCLHelper<EClassifier, ?, ?, Constraint> helper = ocl.createOCLHelper();
  helper.setContext(DOMPackage.getTypeDeclaration());

  OCLExpression<EClassifier> expression = helper.createQuery(
    "self.bodyDeclarations -> select(bd : BodyDeclaration bd.oclIsKindOf(MethodDeclaration))");

  Query<EClassifier, EClass, EObject> query = ocl.createQuery(expression);
  Collection <MethodDeclaration> methodDeclarations =
        (Collection<MethodDeclaration>)query.evaluate(typeDeclaration);
  for (MethodDeclaration methodDeclaration : methodDeclarations) {
    OCLHelper<EClassifier, ?, ?, Constraint> helper2 = ocl   .createOCLHelper();
    helper2.setContext(methodDeclarationClass);
    OCLExpression<EClassifier> expression2 = helper2.createQuery(
      "self.returnType.oclAsType(SimpleType).name.fullyQualifiedName =
        '" + typeDeclaration.getName().getFullyQualifiedName() +
        "' and self.modifiers -> exists(m : ExtendedModifier | m.oclAsType(Modifier)._static)" +
        " and self.modifiers -> exists(m : ExtendedModifier | m.oclAsType(Modifier).public)");
    Query<EClassifier, EClass, EObject> query2 = ocl.createQuery(expression2);
    Object oclResult2 = query2.evaluate(methodDeclaration);
    if (oclResult2 instanceof Boolean && ((Boolean) oclResult2))
      return true;
  }
  return false;
}
```

Listing 9.10 MDT OCL query for the TypeDeclaration check using generated EMF

Metric	Meaning	Value
NQ	Number of queries against the persistent storage	3
DC	Majority of code is declarative	True
TC	Absence of type castings and checks	True
RC	Absence of mixed non-query code	False
CS	Consistency with established query languages	True
SY	Syntactic checking prior to execution	False
SE	Semantic checking prior to execution	False
IN	Independence from external tools	True
FB	Existence of a formal basis	False

Table 9.7 Evaluation of the **effectiveness**, **usability** and **safeness** of the metrics for the MDT OCL approach

The **effectiveness** of this approach is *low*. Three separate queries are needed to obtain the results: one to get all the TypeDeclarations, one to get all their MethodDeclaration and a last one to check each one of the latter. However, if the original OCL query (see Listing 4.1) could have been used, the effectiveness would have been *high*, since only one query would have been needed, thanks to the use of named variables.

The **usability** of this approach is *high*: the OCL code (lines 15, 23 to 25, 41 and 49 to 52 in Listing 9.10) is completely declarative and type castings are only done when necessary, although Java loop and conditional structures are needed to relate the results of the three queries. Moreover, all the code that supports the query but it is not part of it (e.g. the Helper objects) is clearly separated. The language is consistent with a well-established language since it is the proposed standard for MOF.

The **safeness** of this approach is *low*: no syntactic or semantic checking is done before the query is executed, as the queries are specified as text strings. Moreover, no formal basis is used; however, the approach is independent from external tools.

```
//With dynamic EMF
OCL<?, EClassifier, ?, ?, ?, ?, ?, ?, ?, Constraint, EClass, EObject> ocl =
  OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);

OCLHelper<EClassifier, ?, ?, Constraint> helper = ocl.createOCLHelper();
helper.setContext(typeDeclarationClass);

OCLExpression<EClassifier> expression = helper.createQuery("TypeDeclaration.allInstances()");
Query<EClassifier, EClass, EObject> query = ocl.createQuery(expression);

Collection<EObject> typeDeclarationList = (Collection<EObject>)query.evaluate(resource);
List<EObject> result = new LinkedList<EObject>();
for (EObject typeDeclaration : typeDeclarationList) {
  if (checkTypeDeclaration(typeDeclaration))
    result.add(typeDeclaration);
}

//With generated EMF
OCL<?, EClassifier, ?, ?, ?, ?, ?, ?, ?, Constraint, EClass, EObject> ocl =
  OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);

OCLHelper<EClassifier, ?, ?, Constraint> helper = ocl.createOCLHelper();
helper.setContext(DOMPackage.getTypeDeclaration());

OCLExpression<EClassifier> expression = helper.createQuery("TypeDeclaration.allInstances()");
Query<EClassifier, EClass, EObject> query = ocl.createQuery(expression);

Collection<TypeDeclaration> typeDeclarationList = (Collection<TypeDeclaration>)query.evaluate(resource);
List<TypeDeclaration> result = new LinkedList<TypeDeclaration>();
for (TypeDeclaration typeDeclaration : typeDeclarationList) {
  if (checkTypeDeclaration(typeDeclaration))
    result.add(typeDeclaration);
}
```

Listing 9.11 Code for obtaining the TypeDeclarations from XMI with MDT OCL

9.2.4 IncQuery

Listing 9.12 shows how the test query is written in IncQuery: first, an IncQuery-specific file is written with the actual pattern (i.e. query) and then that pattern is invoked from the client application. Table 9.8 shows the evaluation metrics for this approach.

```
//IncQuery file
package grabats

import "org.amma.dsl.jdt.dom"
import "org.amma.dsl.jdt.primitiveTypes"

pattern grabats(T : TypeDeclaration) = {
  TypeDeclaration.bodyDeclarations(T, M);
  MethodDeclaration.returnType(M, R);
  SimpleType(R);
  SimpleType.name.fullyQualifiedName(R, S);
  TypeDeclaration.name.fullyQualifiedName(T, S);
  find methodWithReturnType(M, S);
  MethodDeclaration.modifiers(M, MF1);
  MethodDeclaration.modifiers(M, MF2);
  MF1 != MF2;
  Modifier.public(MF1, true);
  Modifier.static(MF2, true);
}

//Invocation from client application
List<TypeDeclaration> result = new LinkedList<TypeDeclaration>();
IncQueryMatcher matcher = MatcherFactoryRegistry
  .getMatcherFactory("grabats.grabats").getMatcher(resource);

Collection<IPatternMatch> matches = matcher.getAllMatches();
for (IPatternMatch match : matches) {
  result.add(match.get("T"));
}
```

Listing 9.12 IncQuery query for the test case

Metric	Meaning	Value
NQ	Number of queries against the persistent storage	1
DC	Majority of code is declarative	True
TC	Absence of type castings and checks	True
RC	Absence of mixed non-query code	True
CS	Consistency with established query languages	False
SY	Syntactic checking prior to execution	True
SE	Semantic checking prior to execution	True
IN	Independence from external tools	False
FB	Existence of a formal basis	True

Table 9.8 Evaluation of the **effectiveness**, **usability** and **safeness** metrics for the IncQuery approach

The **effectiveness** of this approach is *high*: only one query must be performed against the persistence solution to get the expected results.

The **usability** of this approach is *high*: the code is completely declarative and since an external DSL is used, all the statements are for query definition, so there is no Java code mixed in between of it. However, the graph pattern matching is not so well-established as a common query language.

The **safeness** of this approach is *high*: although it completely depends on external tools to define queries and even to execute them, as the generated pattern matchers must be registered as plug-ins in the Eclipse platform, it checks for syntactic and semantic correctness and is based on the graph pattern formalism.

9.2.5 CDO OCL

Although it is also an implementation of OCL, CDO OCL cannot use the query defined for MDT OCL (see Listing 9.10), because its lack of type checks (i.e. oclIsKindOf invocations) makes the query return OCLInvalid objects when a type cast fails. This is not relevant for MDT OCL because it just discards the rest of query in a short-circuit fashion and continues evaluating the next partial result, but CDO OCL throws an exception whenever an OCLInvalid object is produced, so type checking is a must. Since the and operator of OCL is not short-circuited, the type checks must be done using conditional blocks;

Listing 9.13 shows how the original OCL query (see Listing 4.1) would have to be written for its execution within CDO. However, since CDO uses MDT OCL on the server side, the problems that we encountered for MDT OCL also appeared here, so we had to split the original query into several ones, as shown in Listing 9.14. Again, the code in Listing 9.5 is needed to provide the TypeDeclarations to the query; since CDO OCL can only be used with the CDO persistence solution, there are not any other options to get the TypeDeclarations as it happens with the previous approaches. Table 9.9 shows the evaluation of the metrics of this approach.

```
TypeDeclaration. allInstances () ->
  select (td : TypeDeclaration | td.bodyDeclarations ->
    exists (bd : BodyDeclaration |
      if bd.oclIsKindOf(MethodDeclaration) then
        if bd.oclAsType(MethodDeclaration).returnType
          .oclIsKindOf(SimpleType) then
          if bd.oclAsType(methodDeclaration).returnType
            .oclAsType(SimpleType).
            name.fullyQualifiedName = td.name.fullyQualifiedName then
            bd.modifiers -> exists (em : ExtendedModifier |
              if em.oclIsKindOf(Modifier) then em.oclAsType(Modifier). _static else false endif)
            and
            bd.modifiers -> exists (em : ExtendedModifier |
              if em.oclIsKindOf(Modifier) then em.oclAsType(Modifier).public else false endif)
          else false
          endif
        else false
        endif
      else false
      endif
  )
)
```

Listing 9.13 OCL query for the test case using CDO

```
//With generated EMF
boolean checkTypeDeclaration(TypeDeclaration typeDeclaration) {
  CDOQuery query = view.createQuery("ocl",
    " self . bodyDeclarations -> select(bd : BodyDeclaration |
      bd.oclIsKindOf(MethodDeclaration))", typeDeclaration);

  List<MethodDeclaration> methodList = query.getResult();
  for (MethodDeclaration methodDeclaration : methodList) {
    CDOQuery query2 = view.createQuery("ocl",
      " if self . returnType.oclIsKindOf(SimpleType) then "
      + " if self . returnType.oclAsType(SimpleType).name.fullyQualifiedName = '" +
        typeDeclaration .getName().getFullyQualifiedName() + "' then " +
        " self . modifiers -> exists(em : ExtendedModifier | "
          + " if em.oclIsKindOf(Modifier) then em.oclAsType(Modifier). _static else false endif) and "
            + " self. modifiers -> exists(em : ExtendedModifier | "
          + " if em.oclIsKindOf(Modifier) then em.oclAsType(Modifier). _public else false endif) "
      + "else false endif else false endif", methodDeclaration);
    if ((Boolean) methodQuery.getResult()) return true;
  }
  return false ;
}
```

Listing 9.14 CDO OCL query for the TypeDeclaration check

Metric	**Meaning**	**Value**
NQ	Number of queries against the persistent storage	3
DC	Majority of code is declarative	False
TC	Absence of type castings and checks	False
RC	Absence of mixed non-query code	False
CS	Consistency with established query languages	True
SY	Syntactic checking prior to execution	False
SE	Semantic checking prior to execution	False
IN	Independence from external tools	True
FB	Existence of a formal basis	False

Table 9.9 Evaluation of the **effectiveness**, **usability** and **safeness** metrics for the CDO OCL approach

The **effectiveness** of this approach is *low*. Three separate queries are needed to obtain the results: one to get all the TypeDeclarations, one to get all their MethodDeclaration and a last one to check each one of the latter. As happens with MDT OCL, if the original OCL query (see Listing 9.13) could have been used, the effectiveness would have been *high*.

The **usability** of this approach is *low*: it shows the same issues as MDT OCL (Java statements to relate the results of each query), but in this case even the original query shown in Listing 9.13 is full of conditional statements and type checks because of the implementation of the server side of CDO. If the original query could have been used, the readability would have been *high*.

The **safeness** of this approach is *low*: no syntactic or semantic checking is done before the query is executed, as the queries are specified as text strings. Moreover, it is even worse than MDT OCL, since the errors are returned as exceptions with very little detail.

9.2.6 Morsa Query Language

As happened with other studied approaches, the test query must be split into two separate steps, one for obtaining each TypeDeclaration and another one to check it. The first step is implemented by the code shown in Listing 9.4; since MorsaQL can only be used with the Morsa persistence solution, there are not any other options to get the TypeDeclarations as it happens with the other previous approaches. Listing 9.15 shows the implementation of the checkTypeDeclaration method that performs the second step.Table 9.10 shows the evaluation of the metrics for this approach.

```
//With dynamic EMF
boolean checkTypeDeclaration(EObject typeDeclaration) {
  EPackage domPackage = typeDeclaration.eClass().getEPackage();
  EClass typeDeclarationClass = typeDeclaration.eClass();
  EClass nameClass = (EClass)domPackage.getEClassifier("Name");
  EObject typeDeclarationName = (EObject)typeDeclaration
     .eGet( typeDeclarationClass . getEstructuralFeature ("name"));

  String typeDeclarationFQName = (String)typeDeclarationName
     .eGet(nameClass.getEStructuralFeature(" fullyQualifiedName"))

  MorsaQuery query = SELECT("org.amma.dsl.jdt.dom/MethodDeclaration")
     .FROM(typeDeclaration)
     .WHERE(navigate("returnType")
       .ofType("org.amma.dsl.jdt.dom/SimpleType")
       . navigate("name").STREQ("fullyQualifiedName", typeDeclarationFQName))
     .AND(navigate("modifiers")
       .ofType("org.amma.dsl.jdt.dom/ExtendedModifier")
       .BOOLEQ("static", true))
     .AND(navigate("modifiers")
       .ofType("org.amma.dsl.jdt.dom/ExtendedModifier")
       .BOOLEQ("public", true))
     . asProxies() . includeSubtypes() . limit (1) .done();

  Collection <EObject> result = ((MorsaResource)resource).query(query);
  return ! result .isEmpty();
}

//With generated EMF
boolean checkTypeDeclaration(TypeDeclaration typeDeclaration ) {
  MorsaQuery query = SELECT(DOMPackage.getMethodDeclaration())
     .FROM(typeDeclaration)
     .WHERE(navigate("returnType")
       .ofType(DOMPackage.getSimpleType())
       . navigate("name").STREQ("fullyQualifiedName",
         typeDeclaration .getName().getFullyQualifiedName()))
     .AND(navigate("modifiers")
       .ofType(DOMPackage.getExtendedModifier())
       .BOOLEQ("static", true))
     .AND(navigate("modifiers")
        .ofType(DOMPackage.getExtendedModifier())
        .BOOLEQ("public", true))
    . asProxies () . includeSubtypes () . limit (1) .done();

  Collection <EObject> result = ((MorsaResource)resource).query(query);
  return ! result .isEmpty();
}
```

Listing 9.15 Morsa Query Language query for the TypeDeclaration check

Metric	Meaning	Value
NQ	Number of queries against the persistent storage	2
DC	Majority of code is declarative	True
TC	Absence of type castings and checks	True
RC	Absence of mixed non-query code	True
CS	Consistency with established query languages	True
SY	Syntactic checking prior to execution	True
SE	Semantic checking prior to execution	False
IN	Independence from external tools	True
FB	Existence of a formal basis	False

Table 9.10 Evaluation of the **effectiveness**, **usability** and **safeness** metrics the Morsa Query Language approach

The **effectiveness** of this approach is *medium*: two separate queries are needed to obtain the results: one to get all the TypeDeclarations and one to check each one of them. As happens with EMF Query, this is due to the lack of named variables.

The **usability** of this approach is *high*: thanks to its implementation as an internal DSL, there is little or no Java code in between the query statements, which are completely declarative, and no type checks or casts are done; instead, the type constraint for an intermediate result is embedded on the condition that calculates it, and only if necessary. Moreover, the concrete syntax mimics SQL, which is a well-established query language.

The **safeness** of this approach is *medium*: syntactic checking is provided by the internal DSL, although there is no semantic checking, and the integration between the query language and the client applications is clean, since no external tools are needed. However, there is no formal basis.

9.3 Efficiency evaluation

In this section, the **efficiency** dimension defined in Section 7.3 will be evaluated against each model querying approach using XMI, CDO and Morsa. This section is organized as follows: (i) the evaluation scenario will be described; (ii) the test results for each model will be explained and finally (iii) the results for Morsa and MorsaQL will be commented.

We have tested the combinations between persistence solutions and querying approaches shown in Table 9.1 (except the ones marked as *Not available* or *Not considered*). Each combination has been configured to give the best balance between time and memory consumption.

9.3.1 Evaluation scenario

The execution environment has been an Intel Core i7 2600 PC at 3.70GHz with 8GB of physical memory running 64-bit Fedora Core 17 and OpenJDK JVM 1.7. CDO has been configured using DBStore over a dedicated MySQL 5.0.51b database. Morsa has been deployed over a MongoDB 1.8.2 database.

The test query has been executed twice over each test model and each of the combinations defined above. Six measurements have been performed: the time and memory it takes to initialize the combination (e.g. model load from XMI or metamodel load from Morsa), the time and memory it takes to execute the first query and the time and memory it takes to execute the second query. We have executed the same query twice to evaluate also the performance of the caching mechanisms of the different combinations.

9.3.2 Results for every approach

Figures 9.1 to 9.5 show the results for the different test models. We could not store models Set3 and Set4 in CDO. The Init Time, Init Mem, Query 1 Time, Query 1 Mem, Query 2 Time and Query 2 Mem labels stand for initialization time and memory, first query time and memory and second query time and memory, respectively.

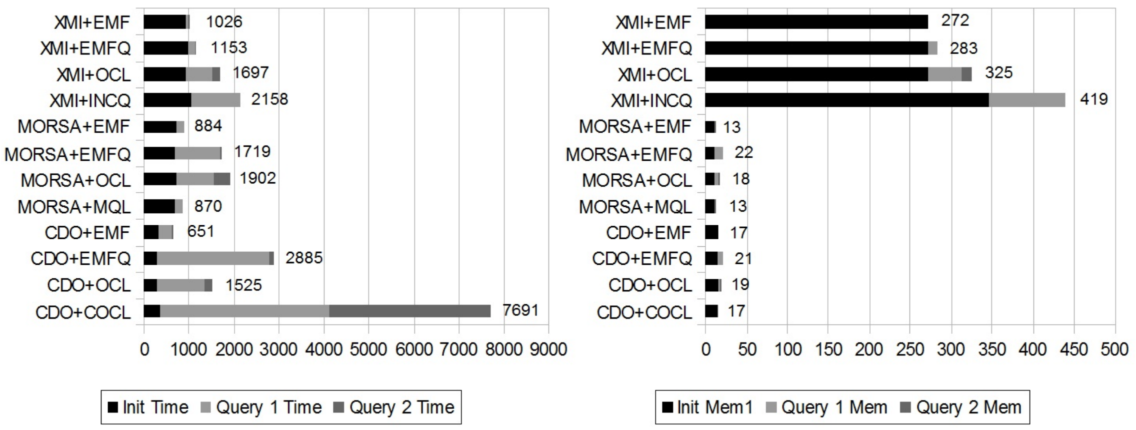


Figure 9.1 Time (milliseconds) and memory (MegaBytes) consumption for Set0

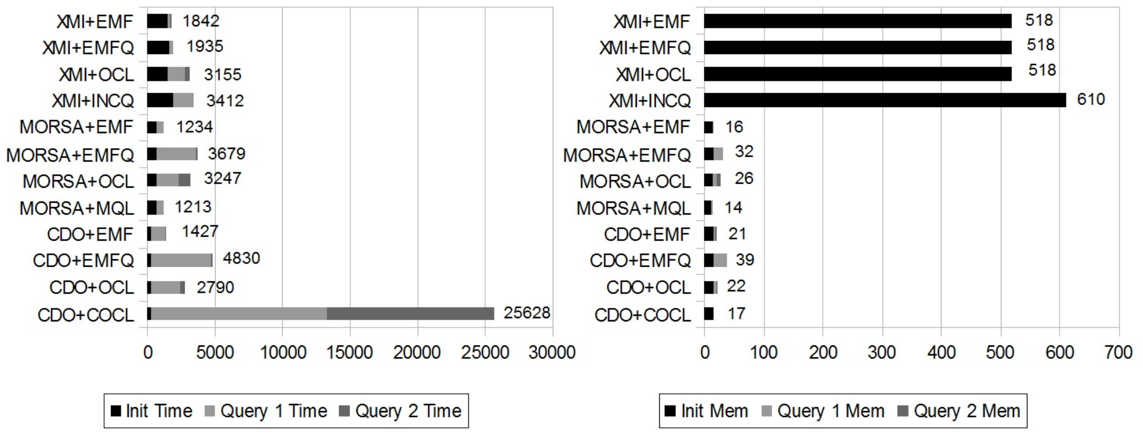


Figure 9.2 Time (milliseconds) and memory (MegaBytes) consumption for Set1

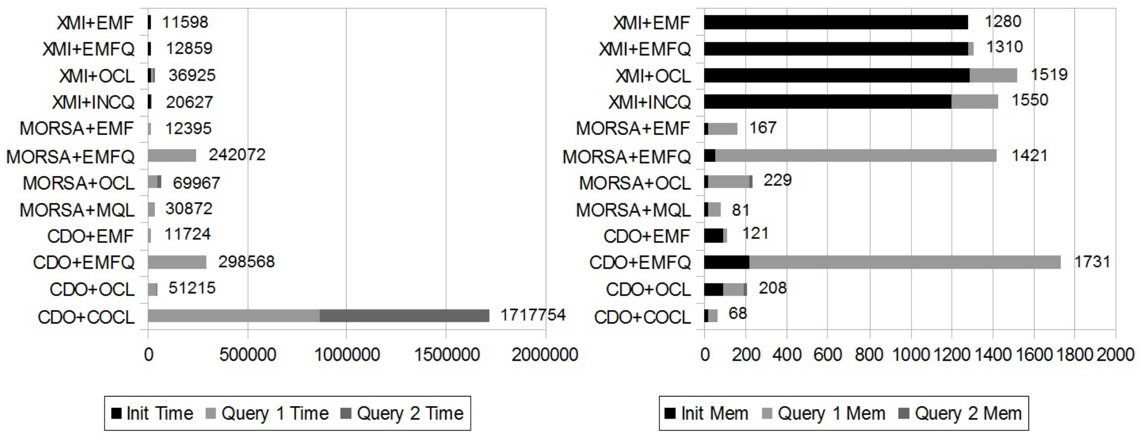


Figure 9.3 Time (milliseconds) and memory (MegaBytes) consumption for Set2

The total time or memory is shown on the right side of each bar. Note that for the XMI combinations, most of the time and memory is spent on initialization (i.e. model load), while the Morsa and CDO ones spent most of its time and memory on the first query. An exception for this is CDO+COCL, which spends almost the same time for the first and the second query and uses very little memory, as the query as calculated by the server. It is worth noting that, although it cannot be perceived on the figures, the evaluation time for the second query (and hence, every other one past it) for IncQuery is less than one millisecond, so for many repeated queries it would perform better than the rest; the opposite happens to plain EMF, EMF Query and MDT OCL, which need almost the same time for each query as the size of the input model grows.

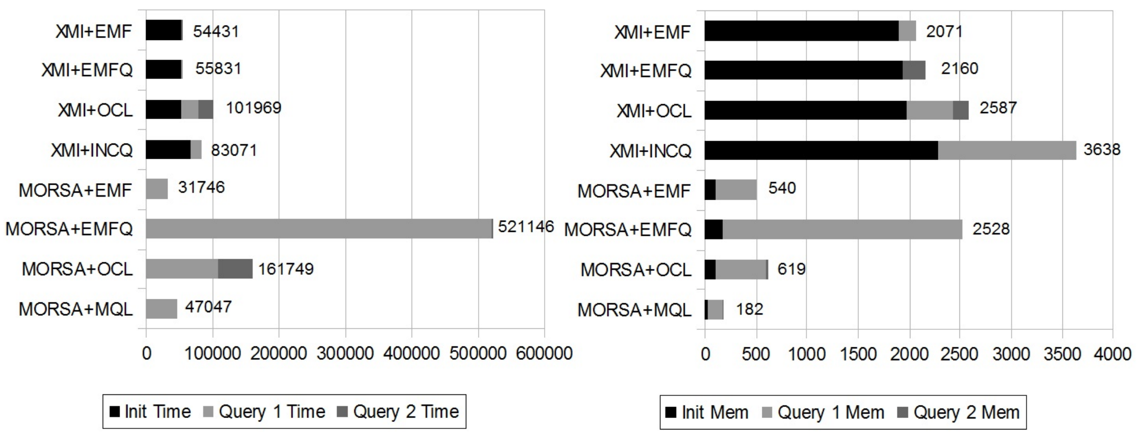


Figure 9.4 Time (milliseconds) and memory (MegaBytes) consumption for Set3

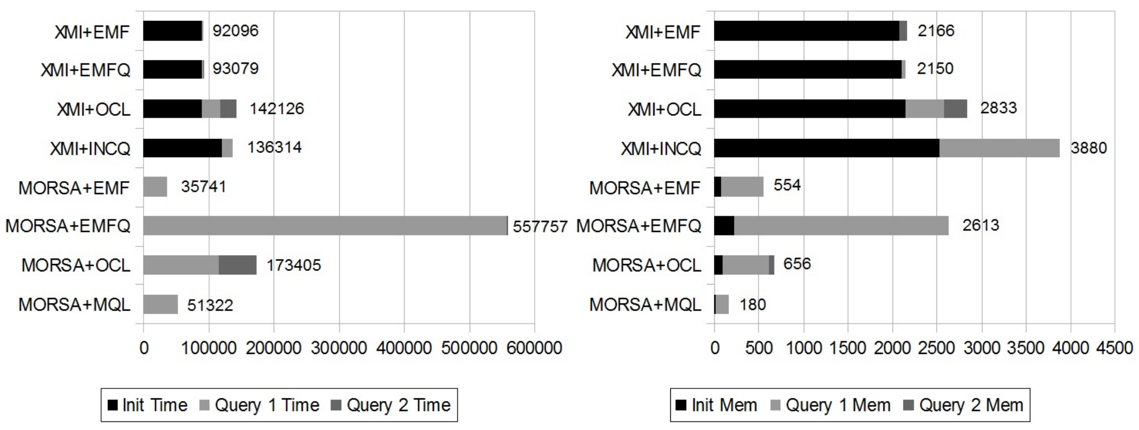


Figure 9.5 Time (milliseconds) and memory (MegaBytes) consumption for Set4

Overall, Morsa and CDO use much less memory and are faster than XMI, and their fastest combinations are with plain EMF. Morsa is usually faster and uses less memory than CDO. Morsa is only beaten in memory by CDO+COCL, which uses almost the same client memory for all the test cases thanks to its server-side querying, but is between 8 and 30 times slower than Morsa. It is worth noting that EMF Query performs very badly over both repositories and MDT OCL performs worse than EMF Query over XMI but better over Morsa and CDO.

9.3.3 Results for Morsa

Focusing only on the performance of the Morsa combinations, Figure 9.6 shows the evolution of performance of the different querying approaches over Morsa: the memory and time consumption of EMF Query grows dramatically as the size of the test models grows, while the rest of the approaches grow at a more lineal pace.

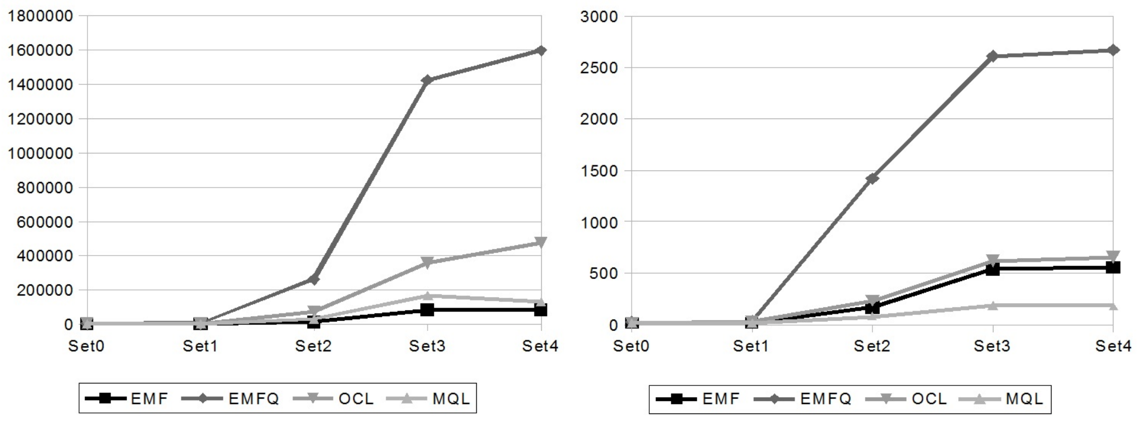


Figure 9.6 Evolution of the time (left, in milliseconds) and memory (right, in MegaBytes) consumption of the different querying approaches in Morsa between the test models

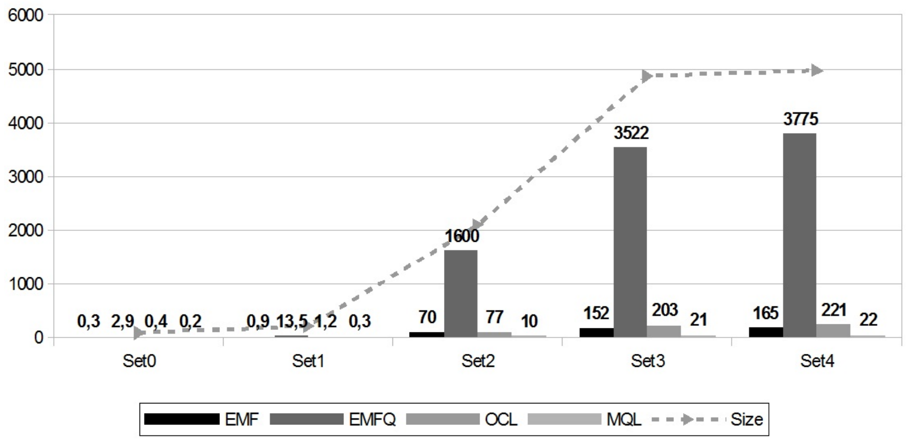


Figure 9.7 Number of objects (in thousands) loaded from the Morsa repository for each test model and querying approach. The size of the test models is represented by the dashed line

Figure 9.7 shows how many model elements are actually accessed by each querying approach compared to the actual size of each model; as can be seen, the poor performance of EMF Query is directly related to the amount of model elements it loads, while the good memory performance of MorsaQL is based on its low element demand. In fact, the number of elements that MorsaQL requests to the repository for each result is quite low, as shown in the right side of Figure 9.8; this gives an idea of the efficiency of the query language (i.e. an ideal query language will only load the results). Moreover, the left side of Figure 9.8 shows that the bigger the model is queried, the less additional model elements are loaded from the repository relatively to each of its TypeDeclarations.

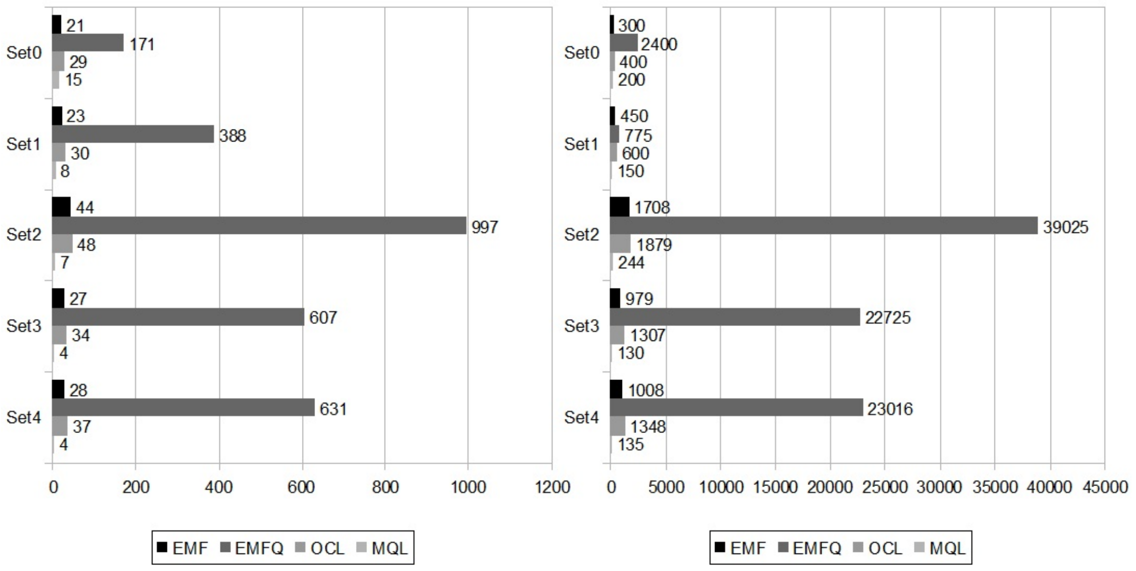


Figure 9.8 Number of objects loaded from the Morsa repository for each test model and querying approach per TypeDeclaration (left) and per result (right)

9.4 Overall results

It is difficult to compare the results of every tested combination, since the behavior of the same querying approach differs depending on which persistence solution is accessing. In order to make a numerical comparison feasible, we have assigned a score to each dimension, following these rules:

- For the **effectiveness** dimension, considering that the maximum value achieved for the *NQ* metric is infinity and the minimum is 1, the score is 0 for infinity and *4* - *NQ* for the rest.

- For the **usability** and **safeness** dimensions, the score is the sum of the amount of `true` values obtained in their metrics.
- For the **efficiency** dimension, the score is relative to their rank for a given persistence solution compared to the other approaches: the best performer scores 4 points, the second scores 3, the third scores 2 and the worst scores 1. This is so because each efficiency measurement is dependent from a persistence solution and should not be compared between persistence solutions.

Given these rules, the **efficiency** dimension has a score from 1 to 4 and the rest have a score from 0 to 4. Table 9.11 shows the evaluation results using the above defined scoring schema. Note that although IncQuery is the least efficient approach over XMI, it might turn out to be the best if the same queries are executed many times, thanks to its indexes.

Name	**Effectiveness**	**Usability**	**Safeness**	**Efficiency XMI**	**Efficiency Morsa**	**Efficiency CDO**
Plain EMF	0	0	3	4	3	4
EMF Query	2	1	2	3	1	2
MDT OCL	1	3	1	2	2	3
IncQuery	3	3	3	1	-	-
CDO OCL	1	1	1	-	-	1
MorsaQL	2	4	2	-	4	-

Table 9.11 Evaluation results

A ranking of querying approaches is something mainly subjective, because it depends on the preferences or needs of the user: for instance, if **usability**, **safeness** and **effectiveness** are a must and **efficiency** and *portability* are not very relevant, the ideal choice would be IncQuery, since its external DSL makes it very easy to write queries; on the other hand, if **efficiency** and *portability* are mandatory, MorsaQL should be used. However, considering every dimension equally relevant, a ranking can be made for each persistence solution, as shown in Table 9.12; in the case of a tie, the approach with the highest mode wins. As can be seen, if **efficiency** would have been the priority, plain EMF would have ranked higher.

Focusing on Morsa, MorsaQL is the best option because it overall scores the highest marks. Given that our main design goal was to achieve **efficiency**, that dimension should have been assigned more weight when building the Morsa ranking, which would have made MorsaQL even better than the other approaches.

Rank	**XMI**	**Morsa**	**CDO**
1	IncQuery (10)	MorsaQL (12)	MDT OCL (8)
2	EMF Query (8)	MDT OCL (7)	EMF Query (7)
3	MDT OCL (7)	EMF Query (6)	Plain EMF (7)
4	Plain EMF (7)	Plain EMF (6)	CDO OCL (4)

Table 9.12 Querying approach ranking by persistence solution

10 Conclusions and Further Work

Man, you should
have seen them
kicking Edgar Allan Poe

This chapter presents our conclusions and further work. To do so, we have categorized the Morsa repository and described it in terms of the categories and dimensions defined in Chapter 3; a comparison between Morsa and the model persistence approaches evaluated in Section 3.3 is also discussed. We also propose some research guidelines based on our experience and on the results of the comparison between model persistence approaches. Finally, the publications, tools, projects and grants related to this thesis are listed after the furter work is commented.

10.1 Categorization of the Morsa repository

In order to classify Morsa in one of the three categories defined in Section 3.3.1, it must be characterized using the dimensions on model persistence explained in Section 3.1. Figure 10.1 shows a feature diagram that depicts the features covered by Morsa:

1. **Storage medium**: Morsa uses a *database* to store models, which in theory may be of any kind, although at this moment only a *NoSQL* one has been implemented. Its identification schema is *loosely coupled*, as it has been implemented with UUIDs (*MorsaIDs*, see Section 5.1.2).
2. **Architecture**: a *fat client* with *offline* coupling implements the client-server architecture of Morsa (see Section 5.1.1).
3. **Access**: Morsa supports all kinds of *Access* using an *in-memory* local copy. A simple *prefetching algorithm* guided by containment relationships is used for load on demand. No transaction support is provided. See Sections 5.2-5.4 for more detail.

4. **Query**: queries may be specified in a dedicated language implemented as an *internal DSL* (MorsaQL, see Chapter 8) and also as an *API* that provides methods that internally rely on MorsaQL for common queries such as getting descedants, ancestors, instances of a given metaclass, etc.
5. **Transparency**: a *customized* integration is achieved. Metamodels are represented in an *explicit* and *read & only* way without any preprocessing or manual registration achieving *total* transparency (see Section 5.1.2).
6. **Version control**: Morsa does not support any version control.
7. **Client interface**: Morsa provides no user interface and its main interface is an EMF *framework* API (see Section 5.5).

Section 3.3.1 defined four different categories for model persistence solutions: *default*, which is the one provided by the modeling framework, usually supporting only few features; *user-oriented*, which provides a rich user interface but has no focus on scalability and integration; *application-oriented*, which provides a poor user interface but a rich API; and *integrated*, which may easily provide every feature since its development has been paired with the one of the modeling framework. Considering this, we can consider that Morsa is an *application-oriented* repository, that is, a persistence solution designed to be integrated with applications. It provides no user interface, but a rich API instead, and it is focused on scalability and integration.

10.2 Comparison with other persistence approaches

Compared to XMI (see Section 3.2.1), the *default* model persistence approach for EMF, Morsa is a better model persistence approach, since it provides many more features. Although Morsa achieves much lower performance than XMI in operations that imply the full traversal of models (see Sections 6.2.1 and 6.2.2), its native support to model querying and its higher performance for load on demand (see Chapter 9) make it a better option for this operations.

Compared to the *user-oriented* persistence solutions analyzed in Chapter 3, that is, ModelBus (see Section 3.2.2) EMFStore (see Section 3.2.3) and CDO (see Section 3.2.4), it can be noted that the former two provide substantially less *Access* and *Query* features than Morsa and their *Transparency* dimension is far less than what Morsa achieves; however, their *Client interface* capabilities are way more advanced, providing useful graphical interfaces that give support to a rich set of *Version control* features (less rich for EMFStore), while Morsa does not provide any of these characteristics.

With regard to CDO, it covers most of the dimensions at at least the same extent as Morsa, particularly the *Version control* and *Client interface* ones, which are unarguably better covered by CDO, as Morsa provides neither of them; the *Access* and *Query* dimensions are covered by CDO at the same extent as Morsa, but as shown in Chapters 6 and 9, they do not achieve the same efficiency and effectiveness than the ones in Morsa. Moreover, the *Transparency* achieved by CDO is lower than the one of Morsa, since only *parallel semantic transparency* and an *implicit, read-only metamodel management* are achieved by the former.

Compared to MongoEMF (see Section 3.2.5), the only *application-oriented* persistence approach evaluated in Chapter 3, Morsa supports more features of the *Access* dimensions, particularly *single load on demand*, a proper *partial load on demand* and a *prefetching algorithm*. Moreover, Morsa provides both an *API* and an *internal DSL* for querying (i.e. MorsaQL), while MongoEMF only provides the former. The degree of *Transparency* achieved by Morsa is also superior than the one of MongoEMF, which requires *manual registration* of metamodels and represents them in an *implicit* way.

Compared to OOMEGA (see Section 3.2.6), the only *integrated* persistence approach evaluated in Chapter 3, Morsa provides less features, which is normal, as both the modeling framework and the model persistence support in OOMEGA have been designed together, while Morsa had to be integrated with EMF. However, Morsa provides *offline server coupling* and does not require *manual* model registration, two features not supported by OOMEGA.

10.3 Conclusions

One critical concern for the industrial adoption of MDE is the scalability of tools when accessing large models. As commented in [KRM+13], achieving scalability in MDE involves four goals: (i) the ability to construct large models, (ii) the support to collaborative modeling, (iii) the creation of efficient model querying approaches and (iv) the efficient storage, indexing and retrieval of large models. As shown in this thesis, we have tackled the third and fourth goals by defining new approaches and tools in the model persistence and model querying research areas.

Model persistence and model querying are two emerging areas of research, as demonstrated by the amount of persistence approaches that are being developed currently as either commercial products (ModelBus, EMFStore, CDO and OOMEGA) or research developments (MongoEMF) and the querying approaches available for the EMF modeling framework (EMF Query, MDT OCL, CDO OCL and IncQuery). These proposals extend modeling frameworks like EMF with capabilities that make them more adapted to industrial-scale projects, where large models are often produced and manipulated, distributed development teams, where versioning is convenient, and model-driven tools like model transformations, where accessing only parts of a model using load on demand and model querying can be critical for performance.

In order to develop our own model persistence and model querying approaches, we studied the state of the art of both model persistence (see Chapter 3) and model querying (see Chapter 7). To do so, we identified a set of dimensions for each of the areas that helped us identifying both the features that a model persistence approach should provide and the criteria for evaluating model querying approaches; moreover, we used these dimensions to characterize six different model persistence approaches (see Section 3.2) and five different model querying approaches and (see Section 7.5) and to compare them (see Section 3.3 and Chapter 9, respectively).

With the knowledge gained by studying the state of the art on model persistence and model querying, we designed and implemented Morsa, a model repository aimed at achieving **scalability** for client applications that access large models. The architectural and data design of Morsa (see Section 5.1) were devised to support fine-grained access to large models that allow for *load on demand* and *incremental store*, using a separate collection for each model instead of a single one (as CDO does) and a *loosely coupled* identification schema paired with metadata inside every MorsaObject that gives support to model querying and *partial load on demand* by representing the structure of the stored models in terms of *containment relationships* (see Sections 5.1.2, 5.3.1.2 and 8.4).

The architectural design of Morsa is composed of a *fat client* (see Section 3.1.2.2) implemented by MorsaDriver and a database backend that is abstracted by a MorsaBackend component, which represents the database as a set of MorsaObjects (i.e. a set of key-value pairs). The MorsaDriver uses an ObjectCache (i.e. a cache of MorsaObjects and a CachePolicy (i.e. a cache replacement policy) components to decide what model elements should be retrieved from the database backend and what model elements are no longer necessary and hence can be *unloaded* from the client application to save memory. The MorsaDriver component implements the persistence interface of the modeling framework to provide **tool integration**, requiring very few changes in existing applications in order to use Morsa as their persistent storage (see Section 5.5).

Using *load on demand* and *incremental store*, Morsa allows large models to be persisted and accessed without overloading the client application's memory, as demonstrated in Chapter 6, where a prototype of Morsa for EMF and MongoDB is compared to XMI and CDO. This comparison consisted in executing four benchmarks against large models taken from the Grabats 2009 contest [JS09] (see Chapter 4), demonstrating that Morsa suits better for partial model access than XMI and CDO, and that it handles larger models than CDO.

Once we had an stable prototype of our model repository, we designed and implemented a query language for Morsa, namely MorsaQL, as presented in Chapter 8. The main design goals of MorsaQL are its **efficiency**, **usability** and, to a lesser extent, its **safeness**, as defined in Section 7.3.

The *abstract syntax* (see Section 2.5) of MorsaQL (see Section 8.2) provides different concepts that allow the definition of *contexts* (i.e. search spaces) and *conditions* (i.e. constraints) on attributes and relationships, the navigation of relationships and the parameterization of the execution of a query in terms of *depth*, *breadth*, object count and *proxy* resolution (see Sections 2.3 and 5.3.1.2).

The *concrete syntax* (see Section 2.5) of MorsaQL (see Section 8.3) has been implemented as an *internal DSL* that resembles SQL and EMF Query, the most used query language and the recommended model querying approach for EMF, respectively, for better *consistency* (see Section 7.3). MorsaQL is implemented using a combination of the *method chaining* and *nested function* patterns described in Section 2.5 that uses the *expression builder* pattern in the form of the QueryBuilder class. The resulting concrete syntax is very *readable*, *writable* and provides *syntactic checking* at coding time; *semantic checking* is done at runtime. Moreover, thanks to its internal DSL nature, MorsaQL can be combined with Java statements to improve its functionality.

The *semantics* (see Section 2.5) of MorsaQL (see Section 8.4) involves the representation of the structure of the stored models provided by the metadata inside every MorsaObject (see Section 5.1.2) when querying the database backend, achieving good performance thanks to a rather low ratio between loaded elements and query results (see Section 9.3.3). Queries are represented also as MorsaObjects following a *query-by-example* model. A cache of query results is maintained by the MorsaDriver to prevent accessing the database backend whenever is possible.

In addition to the Morsa model repository and the MorsaQL model query language, this thesis presents an analysis, evaluation and comparison of seven different model persistence and six model querying approaches (including Morsa and MorsaQL), which is also very helpful for guiding MDE developers on which approaches they can use, depending on their needs. Comparative charts can be found in Sections 3.3, 6.2 and 9.4. The comparison between Morsa and the rest of the analyzed model persistence approaches is done in Section 10.2.

10.4 Research guidelines

Based on our experience, the comparison and categorization made in Chapter 3 and the evaluation made in Chapter 9, we have identified two main research lines that should be a priority on the area of model persistence: convergence and benchmarking.

The *convergence* between user-oriented and application oriented model persistence solutions into solutions that provide the features of both should be addressed in a near future in order to provide both the rich interfaces and versioning capabilities of the former and the scalability and integration of the latter. This should be a natural trend since every kind of client would benefit from using such a persistence solution: human users would use a rich, friendly interface suited for distributed teamwork and client applications such as model transformation would run efficiently in a scalable environment.

However, currently user-oriented persistence solutions are the ones that are getting the most attention and spreading in the MDE community, being CDO the most used and the one with a more mature and active development. While each new version of CDO adds new capabilities for concurrent distributed teamwork, scalability and integration do not get much attention, if any. On the other hand, application-oriented persistence solutions which are focused on scalability and integration, such as MongoEMF or Morsa, are academic researchs that have no focus on teamwork or versioning at the moment.

We think that the technology and knowledge are mature enough to align both categories of persistence solutions keeping the best of each one, so a solid foundation on model persistence is built; then, on top of this foundation of scalability and services, other persistence solutions could be built focusing on specific tasks and scenarios, in the same way as DSLs are built on top of the foundation provided by metamodeling frameworks and transformations. Recent publications are starting to value this approach, e.g. [KRM$^+$13] defines the extensibility of a model repository as a research direction in order to create customizable tools where components for different management issues, such as model versioning, can be plugged in.

Benchmarking of persistence solutions should also be addressed. Just as in the database area [Cat93], the definition of benchmarks for model persistence could be very helpful for the users; for instance, they may give guidance to the developers of MDE solutions when choosing the tool that best fits their needs. However, few attempts have been made on comparing the performance of different persistence solutions [SZFK12][JS09]. This is due to the complexity of developing a benchmark that gives homogeneous performance measurements for the variety of access schemas provided by the different persistence solutions; for instance, it is hard to properly compare a solution that uses file sets with full load and another one that supports partial load on demand over a database. In addition, few benchmarks have been defined to measure the scalability of a persistence solution for both the client and the solution when loading, storing, updating, deleting and querying big models.

10.5 Further work

This thesis tackles both the problem of model persistence and the problem of model querying, so our further work will address both research areas.

10.5.1 Model persistence

With regard to model persistence, our future work is to extend Morsa in order to provide new features to it, as well as to optimize the existing ones.

From an architectural point of view, Morsa is a *fat client*, which is an architecture that has some drawbacks, as explained in Section 3.1.2.2, being the lack of change synchronization between users the most relevant to us; thus, our further work includes refactoring Morsa into an *n-layer* architecture, where the communication between a MorsaDriver running in a client application and a database backend is managed by a dedicated server. This is not a simple task, as the server should not become a bottleneck due to the synchronization of changes between clients.

Once an *n-layer* architecture is obtained, *version control* capabilities (see Section 3.1.6) could be implemented. As this would require deep changes in the data design of Morsa in order to store deltas, revisions and other versioning information, we haven't studied this functionality in depth yet. Version control is one feature that is requested by the MDE community, as stated in [KRM$^+$13], but that hasn't been successfully fulfilled yet.

The optimization of Morsa can be tackled not only through code analysis and debugging, but also by including more metadata and complex algorithms in the repository. In order to obtain more efficient *prefetching algorithms* and cache replacement policies, metadata about the structure of a metamodel and a model could be extracted statically; this may include: average depth and breadth of the relationships between model elements, semantic dependencies between relationships or metaclasses, etc. This metadata may also be beneficial for model querying. This would tackle the research direction of indexing models defined in [KRM$^+$13], providing rich metadata that can work as indexes.

Finally, since cloud computing is becoming very relevant nowadays, we are considering the implementation of a REST [Fie00] interface to Morsa. This would require an *n-layer* architecture, since the interface should be repository-oriented, not database-oriented. This would provide access to the repository from many more technologies than just EMF, although the representation and the functionality of the loaded model elements would be different, mainly because a REST interface can be directly accessed from applications such as web browsers, that do not know anything about MDE and may not use repository drivers.

10.5.2 Model querying

With regard to model querying, our future work is to enhance the **effectiveness** and **efficiency** of MorsaQL.

In order to enhance the **effectiveness** of MorsaQL, the key feature to develop would be the definition and use of *named variables*, that is, variables that are assigned subquery results and can be used by conditions. Introducing this feature is not straightforward, since it requires complex syntactic and semantic checking, but it would be very beneficial for the **effectiveness** of MorsaQL, since a query such as the one defined in Chapter 4 could be performed with a single MorsaQL query (a named variable would be used to store the fullyQualifiedName of a TypeDeclaration) instead of two.

The **efficiency** of MorsaQL could be improved implementing the already mentioned retrieval of metadata about (meta)model structure and also by developing a more advanced caching mechanism that provides a querying mechanism as incremental as possible, so changes on model elements do not invalidate whole sets of query results in the cache, but only the ones directly involved in the change.

10.6 Publications

10.6.1 Journals with impact factor

- J. Espinazo-Pagán, J. Sánchez and J. García-Molina. A Repository for Scalable Model Management. *Software and Systems Modeling*, 2013. Published online at: http://link.springer.com/article/10.1007%2Fs10270-013-0326-8# DOI: 10.1007/s10270-013-0326-8.

  Impact Factor: 1,250 (32/105, 1st third of JCR/Software Engineering)
- J. Espinazo-Pagán and J. García-Molina.Querying Large Models Efficiently. *Information and Software Technology*. Under review.

  Impact Factor: 1,522 (23/105, 1st third if JCR/Software Engineering)
- J. Espinazo-Pagán and J. García-Molina. A Survey on Model Persistence. *ACM Computing Surveys*. Under review.

  Impact Factor: 3,543 (3/100, 1st third of JCR/Theory and Methods)

10.6.2 Conferences of a quality similar to a journal due to their acceptance rate

- J. Espinazo-Pagán, J. Cuadrado and J. García-Molina. Morsa: a Scalable Approach for Persisting and Accessing Large Models. At the *International Model Driven Engineering Languages and Systems (MoDELS) Conference*, vol. 6981, pp 77-92, Wellington, New Zealand, 2011. Springer-Verlag.

  Acceptance Rate: 20% (chosen as one of the five most relevant contributions) *14 external cites*

- J. Espinazo-Pagán, M. Mernárguez and J. García-Molina. Metamodel Syntactic Sheets: An Approach for Defining Textual Concrete Syntaxes. At the *European Conference on Model Driven Architecture - Foundations and Applications*, vol. 5095, pp 185-199, Berlin, Germany, 2008. Springer-Verlag.

  Acceptance Rate: 30%

10.6.3 International conferences/workshops

- J. Espinazo-Pagán and J. García-Molina. A Homogeneous Repository for Collaborative MDE. At the *International Workshop on Model Comparison in Practice* of the *International Conference on Objects, Models, Components, Patterns*, pp 55-65, Málaga, Spain, 2010. ACM.

10.6.4 National conferences/workshops

- J. Espinazo Pagán, J. Sánchez Cuadrado, and J. García Molina. Un repositorio NoSQL para acceso escalable a modelos. At the *Jornadas de Ingeniería del Software y Bases de Datos*, pp 521-534, Almería, Spain, 2012.

10.7 Developed tools

The Morsa model repository (including the MorsaQL query language) is available as an Eclipse plugin at http:/www.modelum.es/morsa/. The documentation of the repository is available at http://www.modelum.es/morsa/doc/morsa/ and http://www.modelum.es/morsa/doc/morsa.mongodb.

10.8 Projects that have used the results of this thesis

The results of this thesis have been applied to the following projects:

- **Impulso de la investigación en tecnologías del Desarrollo de Software (Un entorno para el desarrollo y modernización basados en modelos Forms-ADF)**

  Funding entity: CARM, Consejería de Universidades, Empresa e Investigación. Programa PEPLAN. Subvenciones a proyectos estratégicos en el Plan Regional de Ciencia.

  Duration: 01/01/2009 — 31/12/2010.

  Main researcher: Dr. Jesús Joaquín García Molina (Universidad de Murcia).

The goal of this project was the definition of a software environment for the migration of Oracle Forms applications to ADF. The Metarep model repository was developed as part of this project to enhance the collaboration of the development team when producing (meta)models.

- **Construcción de un lenguaje de modelos tipado estáticamente y con facilidades modulares**

  Funding entity: Ministerio de Ciencia e Innovación TIN2009-11555.

  Duration: 01/01/2010 — 31/07/2012.

  Main researcher: Dr. Jesús Joaquín García Molina (Universidad de Murcia).

  The goal of this project was the development of a statically typed, modular transformation language. Morsa was initially designed to serve as the persistence layer for this language, providing a rich API for querying and performing load on demand.

- **Herramienta orientada a la migración basada en modelos**

  Duration: 01/01/2010 — 31/12/2011.

  Main researcher: Dr. Jesús Joaquín García Molina (Universidad de Murcia).

  This project was aimed at the creation of a tooling to assist the automatic migration of Oracle Forms applications to a Java platform. Morsa was used in this project as a repository to hold the large models that were extracted from the legacy Oracle Forms applications, which included PL/SQL code, widget definitions, database schemas, etc.

- **GUIZMO: Un framework para la modernización basada en modelos de interfaces de usuario**

  Funding entity: CARM, Ayudas a Proyectos de Investigación, Fundación Séneca.

  Duration: 01/01/2012 — 31/12/2013.

  Main researcher: Dr. Jesús Joaquín García Molina (Universidad de Murcia).

  The goal of this project is the development of a model-driven framework for the automatic migration of graphical user interfaces. Both this framework and Morsa where used in the context of the previous project to develop a semiautomated tool for migrating Oracle Forms applications.

10.9 Grants

During the development of this thesis, the candidate enjoyed the following grant:

- **Beca asociada a la realización de proyectos de i+d, innovación y transferencia de tecnología**.

  Funded by Agencia Regional de Ciencia y Tecnología, Fundación Séneca from July 2010 until July 2013, allowed the candidate to develop the Morsa repository and the above mentioned project.

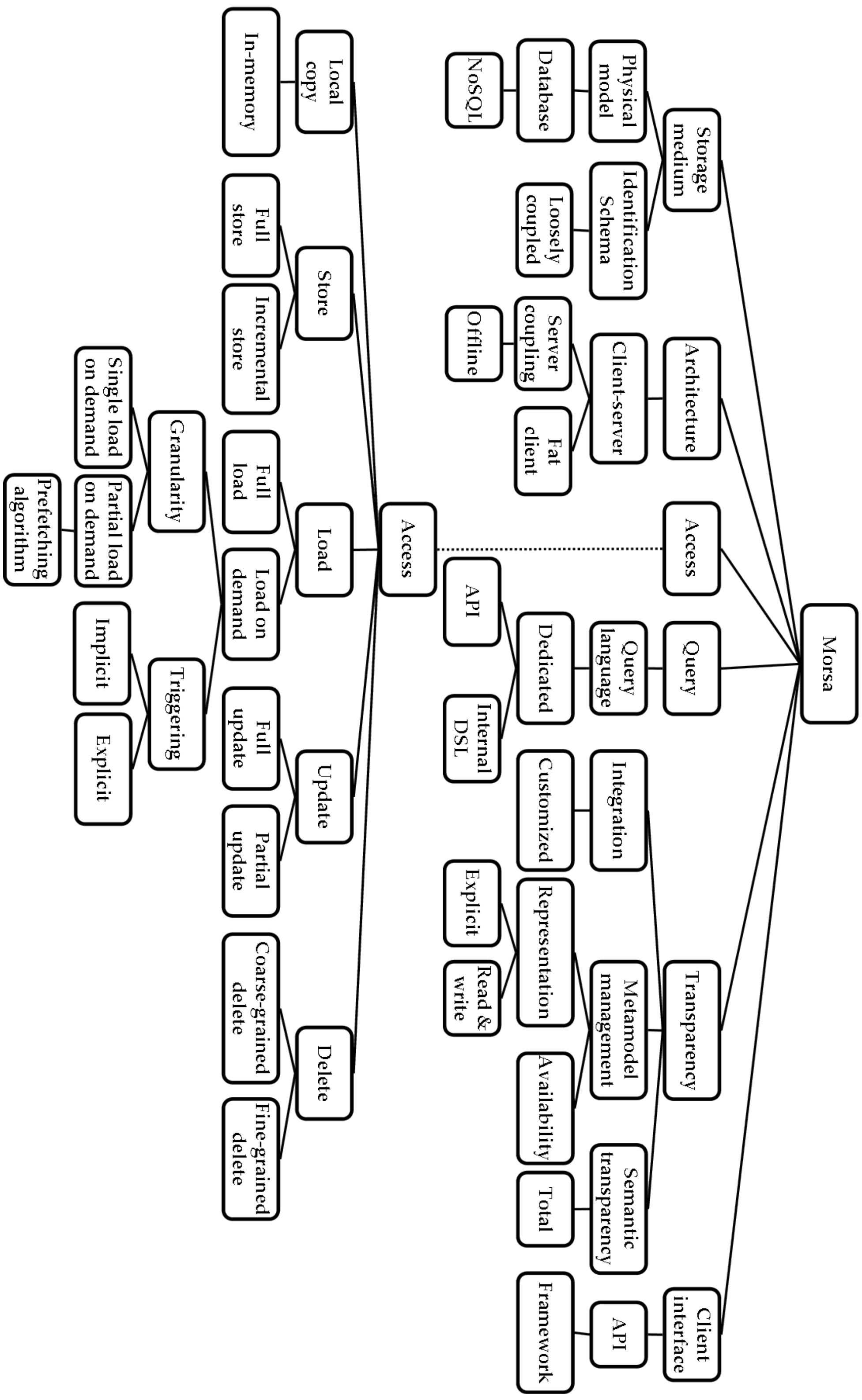


Figure 10.1 Feature model of the Morsa repository according to the dimensions explained in Chapter 3

Bibliography

[ADM07] ADM. Architecture-Driven Modernization: Transforming the enterprise. http://adm.omg.org, 2007.

[ASW09] K. Altmanninger, M. Seidl, and M. Wimmer. A Survey on Model Versioning Approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

[Ban11] K. Banker. *MongoDB in Action*. Manning, 2011.

[BBF09] G. Blair, N. Bencomo, and R. France. Models@ run.time. *IEEE Computer Society*, 42(10):22–27, October 2009.

[BCHK08] B. Bruegge, O. Creighton, J. Helming, and M. Koegel. Unicase - an Ecosystem for Unified Software Engineering Research Tools. In *Proceedings on the 3rd International Conference on Global Software Engineering (ICGSE)*, August 2008.

[BCW12] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[BHH$^+$12] G. Bergmann, A. Hegedus, A. Horkváth, I. Ráth, Z. Ujhelyi, and D. Varró. Integrating efficient model queries into state-of-the-art EMF Tools. In *Proceedings on the 50th International Conference on Object, Models, Components and Patterns (TOOLS)*, pages 1–8. Springer-Verlag, June 2012.

[BK04] C. Bauer and G. King. *Hibernate in Action*. Manning Publications, 2004.

[BP08] C. Brun and A. Pierantonio. Model Differences in the Eclipse Modelling Framework. *UPGRADE*, 9(2):29–34, April 2008.

[BSO10] BSON. Binary JSON. http://www.bsonspec.org, 2010.

[BSRC10] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, September 2010.

[Cat91] R. Cattell. *Object Data Management: Database Systems for Engineering and Object-Oriented Applications.* Addison-Wesley Longman, 1991.

[Cat93] R. Cattell. The Engineering Database Benchmark. In *The Benchmark Handbook.* Morgan Kaufmann, 1993.

[Cat10] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, December 2010.

[CDG$^+$06] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings on the 7th conference on USENIX Symposium on operating systems design and implementation*, November 2006.

[CDO12] CDO. Connected Data Objects. http://www.eclipse.org/cdo, 2012.

[CDT12] C. Caue, C. Didonet, and M. Tisi. Transforming Very Large Models in the Cloud: A Research Roadmap. In *Cloud-MDE Workshop at the ECMFA 2012 Conference*, July 2012.

[CESW08] T. Clark, A. Evans, P. Sammut, and J. Williams. *Applied Metamodelling: A Foundation for Language Driven Development.* Ceteva, 2008.

[CGM10] J. Cánovas and J. García-Molina. An architecture-driven modernization tool for calculating metrics. *IEEE Software*, 27(4):37–43, July 2010.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal - Model-Driven Software Development*, 45(3):621–645, July 2006.

[Cha09] S. Chacon. *Pro Git.* APress, 2009.

[CJKC07] S. Cook, G. Jones, S. Kent, and A. Cameron. *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley, 2007.

[COR12] CORBA. Common Object Request Broker Architecture. Specification: http://www.omg.org/spec/CORBA/, 2012.

[Cou] CouchDB. http://couchdb.apache.org.

[CSFP04] B. Collins-Sussman, B. Fitzpatrick, and C. Pilato. *Version Control with Subversion.* O'Reilly Media, 2004.

[DG04] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. http://labs.google.com/papers/mapreduce-osdi04, 2004.

[DHJ+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings on the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220. ACM, October 2007.

[DPCGM13] O. Díaz, G. Puente, J. Cánovas, and J. García-Molina. Harvesting models from web 2.0 databases. *Software and Systems Modeling*, (1):15–34, February 2013.

[EB10] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[Eff06] S. Efftinge. oAW xText: A Framework for textual DSLs. In *Eclipse Summit Europe*, October 2006.

[Emf] Emfatic. http://www.eclipse.org/modeling/emft/?project=emfatic.

[EMF12] EMFQuery. http://www.eclipse.org/projects/project.php?id=modeling.emf.query, 2012.

[EPGM10] J. Espinazo-Pagán and J. García-Molina. A homogeneous repository for collaborative MDD. In *Proceedings on the 1st International Workshop on Model Comparison in Practice (ICWMP)*, pages 56–65. ACM, July 2010.

[EPMGM08] J. Espinazo-Pagán, M. Menárguez, and J. García-Molina. Metamodel Syntactic Sheets: An Approach for Defining Textual Concrete Syntaxes. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 185–199. Springer Berlin Heidelberg, 2008.

[Eps12] Epsilon. http://www.eclipse.org/epsilon/, 2012.

[EPSGM11] J. Espinazo-Pagán, J. Sánchez, and J. García-Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Proceedings on the 14th International Model Driven Engineering Languages and Systems (MoDELS) Conference*. Springer, October 2011.

[Fie00] R. Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[For82] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[Fow10] M. Fowler. *Domain-Specific Languages.* Addison-Wesley Professional, 2010.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GSKC04] J. Greenfield, K. Short, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools.* Wiley, 2004.

[GZL+04] J. Grey, J. Zheng, Y. Lin, S. Roychoudhury, H. Wu, R. Sudarsan, A. Gokhale, S. Neema, F. Shi, and T. Bapty. Model-Driven Program Transformation of a Large Avionics Network. In *Proceedings on the 3rd International Conference on Generative Programming and Component Engineering*, pages 361–378. Springer, 2004.

[HRW09] C. Hein, T. Ritter, and M. Wagner. Model-Driven Tool Integration with ModelBus. In *Proceedings on the 1st International Workshop on Future Trends on Model-Driven Development (FTMDD), in the context of the 11th International Conference on Enterprise Information Systems (ICEIS)*, pages 35–39. INSTICC Press, May 2009.

[Hun13] Bryan Hunt. Mongo EMF: https://github.com/BryanHunt/mongo-emf/, 2013.

[HWRK11] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480. ACM, May 2011.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: a model transformation tool. *Science of Computer Programming*, (1-2):31–39, June 2008.

[JS09] F. Jouault and J. Sottet. An Amma/ATL Solution for the Grabats 2009 Reverse Engineering Case Study. In *Grabats 2009 5th International Workshop on Graph-Based Tools*, July 2009.

[JSO] JSON. JavaScript Object Notation. http://www.json.org.

[KH10] M. Koegel and J. Helming. EMFStore: A model repository for EMF models. In *Proceedings on the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, volume 2, pages 307–308. ACM, 2010.

[Kle08] A. Kleppe. *Software Language Engineering.* Addison-Wesley, 2008.

[KPP08] D. Kolovos, R. Paige, and F. Polack. Scalability: The Holy Grail of Model-Driven Engineering. In *Proceedings on the Workshop on Challenges in MDE (MoDELS)*, pages 35–47. Springer, September 2008.

[KRM$^+$13] D. Kolovos, L. Rose, N. Matragkas, R. Paige, E. Guerra, J. Sánchez, J. de Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in Model Driven Engineering. In *Proceedings of the Workshop on Scalability un Model Driven Engineering*, pages 1–10. ACM, 2013.

[KT08] S. Kelly and J. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley, 2008.

[MDA03] MDA. Model-Driven Architecture. http://www.omg.org/mda/, 2003.

[MFM$^+$08] P. Monaheghehi, M. Fernández, J. Martell, M. Fritzsche, and W. Gilani. MDE Adoption in Industry: Challenges and Success Criteria. In *Proceedings on he Workshop on Challenges in MDE, 11th International Model Driven Engineering Languages and Systems (MoDELS) Conference*. Springer, September 2008.

[MHS05] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[MOF06] MOF. The Meta-Object Facility. Specification documents: http://www.omg.org/spec/MOF/2.0/, 2006.

[MOF08] MOF2Text. MOF Model to Text Transformation Language. Specification documents: http://www.omg.org/spec/MOFM2T/1.0/, 2008.

[Mor13] Morsa. http://www.modelum.es/morsa, 2013.

[OCL06a] OCL. The Object Constraint Language. Specification documents: http://www.omg.org/spec/OCL/2.0/, 2006.

[OCL06b] OCLInEcore. http://wiki.eclipse.org/MDT/OCLinEcore, 2006.

[OCL12] MDT OCL. http://www.eclipse.org/modeling/mdt/?project=ocl, 2012.

[OOM12] OOMEGA. Open Source Model-Driven Engineering Platform. http://www.oomega.net, 2012.

[OSG03] OSGI. *OSGI Service Platform.* IOS Press, 2003.

[QVT08] QVT. Query/View/Transformation. Specification documents: http://www.omg.org/spec/QVT/1.0/PDF/, 2008.

[SBP07] S. Sen, B. Baundry, and D. Precup. Partial Model Completion in Model-Driven Engineering using Constraint Logic Programming. In *Proceedings on the 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007)*, pages 59–69, October 2007.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional, 2008.

[SCGM07] J. Sánchez-Cuadrado and J. García-Molina. Building Domain-Specific Languages for Model-Driven Development. *IEEE Software*, 24(5):48–55, 2007.

[Sel08] B. Selic. Personal reflections on automation, programming culture and model-based software engineering. *Journal of Automated Software Engineering*, 15(3-4):379–391, December 2008.

[Sel12] B. Selic. What will it take? A view on adoption of model-based methods in practice. *Software and System Modeling*, 11(4):513–526, October 2012.

[SF12] P. Sadalage and M. Fowler. *NoSQL Distilled: A Brieg Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley, 2012.

[SGM09] J. Sánchez and J. García-Molina. A Model-Based Approach to Families of Embedded Domain-Specific Languages. *IEEE Transactions in Software Engineering*, 6(35):825–840, 2009.

[SM95] M. Stonebraker and M. Moore. *Object Relational DBMSs: The Next Great Wave.* Morgan Kauffmann, 1995.

[Sto10] M. Stonebraker. SQL Databases vs NoSQL Databases. *Communications of the ACM*, 53(4):10–11, April 2010.

[Str11] C. Strauch. NoSQL databases. Master's thesis, Stuttgart Media University, 2011.

[SZFK12] M. Scheidgen, A. Zubow, J. Fischer, and T. Kolbe. Automated and transparent Model Fragmentation for Persisting Large Models. In *Proceedings on the ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems MoDELS 2012*, volume 7590, pages 102–118. Springer, September 2012.

[TJF$^+$09] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bezivin. On the Use of Higher-Order Model Transformations. In *Proceedings on the 5th ECMDA-FA Conference*, pages 18–33. Springer, June 2009.

[UN10] W. Ulrich and P. Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Morgan Kaufmann, 2010.

[VP03] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is the Metamodeling of Mathematics). *Software and System Modeling*, 2(3):187–211, October 2003.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 2003.

[XMI11] XMI. The XML Metadata Interchange. Specification documents: http://www.omg.org/spec/XMI/, 2011.

[XPa99] XPath. The XML Path Language. Specification documents: http://www.w3.org/TR/xpath/, 1999.