Model Query Transformation Framework – MQT: from EMF-Based Model Query Languages to Persistence-Specific Query Languages

Xabier De Carlos Garcia

Supervisors:

Dr. Goiuria Sagardui Mendieta

Dr. Salvador Trujillo González

MONDRAGON
UNIBERTSITATEA

A thesis submitted to Mondragon Unibertsitatea
for the degree of Doctor of Philosophy

Department of Electronics and Computer Science
Mondragon Goi Eskola Politeknikoa
Mondragon Unibertsitatea
Arrasate, May 2016

Hamna shida.

Declaration

Hereby I declare, that this work is my original authorial work, which I have worked out by my own. All sources, figures and literature used during the elaboration of this work are properly cited and listed in complete reference to the due source.

Advisors: Dr. Goiuria Sagardui Mendieta, PhD and Dr. Salvador Trujillo González, PhD.

Abstract

Memory problems of XML Metadata Interchange (XMI) (default persistence in Eclipse Modelling Framework (EMF)) when operating large models, have motivated the appearance of alternative mechanisms for persistence of EMF models. Most recent approaches propose using database back-ends. These approaches provide support for querying models using EMF-based model query languages (Plain EMF, Object Constraint Language (OCL), EMF Query, Epsilon Object Language (EOL), etc.). However, these languages commonly require loading in-memory all the model elements that are involved in the query. In the case of queries that traverse models (most commonly used type of queries) they require to load entire model in-memory. This loading strategy causes memory problems when operated models are large.

Most database back-ends provide database-specific query languages that leverage capabilities of the database engine (better performance) and without requiring in-memory load of models for query execution (lower memory footprint). For example, Structured Query Language (SQL) is a query language for relational databases and Cypher is for Neo4J databases.

In this dissertation we present MQT-Engine, a framework that supports execution of model query languages but with the efficiency (in terms of memory and performance) of a database-specific query language. To achieve this, MQT-Engine provides a two-step query transformation mechanism: first, queries expressed with a model query language are transformed into a Query Language Independent Model (QLI Model); and then QLI Model is transformed into a database-specific query that is executed directly over the database. This mechanism provides extensibility and reusability to the framework, since it facilitates the inclusion of new query languages at both sides of the transformation.

A prototype of the framework is provided. It supports transformation of EOL queries into SQL queries that are executed directly over a relational Connected Data Objects (CDO) repository. The prototype has been evaluated with two experimental evaluations. First evaluation is based on the reverse engineering domain. It compares time and memory usage required by MQT-Engine and other query languages (EMF API, OCL and SQL) to execute a set of queries over models persisted with CDO. Second evaluation is based on the railway domain, and compares performance results of MQT-Engine and other query languages (EMF API, OCL, IncQuery, SQL, etc.) for executing a set of queries.

Obtained results show that MQT-Engine is able to execute successfully all the evaluated experiments. MQT-Engine is one of the evaluated solutions showing best performance results for first execution of model queries. In the case of query languages executed over CDO repositories, it is the faster solution and the one requiring less memory. For example, for the largest model in the reverse engineering case it is up to 162 times faster than a model query language executed at client-side, and it requires 23 times less memory. Additionally, the query transformation overload is constant and small (less than 2 seconds).

These results validate the main goal of this dissertation: to provide a framework that gives to the model engineers the ability for specifying queries in a model query language, and then execute them with a performance and memory footprint similar to that of a persistence-specific query language.

However, the framework has a set of limitations: the approach should be optimized when queries are subsequently executed; it only supports non-modification model traversal queries; and the prototype is specific for EOL queries over CDO repositories with DBStore. Therefore, it is planned to extend the framework and address these limitations in a future version.

Resumen

Los problemas de memoria de XMI (mecanismo de persistencia por defecto en EMF) cuando se trabaja con modelos grandes, han motivado la aparición de mecanismos de persistencia alternativos para los modelos EMF. Los enfoques más recientes proponen el uso de bases de datos para la persistencia de los modelos. La mayoría de estos enfoques soportan la ejecución de operaciones usando lenguajes de consulta de modelos basados en EMF (EMF API, OCL, EMF Query, EOL, etc.). Sin embargo, este tipo de lenguajes necesitan almacenar en memoria al menos todos los elementos implicados en la consulta (todos los elementos del modelo en las consultas que recorren completamente el modelo consultado). Esta estrategia de carga de la información para hacer las consultas provoca problemas de memoria cuando los modelos son de gran tamaño.

La mayoría de las bases de datos tienen lenguajes específicos que aprovechan las capacidades del motor de la base de datos (mayor rapidez) y sin la necesidad de cargar en memoria los modelos (menor uso de memoria). Por ejemplo, SQL es el lenguaje específico para las bases de datos relacionales y Cypher para las bases de datos Neo4J.

Este trabajo propone MQT-Engine, un framework que permite ejecutar lenguajes de consulta para modelos con tiempos de ejecución y uso de memoria similares al de un lenguaje específico de base de datos. MQT-Engine realiza una transformación en dos pasos de las consultas: primero transforma las consultas que han sido escritas con un lenguaje de consulta para modelos en un modelo que es independiente del lenguaje (QLI Model); después, el modelo generado se transforma en una consulta equivalente, pero escrita con un lenguaje específico de base de datos. La transformación en dos pasos proporciona extensibilidad y reusabilidad ya que facilita la inclusión de nuevos lenguajes.

Se ha implementado un prototipo de MQT-Engine que transforma consultas EOL en SQL y las ejecuta directamente sobre un repositorio CDO. El prototipo se ha evaluado con dos casos de uso. El primero está basado en el dominio de la ingeniería inversa. Se han comparado los tiempos de ejecución y el uso de memoria que necesitan MQT-Engine y otros lenguajes de consulta (EMF API, OCL y SQL) para ejecutar una serie de consultas sobre modelos persistidos en CDO. El segundo caso de uso está basado en el dominio de los ferrocarriles y compara los tiempos de ejecución que necesitan MQT-Engine y otros lenguajes (EMF API, OCL, IncQuery, etc.) para ejecutar varias consultas.

Los resultados obtenidos muestran que MQT-Engine es capaz de ejecutar correctamente todos los experimentos y además es una de las soluciones con mejores tiempos para la primera ejecución de las consultas de modelos. MQT-Engine es la opción más rápida y que necesita menos memoria entre los lenguajes ejecutados sobre repositorios CDO. Por ejemplo, en el caso del modelo más grande de ingeniería inversa, MQT-Engine es 162 veces más rápido y necesita 23 veces menos memoria que los lenguajes de consulta de modelos ejecutados al lado del cliente. Además, la sobrecarga de la transformación es pequeña y constante (menos de 2 segundos).

Estos resultados prueban el objetivo principal de esta tesis: proporcionar un framework que permite a los ingenieros de modelos definir las consultas con un lenguaje de consulta de modelos y además ejecutarlas con una con tiempos de ejecución y uso de memoria similares a los de un lenguaje específico de bases de datos.

Sin embargo, la solución tiene una serie de limitaciones: solo soporta consultas que recorren el modelo completamente y sin modificarlo; el prototipo es específico para consultas en EOL y sobre repositorios CDO (relacionales); y habría que optimizar la ejecución de las consultas cuando estas se ejecutan más de una vez. Se ha planeado resolver estas limitaciones en versiones futuras del trabajo.

Laburpena

Eredu handien gainean lan egitean XMI formatuak (persistentzia lehenetsia Eclipse Modelling Frameworkean) eragindako memoria arazoek, EMF ereduak iraunkor bihurtzeko mekanismo berrieak sortzea motibatu du. Planteamendu berrienek datu baseen erabilpena proposatzen dute eta EMFn oinarritutako ereduak kontsultatzeko lengoaiak erabiliz (EMF API, OCL, EMF Query, EOL, etab.), ereduen gainean lan egitea ahalbidetzen dute. Mota honetako lengoaiek eskatzen dute kontsultan inplikatuta dauden ereduko elementuak memorian kargartzea (eredua osorik zeharkatzen duten kontsulten kasuan elementu guztiak). Kargatzeko estrategia horrek memoria arazoak eragiten ditu ereduak tamaina handikoak direnean.

Datubase gehienek datubase motorraren gaitasunak aprobetxatzen dituzten eta ereduak memorian kargatu beharrik ez duten berariazko lengoaiak dituzte. Adibidez, SQL datubase erlazionalen berariazko lengoaia da eta Cypher Neo4J datubaseena.

Lan honetan MQT-Engine aurkezten da. Ereduak kontsultatzeko lengoaiak datubaseentzako berariazko lengoaia baten antzeko exekuzio denbora eta memoria erabilpenarekin exekutatzea ahalbidetzen du. MQT-Engine bi pausoko bihurketa bat burutzen du: lehenengo pausoan, ereduak kontsultatzeko lenguaia batean idatzitako kontsulta lengoaiarekiko independente den eredu bat bihurtzen du (QLI Model); ondoren, eredua datubasearen berariazko den lengoaia batean idatzitako kontsulta bihurtzen du. Bi pausoko bihurketak berrerabilpena eta hedagarritasuna eskaintzen ditu, eta lengoaia berriak sartzea errazten du.

MQT-Engine prototipo bat sortu da. Prototipo horrek EOL kontsultak SQL kontsulta bihurtzen ditu, eta ondoren CDO biltegi baten gainean exekutatzen ditu. Prototipoa bi erabilpen kasurekin ebaluatu da. Lehenengoa, alderantzizko ingeniaritzan dago oinarrituta eta MQT-Enginek eta beste kontsulta lengoaiek (EMF API, OCL, eta SQL) hainbat kontsulta CDO ereduen gainean exekutatzeko behar duten denbora eta memoria neurtzen ditu. Bigarren erabilpen kasua, trenbideen domeinuan dago oinarrituta eta MQT-Enginek eta beste lengoaiek

(EMF API, OCL, IncQuery, etab.) hainbat kontsulta exekutatzeko behar duten denbora konparatzeko erabili da.

Jasotako emaitzek erakusten dutenez, MQT-Engine esperimentu guztiak behar bezala exekutatzeko gai da. Ereduen gainean lehen kontsulta egitean denboraren aldetik emaitza onenetarikoak erakutsi ditu eta CDO ereduen gainean exekutatutako lengoaietatik aukera onena da (denboraren eta memoriaren aldetik). Adibidez, alderantzizko ingeniaritza kasuko eredu handienaren gainean kontsulta exekutatzean, MQT-Engine bezeroaren-aldean exekutatzen diren lengoaiak baino 162 aldiz azkarragoa izan daiteke, eta 23 aldiz txikiagoa den memoria kopurua erabiliz. Gainera, bihurketaren gainkarga txikia eta etengabekoa da (2 segundu baino gutxiago).

Emaitza hauek tesi honen helburu nagusia frogatzen dute: eredu ingeniariei, kontsultak, ereduak kontsultatzeko lengoia batekin idazteko, eta persistentziaren berariazko den lengoaia baten antzeko exekuzio denbora eta erabilitako memoria balioak erakusten dituen soluzio bat sortzea.

Hala ere, MQT-Enginek hainbat muga ditu: aldaketarik engin gabe, eredua osorik zeharkatzen duten kontsultak bihurtzeko gai da soilik; prototipoa EOL kontsultentzako dago diseinatua eta bihurtutako SQL kontsulta CDO biltegi erlazionalen gainean exekutatzeko dira; kontsulta berdina hainbat aldiz exekutatzen denean beste soluzio batzuk baino emaitza txarragoak ematen ditu. Etorkizunean, MQT-Engineren bertsio berrietan muga horiek jorratzeko asmoa dago.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Goiuria Sagardui for the continuous support and guidance of my Ph.D study and for her patience and motivation.

IK4-Ikerlan, and specially Dr. Salvador Trujillo for offering me the opportunity to complete the thesis. All co-workers who have provided excellent feedback and help for this thesis, and I would particularly like to thank Xabier Mendialdua and Aitor Murguzur.

My sincere thanks also goes to Dr. Dimitris Kolovos for making possible my stay at University of York. And I do not wish to leave out the rest of the team at the Enterprise Systems group.

Nire esker beroenak ere abentura honetan modu batean edo bestean lagundu izan duzuenoi: Aizea, Alain, Angel, Asier, Arturo, Arantxa, Beñat, Cristina, Cristobal, Josune, Koper, Koldo, Mikel, Santi, Oskar eta gainerako guztioi.

Eta amaitzeko, esker on berezia nire familiako kide bakoitzari: Iraia, Aner, Aita eta Ama, Julen, Gontzal, Kepa eta Ainara, Margari eta Pedro, Aitite eta Amama. Zuek eskua luzatu bait didazue lurrera erori naizen bakoitzean.

Bihotz-bihotzez, ESKERRIK ASKO!

Contents

List of Figures

List of Tables

List of Code Listings

Acronyms

API Application Programming Interface. 3, 17, 23, 31, 33–35, 43, 119

AST Abstract Syntax Tree. 41

CDO Connected Data Objects. vii–x, 2–5, 9, 18, 22, 23, 25, 26, 32, 44, 68, 75, 79, 91–93, 101, 104–107, 113, 115, 118–121, 123, 124, 127, 129, 133, 135–138, 142–145, 147, 149

CSP Constraint Satisfaction Problem. 37

DBMS Database Management System. 38

DSML Domain Specific Modelling Language. 14

EMF Eclipse Modelling Framework. vii, ix, xi, 1, 2, 8, 13, 17–25, 29, 31–33, 35, 36, 38–41, 43, 44, 53, 104, 113, 115, 119, 142, 145, 146

EOL Epsilon Object Language. vii–xii, 2, 4, 5, 9, 24, 32, 33, 35, 39, 41, 43, 44, 65–68, 75, 79–84, 86, 88–90, 101, 105, 107, 110, 113, 115, 119, 123, 137, 142, 143, 145, 146, 167, 173–176

EOS Eye OCL Software. 39

GPML General Purpose Modelling Language. 14

HQL Hibernate Query Language. 22, 34, 43

IDE Integrated Development Environment. 1

JVM Java Virtual Machine. 24

M2M Model to Model. 15, 16

M2T Model to Text. 16

MDD Model Driven Development. 1, 8, 13, 14, 16, 17, 51

OCL Object Constraint Language. vii–xii, 2, 3, 23, 31–33, 35–41, 43, 44, 75, 104, 112, 113, 115, 119, 143, 145, 166, 167

OMG Object Management Group. 2, 31

OSGi Open Services Gateway initiative. 19

SQL Structured Query Language. vii–xii, 3–5, 22, 23, 33, 34, 37–41, 43, 44, 68–71, 76, 79, 91, 92, 94, 96, 98, 111, 113, 142–148

UML Unified Modelling Language. 1, 14, 17, 31, 32, 36–41, 44, 145

URI Uniform Resource Identifier. 19, 20

XMI XML Metadata Interchange. vii, ix, xi, 2, 18, 23, 24, 26, 37, 43, 105, 116, 119, 121, 126, 129, 137, 138, 147, 149

1
Introduction

Automatizing and optimizing development processes is crucial to reduce development efforts and time to market of industrial projects. Model Driven Development (MDD) promises improvements in the development process through an intensive use of abstractions that are specified within models. Using models, developers are able to focus in the domain-specific problems rather than on problems derived from the technical implementation. Models are considered first class entities during the development process. They are operated using modelling tools, which perform different types of tasks over them (edition, validation, simulation, execution, transformation or code generation). Among all the activities, model queries are intensively used. Therefore, the impact of query performance on tool performance and user experience is significant [Ber10].

EMF is a modelling framework that is widely used by the Academia. Moreover, it is one of the leading industrial modelling ecosystems [Ujh15]. EMF is part of the Eclipse Integrated Development Environment (IDE) and there is a large collection of EMF-based approaches that provide support for creating modelling tools to operate and edit EMF models. EMF provides *Ecore* format, which is a subset of Unified Modelling Language (UML), and it is used to define domain-specific metamodels where all domain abstractions are specified. *Ecore* metamodels are conformed by EMF models where domain-specific products are specified.

EMF models are natively persisted using XMI format standardised by the Object Management Group (OMG). One of the main characteristics of this format is that before operating a model, required information has to be loaded first from the physical XMI file into the memory. Similarly, if model content is modified, all the information that has been loaded in-memory (including modified parts) has to be transferred from the memory into the physical XMI file. Different studies have shown that this strategy to load and save operated models when they are persisted in XMI files entails memory problems [EP13b, Góm15b].

To overcome memory problems of XMI, recent approaches have proposed the use of database back-ends for persistence of EMF models. This way, the persistence mechanism leverage capabilities of the database back-end: partial load of the information, load on demand, advanced caching mechanisms, and incremental storage of the information. Each proposed approach uses a different back-end strategy for persistence of models: Morsa [EP13b], MongoEMF [Hun14], NeoEMF/Graph [Ben14] and EMF Fragments [Sch13] provide persistence using NoSQL database back-ends; Teneo [Ten12] persists models in relational databases using Hibernate; and CDO [Cdo16] provides support for persisting models in several kinds of databases (in-memory databases, relational databases and NoSQL databases). Persistence approaches have been compared in different studies [EP13b, Sch12, Ben14, Góm15b] and results of the comparison have shown that each persistence approach is more appropriate for a specific modelling scenario.

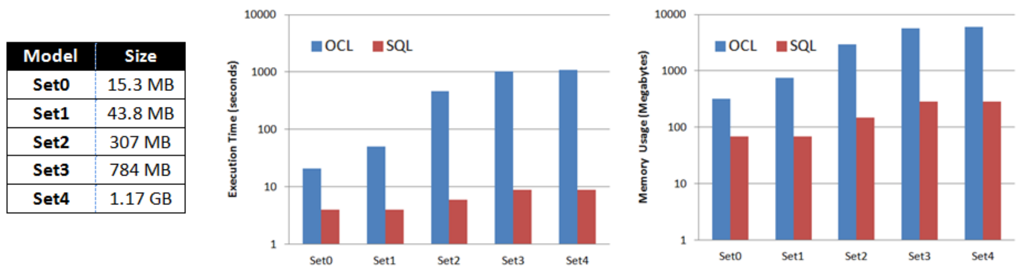
1.1 Motivation

Among all the activities, model queries are intensively used by modelling tools and also by model engineers. Consequently, proper operation and performance of queries is essential. All the database-based persistence approaches for EMF models support executing queries using EMF-based model query languages (e.g. EMF Query, OCL, EOL, etc.). Model query languages are focused on interacting with models using domain-specific abstractions. Thus, they are close to the domain and to the knowledge of model engineers. Moreover, model query languages are persistence-agnostic, and model engineers do not require to know how the information is persisted.

Model query languages are commonly executed at client-side and require loading in-memory at least all the model elements that have to be operated or queried. In practise, queries that traverse the entire model are the most commonly used type of queries [Góm15b]. These queries obtain all the instances of a specific type and require traversing the entire model. Therefore, most of the model query languages require to load in-memory the entire model to be queried.

In-memory load of the information could be avoided using database-specific query languages, since they leverage database capabilities such as load on-demand and partial load of the information. Most of the database-back ends provide database-specific languages (e.g. SQL in relational databases, Cypher in Neo4J) or Application Programming Interfaces (APIs) (e.g. MongoDB Core API) to operate information persisted within the database. Unlike model query languages, persistence-specific query languages are dependent on the persistence, and they are commonly executed over the database at server-side. This way, the query result is obtained directly from the database, and they do not require to load intermediate results in-memory.

Figure 1.1 illustrates a performed preliminary evaluation where a model query language (OCL) and a persistence-specific query language (SQL) were used to query models persisted in a relational database and using CDO. As results show, the persistence-specific query language requires less time and memory than the model query language. The difference between results increases as the size of the queried model increases.



Model	Size
Set0	15.3 MB
Set1	43.8 MB
Set2	307 MB
Set3	784 MB
Set4	1.17 GB

Figure 1.1: Preliminary evaluation of query languages.

However, persistence-specific languages require engineers to be aware of the way the information is persisted and also to learn persistence-specific concepts and languages. These facts increase programming effort to get complex queries correct, and make difficult for model engineers the adoption of persistence-specific query languages.

Differences between model query languages and persistence-specific query languages motivate this dissertation. Thus, the main goal is the development of a solution that is able to execute queries expressed with a model query language, but with the efficiency (in terms of execution time and memory usage) of a persistence-specific query language. To validate the feasibility of this goal, a preliminary prototype was implemented [Car15a, Car16a]. This prototype transforms EOL queries into SQL queries that are executed over an ad-hoc persistence. Ad-hoc persistence is based on a relational database with a metamodel-agnostic schema [Car14a].

1.2 Contribution

This dissertation presents MQT-Engine framework, an approach for (a) transformation of queries from a model query language into a persistence-specific query language; and (b) execution of the generated query over the persistence and at server-side. This way, MQT-Engine aims to provide performance and memory usage optimizations when models are operated by engineers using a model query language.

The framework performs query transformation and execution in three phases. First phase is the query transformation, and it is executed in two-steps: (1) the query expressed in a model query language is transformed into a query language independent model; and (2) the query language independent model is transformed into a query expressed with a language specific of the target persistence. The generated query is executed directly over the database in the second phase. Last phase processes results that are obtained from the persistence and formats them in the data-types expected by the query in the model query language.

The two-step transformation of queries provided by the framework separates concepts of model query languages and persistence-specific query languages. This strategy aims to facilitate the inclusion of new query languages in MQT-Engine framework.

This dissertation provides a prototype of the framework. The prototype supports transforming queries expressed with EOL into SQL queries that are executed over a CDO repository with a relational database for persistence of models.

MQT-Engine prototype is evaluated using two benchmark cases. The first experimental evaluation is based on a reverse engineering benchmark case [Sot09]. This benchmark case proposes a set of models that specify source code of Java projects with the aim of comparing execution time and memory usage metrics required by approaches to execute a complex query over them.

The complex query extracts all the singleton classes that are specified within the models. The second experimental evaluation is based on the Train Benchmark Case [Szá15]. This benchmark case uses models and queries that are close to a real industrial domain (railway domain).

MQT-Engine results have been compared with the performance results of executing the queries using other persistences and query languages. Results show that MQT-Engine is able to execute queries expressed with a model query language (EOL), but with performance and memory usage similar to a persistence-specific query language (SQL). In most of the experiments MQT-Engine requires less time and memory than other query languages executed over the evaluated CDO models. Moreover, experiments show that using MQT-Engine for querying models, CDO is one of the persistences showing best results.

1.2.1 Technical Contribution

This dissertation provides the following technical contributions:

- Design and implementation of MQT-Engine framework.
- Design and implementation of a query language independent metamodel used to increase abstraction of queries during query transformation process.
- Implementation of the MQT-Engine Framework prototype for transforming EOL queries into SQL, and executing them over relational CDO repositories.
- Experimental evaluation of the prototype using two benchmark cases: reverse engineering case and train benchmark case.

1.2.2 Publications

The presented dissertation lead to the following publications:

Book Chapters

- Xabier De Carlos, Goiuria Sagardui, Aitor Murguzur, Salvador Trujillo, and Xabier Mendialdua. Runtime translation of model-level queries to persistence-level. In *Model-Driven Engineering and Software Development - Third International Conference, MODELSWARD 2015, Angers, France, February 9-11, 2015, Revised Selected Papers*, pp. 97–111. 2015

International Conferences

- Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. Two-Step Transformation of Model Traversal EOL Queries for Large CDO Repositories. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pp. 141–157. 2016

- Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. CRUD Model Operations from EOL to SQL. In *MODELSWARD 2016 - Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016.* 2016

- Xabier De Carlos, Goiuria Sagardui, Aitor Murguzur, Salvador Trujillo, and Xabier Mendialdua. Model Query Translator - A Model-level Query Approach for Large-scale Models. In *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, 9-11 February, 2015.*, pp. 62–73. 2015

International Workshops

- Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. MQT, an Approach for Run-Time Query Translation: From EOL to SQL. In *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, pp. 13–22. 2014

- Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. Scalable Model Edition, Query and Version Control Through Embedded Database Persistence. In *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 28 - October 3, 2014.*, pp. 11–15. 2014

Other Publications

- Alessandra Bagnato, Etienne Brosse, Andrey Sadovykh, Pedro Maló, Salvador Trujillo, Xabier Mendialdua, and Xabier De Carlos. Flexible and Scalable Modelling in the MONDO Project: Industrial Case Studies. In *Proceedings of the 3rd Workshop on Extreme Modeling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, XM@MoDELS 2014, Valencia, Spain, September 29, 2014.*, pp. 42–51. 2014

- Aitor Murguzur, Xabier De Carlos, Salvador Trujillo, and Goiuria Sagardui. On the Support of Multi-perspective Process Models Variability for Smart Environments. In *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pp. 549–554. 2014

- Aitor Murguzur, Xabier De Carlos, Salvador Trujillo, and Goiuria Sagardui. Context-Aware Staged Configuration of Process Variants@Runtime. In *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*, pp. 241–255. 2014

- Csaba Debreceni, István Ráth, Dániel Varró, Xabier De Carlos, Xabier Mendialdua, and Salvador Trujillo. Automated model merge by design space exploration. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pp. 104–121. 2016

1.2.3 Awards

- Best Poster Award at MoDELS 2014: *Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. Scalable Model Edition, Query and Version Control Through Embedded Database Persistence. In* Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 28 - October 3, 2014.*, pp. 11–15. 2014*

1.2.4 Research Visits

A research stay has been made during the first year of the PhD program, and at the *Enterprise Systems* group of the *Department of Computer Science* in the *University of York*. The visit was supervised by *Dr. Dimitris Kolovos*. During this visit, an analysis of the state of the art and a preliminary implementation was performed.

This stay was completed with a second short-visit in the second year of the PhD program.

1.3 Support

This thesis has been supported by:

- IK4-Ikerlan
- Research Grant from *Fundación Centros Tecnológicos Iñaki Goenaga*

1.4 Outline

This dissertation is structured as follows:

Chapter 2 introduces **context** of this dissertation providing some background about MDD. First, modelling scenarios and artifacts are described. Then different mechanisms for persisting models that have been proposed for EMF-based MDD scenarios are described and compared.

Chapter 3 describes languages that can be used to query EMF models. Each language is described, classified and compared with the rest of the languages. This chapter also includes description of approaches for query language transformation. The chapter ends with a critical analysis of the existing solutions that motivated the work performed in this dissertation.

Chapter 4 presents the theoretical framework followed by this work, which includes: hypotheses to be validated by the performed work, goals to be achieved, operative goals to be implemented by the solution, and case studies to evaluate the implemented solution.

Chapter 5 provides description of foundations of the proposed framework. The framework transforms queries from a model query language into a persistence-specific query language, and then executes it. In this chapter different aspects of the framework are analysed: involved roles, architecture of the framework and execution process. It also provides description of a query language independent metamodel that is used to perform query transformation.

Chapter 6 introduces implementations for the framework that support: (i) transforming and executing EOL queries into SQL; and (ii) executing generated queries over relational CDO repositories where models are persisted.

Chapter 7 presents an experimental evaluation of the proposed framework that is based on a reverse engineering case study. Performance and memory usage required by the framework for querying models are obtained and compared with other approaches for querying models.

Chapter 8 presents another experimental evaluation that is based on the Train Benchmark Case. This benchmark case uses models and queries that are close to a real industrial domain (railway domain).

Chapter 9 finalizes this document providing some conclusions about the performed work. It also provides a set of tasks that could be performed in future works.

Part I

State of the Art

2 Background and Context

This chapter provides some background about MDD, which helps in the comprehension of the PhD work. The chapter is structured as follows: first, a general overview of MDD is provided, analysing different artifacts of this paradigm; next, mechanisms to persisting models in EMF-based MDD scenarios are described.

2.1 Model Driven Development

MDD is a paradigm that raises abstraction-level of software development processes. It is a continuation of the software development trend where developers aim to specify what should be done rather than how should be done [Atk03]. MDD allows modelling required functionalities and specifying the system architecture instead of implementing all the details of a system using a programming language.

MDD is a development process driven by one or several models where software implementation is automatically generated from models. Developers (modelling engineers) specify models with concepts that are closer to the target problem domain rather than to the commonly used programming languages [Sel03]. This paradigm promises a number of benefits including: productivity increase, development time reduction and increase quality of the software.

2.1.1 Modelling Artifacts

Modelling artifacts are the elements involved in MDD processes: models, metamodels, transformations, modelling tools and queries. Modelling scenarios composed by these elements are also referred as meta-modelling ecosystems [DR12] (see Figure 2.1).

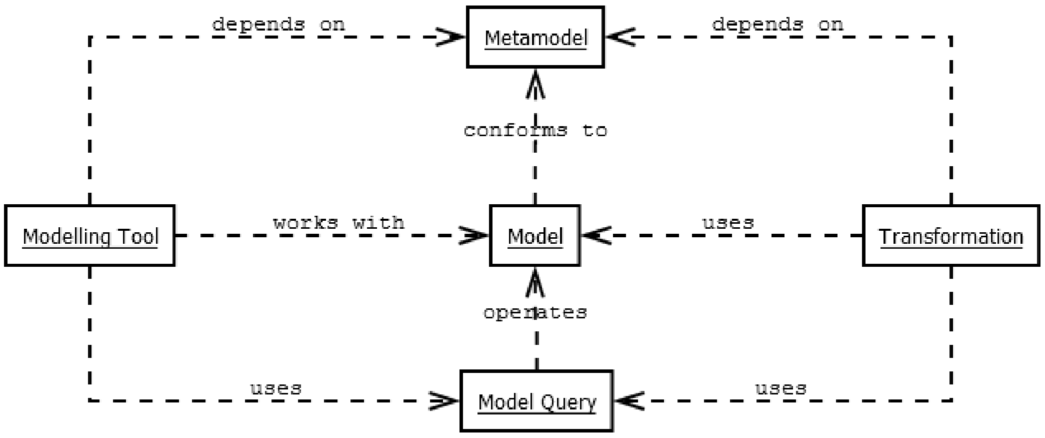
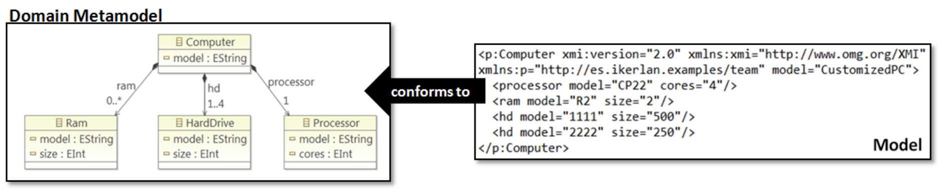


Figure 2.1: Meta-modelling ecosystem.

Models and Metamodels

Models are the way for representing reality using abstractions [Béz05] and most typical uses include: code generation, design and specification, validation, simulation, etc. Models are specified using modelling languages of different types (e.g. textual modelling languages, graphical modelling languages, programming abstractions, etc.). Modelling languages can be classified into two groups: General Purpose Modelling Languages (GPMLs) and Domain Specific Modelling Languages (DSMLs). GPMLs provide a generic modelling language that can be used in any domain (e.g. UML). DSMLs are modelling languages that are created to resolve needs and problems of a specific domain.

Models are the instances of one or various metamodels. Metamodels specify the domain (elements, relations and constraints) and the modelling language of the domain. If a model is specified using the modelling language defined within a



```xml
<p:Computer xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:p="http://es.ikerlan.examples/team" model="CustomizedPC">
  <processor model="CP22" cores="4"/>
  <ram model="R2" size="2"/>
  <hd model="1111" size="500"/>
  <hd model="2222" size="250"/>
</p:Computer>
```

Figure 2.2: Sample metamodel and model.

metamodel, the model is an *instance of* the metamodel. In this case the model *conforms to* the metamodel.

Figure 2.2 depicts a sample model where a computer is specified using some abstractions. These abstractions represent physical parts of a computer like memory, processor or hard disc. The abstractions are specified in the domain metamodel conformed by the model.

Transformations

Transformations are used to generate text, code or another models from models. These are defined using transformation definitions where transformation rules are implemented. Transformation rules specify how each element and relation of the domain must be converted. As transformations use domain abstractions specified in the metamodel, they are dependent on and related to the domain metamodel. There are two types of model transformations:

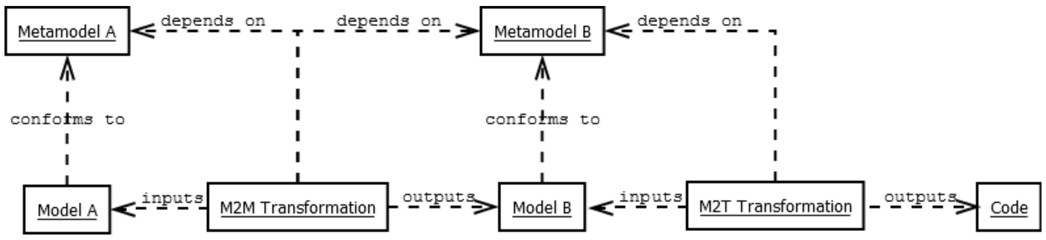


Figure 2.3: Model transformations.

- **Model to Model (M2M) transformations** transform a model of a specific domain into a model that conforms to other domain. They are commonly used when the abstractions within a model must be adapted to conform abstractions of a different domain metamodel. Figure 2.3 illustrates a M2M transformation where the *Model A* that conforms to *Metamodel A* is transformed into *Model B* that conforms to *Metamodel B*.

- **Model to Text (M2T) transformations** generate code or text from the input models. They are used to perform several types of tasks which include generation of textual documentation for an specific system or project or generation of source-code that is executed in a system hardware. Figure 2.3 illustrates a M2T transformation where *Code* is generated from *Model B* that conforms to the *Metamodel B*.

Modelling Tools

Modelling tools are the environment for working with models. As these tools are adapted to the domain and work with models that conform to a domain metamodel, they depend on a domain metamodel. Modelling tools allow stakeholders operating models. For example, validate the system or execute simulations using the model. Modelling tool types include: graphical editors that are used to work with models graphically; textual editors, where models are specified using a textual language; hierarchical editors, where models are specified in tree view; validation views, where errors and warnings of the models are shown; and transformation tools, containing transformations and allowing to perform model transformations.

Model Queries

Operating models encompasses tasks such as model validation, constraint checking of the specified models, generate code or models, etc. Therefore, these tasks require executing model queries for getting information from them and also for modifying them.

2.1.2 Involved Roles

Stakeholders of different roles are involved in MDD:

- **Domain Expert** is a stakeholder specialized in the domain [Völ09]. Domain expert must know about existing domain elements and relations between them, modifications performed over the domain, relevant concepts, etc.

- **Model Engineer** (a.k.a Domain User) is the stakeholder that is not as specialized in the domain as the domain expert, but is able to work with it [Völ09]. Model engineer is responsible to operate models using modelling tools, transformations and queries.

- **Domain Specific Modelling Tool Developer** is the stakeholder specialized in MDD. It must have different skills: expert in metamodel specification and expert in implementing modelling tools.

2.1.3 Eclipse Modelling Framework

Eclipse Modelling Framework (EMF) is a widely used modelling framework that is part of the Eclipse IDE. It is one of the leading industrial modelling ecosystems [Ujh15]. Domain metamodels in EMF are commonly specified using the *Ecore* format, which is a subset of UML class diagrams. EMF provides support to generate a domain-specific Java API (a.k.a. EMF API) which is able to interact with EMF models. Each EMF model specifies a domain-specific product and it conforms to a *Ecore* metamodel which specifies the domain.

There is a large collection of technologies that provide support for creating modelling tools that are EMF-based, and are able to operate EMF models. Table 2.1 illustrates some of the existing technologies. They are grouped by the type of modelling tool provided by each solution.

Table 2.1: Some of the EMF-Based technologies.

EMF-Based Technologies				
Edition	**Transformation**	**Query**	**Version Control**	**Comparison**
GMF	ATL	OCL	ModelCVS	EMF Compare
Sirius	Mofscript	IncQuery	AMOR	EMF Diff/Merge
Graphiti	ETL	EMF Query	EMFStore	DSE Merge
EMF Parsley	EGL	EMF Query 2	CDO	
EMF Forms	Acceleo	EOL		

2.2 Model Persistence

By default, EMF persists models using the XMI persistence format. One of the main features of XMI is that before operating or querying a model, it requires to load in-memory all the information that is involved. Once the information is in memory, it is operated in different ways: editing models, querying models, generating code, executing model transformations, etc. If the model has been modified, information in memory has to be stored again in the XMI file. Different studies have shown that XMI entails memory problems for large models [EP13b, Góm15b].

In order to overcome memory problems of XMI, recent approaches have proposed alternative persistence mechanisms for EMF models that are based on databases. This way, the persistence mechanism leverage database capabilities (caching mechanisms, partial load, load on demand, etc.) when operating and querying models. Each approach proposes a different back-end strategy for persistence: noSQL databases in the case of Morsa, MongoDB, NeoEMF/Map, NeoEMF/Graph and EMF Fragments; relational databases in Teneo; or several database back-ends in CDO. In addition to the database-based persistence, there are approaches that have opt for other types of mechanisms for persistence: file-based persistence with fragmentation mechanisms (e.g. EMF Splitter, EMF Fragments) or file-based binary persistence (e.g. binary persistence provided by EMF).

2.2.1 Classification Factors for Persistences

Different classification factors have been used to analyse and compare persistence approaches. They are described in Table 2.2.

2.2.2 Persistence Approaches

Morsa [EP13b, EP13a] provides support for persistence of large models using MongoDB, a document-based NoSQL database. Morsa follows two main design goals: provide transparent integration with the existing modelling tools; and provide scalability when operating models. Morsa uses client/server architecture, and client-side provides transparent integration with EMF-based modelling tools. Morsa supports persisting in a MongoDB [Mon16] database both dynamic and generated EMF models without requiring to generate code.

Table 2.2: Description of factors used to classify persistences.

Factor	Description	Value
Persistence type	Type of persistence used by the approach: DB if a database is used to persist the information; or File if the information is persisted in physical files.	*DB / File*
Database type	Type of the used database back-end. This factor is only applicable for approaches that use databases.	*Relational / NoSQL*
Back-end	Name of the concrete back-end that is used by the approach to persist models.	Back-end name
Requires server	The approach requires a server.	*Check mark*
Requires code generation	The approach requires generating domain-specific code (using metamodel).	*Check mark*
Persistence Load	The approach explicitly provides improved model load that is persistence-specific.	*Check mark*
Persistence Caching	The approach explicitly provides improved model caching that is persistence-specific	*Check mark*
Persistence Save	The approach explicitly provides improved storage of models that is persistence-specific.	*Check mark*
Versioning Support	The approach provides model versioning.	*Check mark*
Web Visualization	The approach supports web-based visualization of models and model elements.	*Check mark*
Multiple Models	The approach supports to persist more than one model in the same file or database.	*Check mark*
Persistence Specific Query	Persistence-specific query languages supported by the approach.	Language names
Evaluation	There is evidence in the literature that the approach has been evaluated.	*Check mark*
Compared With	Other approaches compared with the approach.	Approach names

Morsa provides different mechanisms for model load: full load, which completely loads model in-memory before working with it; and load on-demand, which only loads required model elements and decreases memory footprint. This approach provides different cache replacement policies that can be selected by the user, and two mechanisms for storing and modifying models (full store and incremental store). Morsa supports model queries in different ways: using the Morsa-specific implementation of the EMF Resource, queries can be expressed using the EMF API; and using MorsaQL [EP14], a Morsa-specific query language.

MongoEMF [Hun14] provides a framework for persistence that is based on MongoDB [Mon16]. The framework is based on Open Services Gateway initiative (OSGi) services and is extensible and flexible. This approach uses client-server architecture. Client-side uses a MongoEMF-specific implementation of EMF and it is integrated with EMF-based modelling tools. Server-side uses MongoDB database back-end for storing physically models.

Models and model elements are loaded using Uniform Resource Identifiers

(URIs). Each URI contains information about server *hostname*, server *port*, *database* name, database *collection* name and model element *id*. MongoEMF provides a set of options for configuring strategies for model loading and saving [Hun14]. However, it has not be found any information about MongoEMF-specific caching mechanism. This approach provides two ways for querying models: with the EMF API, but using MongoEMF URIs for identifying model elements; and with MongoDB native queries.

NeoEMF/Graph (a.k.a. Neo4EMF) [Ben14, Atl15] provides model persistence using Neo4J [Neo16] NoSQL database. NeoEMF/Graph supports mapping between EMF models and graph databases and the database connection is performed via Blueprints. NeoEMF/Graph uses client/server architecture. Client-side is integrated with EMF-based modelling tools using an implementation of the EMF Resource which is able to access models that are persisted in the server. NeoEMF/Graph provides two ways to obtain EMF Resource implementation: a metamodel-agnostic implementation that uses a dynamic API and a metamodel-specific generated implementation (the code generator is provided by NeoEMF/Graph). In both options, Resource communicates with the *server-side* where the Neo4J database, which persists models as graphs, is located. NeoEMF/Graph supports embedded execution of the server.

NeoEMF/Graph provides load on-demand of model elements that reduces memory footprint, and it is able to load and query large models with limited memory. If generated Java API is used, NeoEMF/Graph separates the objects data from the objects and this provides a lightweight first-time load of each model element. NeoEMF/Graph only loads the model elements that are required by the query or operation. Therefore, before loading a model element, it searches in the cache, and if it does not exists, the element is obtained directly from the database. Moreover, the approach provides lightweight persistence of model changes which only saves modified parts. NeoEMF/Graph supports execution of queries using the EMF API. Persisted information could be queried with the Neo4J-specific query languages (cypher and Neo4J Core API). However, there has not been found support for persistence-specific queries integrated with the EMF API.

NeoEMF/Map (a.k.a. Kyanos) [Góm15b] provides model persistence using MapDB. MapDB [Map16] is an embedded database engine for Java that provides memory allocators, caches, storages, indexes or serializers. NeoEMF/Map provides transparent integration with EMF-based modelling tools through an specific implementation of the EMF Resource. This implementation uses three different maps to access and persist model information: a *property map* that stores all information of model elements; a *type map* that specifies type for each model element; and *containment map* that stores containment relationships. Each model element will have a unique identifier and it is used to obtain the information from the different maps.

NeoEMF/Map provides a lightweight mechanism to load information on-demand [Góm15b, p. 9]. Moreover, the approach allows the garbage collector to deallocate model elements that are not directly referenced by the performed model operation. It has not be found any description about the mechanisms used by NeoEMF/Map for storing models. Persisted models can be queried using the native EMF API, but NeoEMF/Map does not support execution of queries using a database-specific query language.

NeoEMF/HBase [Góm15a] provides EMF models persistence using the Apache HBase data store. It provides decentralized model persistence, and consequently, models can be accessed in a distributed and concurrent way. The client provides a specific implementation of the EMF Resource which is integrated with EMF-based modelling tools. The server-side contains HBase database (combined with other Apache technologies) where models are persisted.

NeoEMF/HBase provides lightweight on-demand loading. This mechanism is based on a delegate object that is responsible for tracking model elements and loading them directly from the persistence back-end when they have not been previously loaded in-memory and are required. The approach uses a caching strategy provided by NeoEMF[1]. Moreover, it provides lightweight on-demand saving, which also makes use of the delegate objects. All model changes are automatically reflected in the underlying storage, making changes visible to any client [Góm15a, p. 6]. NeoEMF/HBase supports querying models using the native EMF API.

Teneo [Ten12] provides model persistence using relational databases. It uses Hibernate and supports mapping between EMF objects and relational databases. Teneo uses a client-server architecture, where client-side implements the Resource of EMF and is integrated with EMF-based tools. Client-side communicates with the server-side that contains a relational database. The interaction is performed

[1]More details at: `http://www.neoemf.com`

via Hibernate that is also responsible for mapping model elements and relational databases. Teneo supports both metamodel-independent and metamodel-specific schemas to persist model information and provides support for customizing the database schemas to be used. Customizations are based on annotations added on the domain metamodel.

Teneo provides a huge quantity of configuration options that support specifying behaviour to load models[2]. Moreover, collections are loaded lazily only when they are first accessed. If the information is too large to be loaded in-memory, it also provides the extra-lazy functionality provided by Hibernate. Teneo uses the cache and storage mechanisms provided by Hibernate. Moreover, the approach provides three different ways for querying models: (1) using the native EMF API; (2) using Hibernate Query Language (HQL), a Hibernate-specific query language; and (3) using SQL queries over relational databases.

CDO [Cdo16] is an approach that provides transparent persistence of EMF models using all kind of databases. CDO uses client/server architecture. *Client-side* natively supports basic features with dynamic and legacy models. However, take advantage of all functions of CDO requires to generate CDO-aware metamodel implementation. Client-side is able to work off-line over a repository, cloning the repository in the local machine of the user. Then cloned repositories are synchronized at background with the repositories at server-side when they are online. Server-side provides a repository where all the information such as models, metamodels or history are persisted together. It supports other features such as: definition of authentication options; multi-user access to the models; or model versioning and branching. CDO supports the embedded execution of the server at client-side. The server-side supports model persistence in several database kinds: in-memory databases, relational databases, relational databases with hibernate, Objectivity/DB, MongoDB and DB4O. Database back-ends are supported by different IStore implementations of CDO, and DBStore which supports relational databases is the most mature. DBStore provides support for all the CDO features and in practice mainly relational back-ends are used with CDO [Ben14].

CDO provides load on-demand mechanisms that support obtaining only the required information from the database back-end, and it entails a reduction in the memory usage when operating models. CDO provides caches in three different places: two are at client-side, and other one at server-side. Additionally, CDO is able to use the storage mechanisms provided by the database back-end that

[2]Extended information about configuration options at `https://goo.gl/Hv42bc`

is used in each repository for persistence of models. CDO repositories provide support for querying models using the native EMF API. However, each store supports server-side execution of alternative query languages. In the case of DBStore, it provides server-side execution of OCL and SQL queries.

Binary persistence. EMF provides a persistence mechanism alternative for XMI that persist models in a binary format. It is supported through an alternative implementation of the EMF Resource.

The loading mechanism is the same that is used with XMI, the default persistence in EMF. The main difference is that the binary persistence requires less time to load information since it sacrifices readability in exchange for compactness. If the model has to be traversed, entire model is loaded in-memory. Model caching and storing mechanisms are the same that are used with XMI. Binary models are queried using the native EMF API. Persistence-specific query languages are not supported.

EMFJson [Hil15] provides persistence of models using the JSON format. EMFJson format is simple and customizable and preserves the features of XMI. It provides a specific implementation of the EMF resource that supports persisting models in JSON documents. EMFJSON does not support specific loading mechanisms. However, it is integrated with EMF and native loading mechanisms of EMF can be used. It does not support specific caching mechanism. EMFJSON does not support specific saving mechanisms. However, it is integrated with EMF and native saving mechanisms of EMF can be used. EMFJson supports querying models using the native EMF API. Persisted information could be queried also using JSON-specific query languages.

EMFFragments [Sch12, Sch13] is a framework that persists model information in model fragments instead of at model-element level. The approach can be used with NoSQL back-ends such as MongoDB, HBase or also with distributed file-systems. This approach groups model elements (and relationships) in different fragments, and then persists the fragments at the persistence back-end. The fragmentation strategy is specified by the users at metamodel-level and using annotations that are added over containment references. Using EMFFragments requires to perform a set of modifications in the EMF metamodel and API generation.

The fragmentation strategy provides support to load in memory only the fragments that are required. Hence, is not required to load entire model and the memory usage is decreased. Loading a fragment implies loading in-memory the model elements that are specified within it. Loaded fragments know if they are referenced by other fragments that are loaded in memory. Therefore, when all the strong references to a fragment content have disappeared, the fragment is collected by the garbage collector of the Java Virtual Machine (JVM). However, it has not been found any description about mechanisms for persisting modified elements. From the description, it can be deduced that EMF Fragments traverses the entire modified fragment from memory to the persistence back-end. EMF Fragments provides support for querying persisted models using the native EMF API.

EMFSplitter [Gar14] is an approach focused on the structured construction of EMF models that are persisted using XMI. It provides a set of model annotations (project, package and unit) that are used to specify the modularity strategy. Models are modularized at runtime, and EMF Splitter also supports modularization of a previously created and monolithic model instances. EMFSplitter structures each model instance physically in several XMI files.

The decomposition of a monolithic model in different physical files supports to load only these files that are related to the performed model operation. This strategy could decrease execution time and memory usage values. Using EMFSplitter, it would be sufficient to save only the XMI files that contain the modified model elements, requiring less time for performing save operation. Models that are persisted using this approach can be queried using the native EMF API (not any supported persistence-specific query language).

Other persistence prototypes Additional model persistence prototypes have been found in the literature. Two NoSQL database-based prototypes are presented at [Bag14]. One uses Neo4J database back-end for persistence and the other uses OrientDB. Both prototypes provide a graph-based solution for persistence, and they use transactions for inserting and modifying model elements. They support querying models using EMF API and EOL query language.

In [Sha14] a framework to benchmark NoSQL back-ends for large-model persistence is presented. This study contains a set of prototypes for model persistence that are based in the following database back-ends: *Neo4J*, *OrientDB*, *BerkleyDB*, *Cassandra*, *MongoDB*, *Sesame*, *ArangoDB*, *PostgreSQL*.

2.2.3 Classification and Comparison

Table 2.3 resumes features of the previously described persistences. Most of the approaches that provide alternative solutions for EMF model persistence have opted to use databases (8 from total of 11 approaches). Database-based approaches support persistence of different models at the same repository, and most of them provide persistence-specific mechanisms to load, cache and save model information. Moreover, to support all features of persistences, most of the approaches require to generate persistence-specific code. In the case of NeoEMF/HBASE there has not been found explicit evidence about if the code generation is required.

Only four approaches provide execution of queries using a persistence-specific query language that is integrated with EMF Resource: Morsa provides support for executing queries in MorsaQL language; MongoEMF using the MongoDB Core API; and Teneo and CDO using SQL and HQL. From the database-based approaches 6 provide support for model persistence using NoSQL databases. NoSQL databases have features that may be beneficial for model persistence [EP13b, p. 5]: scale better than relational databases; are schemaless; and provide mechanisms for accessing the information via HTTP or REST. However, there are two database-based approaches that provide persistence in relational databases: Teneo and CDO.

There are only three approaches that provide integrated versioning. All these approaches are database-based. In the case of Teneo and NeoEMF/HBase, versioning is supported by the underlying database back-end, but integration of versioning with EMF-based technologies is limited. In the case of CDO, versioning is integrated with EMF, and it is supported by the implementation of EMF Resource. Four of the approaches provide file-based persistence of models: Binary, where models are persisted in binary files; EMFJson, which persists models in JSON format; EMF fragments, which besides database-based persistence it also supports to store models in distributed file systems; and EMF Splitter which persists model partitions in XMI files. EMF Fragments and EMF Splitter require generation of code. From file-based approaches, EMFJson is the only one providing support for querying models in a persistence-specific way (using JSON). Only two approaches provide features for web-visualization of persisted models (CDO and EMFJson). And as table shows, CDO is the approach which is used more often for comparison with other persistences.

These persistence approaches are compared with other approaches in different studies. [EP13b] compares performance and memory usage of Morsa with XMI and CDO. Previous approaches are also compared together with EMF Fragment at [Sch12]. [Ben14] includes a comparison between XMI, CDO and NeoEMF/Graph. NeoEMF/Graph is also compared with NeoEMF/Map and CDO at [Góm15b]. [Bar12] compares different NoSQL prototypes for model persistence with XMI, Teneo and CDO.

All these studies compare XMI and CDO with other model persistence approaches. And results of comparisons show that each persistence approach is appropriate for an specific scenario.

Table 2.3: Features of persistence approaches.

Name	**Persistence-type**	**Database type**	**Back-End**	**Requires server**	**Requires code-gen**	**Persist. load**	**Persist. caching**	**Persist. save**	Versioning support	Web visualiz.	Multiple models	**Persist. spec. query**	**Evaluation**	**Compared with**
Morsa [EP13b]	DB	NoSQL	MongoDB	✓	✓	✓	✓	✓			✓	MorsaQL	✓	XMI, CDO, EMFFragments
MongoEMF [Hun14]	DB	NoSQL	MongoDB	✓	✓	✓		✓			✓	MongoDBAPI		
NeoEMFGraph [Ben14]	DB	NoSQL	Neo4J		✓[3]	✓	✓	✓			✓		✓	XMI, CDO, NeoEMF/Map
NeoEMFMap [Góm15b]	DB[4]		MapDB		✓[3]	✓	✓				✓		✓	NeoEMF/Graph, CDO
NeoEMFHBase [Góm15a]	DB	NoSQL	Hbase	✓		✓	✓	✓	✓		✓			
Teneo [Ten12]	DB	Relational	RDBMS	✓[5]	✓	✓	✓	✓	✓		✓	SQL, HQL	✓	XMI, CDO
CDO [Cdo16]	DB	NoSQL, Relational	RDBMS, MongoDB, ...	✓[5]	✓[3]	✓	✓	✓	✓	✓	✓	SQL, HQL	✓	Morsa, Teneo, NeoEMF/Graph, NeoEMF/Map, XMI, EMF Fragments
Binary	File	-	Binary file											
EMFJson [Hil15]	File	-	JSON file							✓		JSON		
EMF Fragments [Sch13]	File, DB	NoSQL	several		✓	✓	✓				✓		✓	XMI, CDO, Morsa
EMF Splitter [Gar14]	File	-	XMI files		✓									

[3]Generation is required to support all features of the persistence approach.

[4]MapDB provides a database-engine but not database-based persistence.

[5]Requires a separated server if the database is not executed in embedded-mode.

3 Model Queries

Model queries are intensively used by modelling tools and modelling engineers. Therefore impact of query performance over tools performance and user experience is significant [Ber10]. In this chapter, first languages that can be used to query EMF models are described, classified and compared. Next, different approaches for query language transformation are described and classified. This chapter ends with a critical analysis of existing approaches which motivates the work done in this dissertation.

3.1 Languages for Querying Models

There are different query languages that provide support for querying EMF models. Different factors for classifying query languages are described below.

3.1.1 Classification Factors for Query Languages

Query languages have been classified depending on: (C1) where they are executed; (C2) abstraction-level of the language; (C3) type of the language; and (C4) modification support. Following each of these characteristics is described with more detail:

C1: Execution. It specifies the location where the query is executed for processing models and calculating results:

- **Memory.** Majority of the approaches that provide database back-end based persistence of models support the execution of EMF-based query languages. These languages require to load in-memory at least all the model elements that are involved in the query (as occurs on XMI).

- **Persistence.** Most of the database-based model persistence approaches also support execution of queries directly over the persistence. This way, queries leverage capabilities of the persistence and provide faster queries or less memory usage.

C2: Abstraction-Level. It specifies which type of abstractions are handled by the query language syntax:

- **Application Programming Interfaces (API).** This group includes the Application Programming Interfaces (APIs) that support executing model queries using Java programming language. APIs are appropriate for engineers that have Java-programming knowledge. Additionally, using the API requires low-level knowledge of the provided API.

- **Model Query Languages (MQL).** This group includes query languages that are focused on interacting with models using abstractions and independently of the used persistence mechanism. If the model-persistence mechanism is changed or evolves, queries expressed using a model query language still remain valid. This type of languages are closer to modelling engineers.

- **Persistence-Specific Query Languages (PQL).** These query languages are specific and it depends on a particular persistence back-end. Main advantage of languages of this type is that they leverage capabilities of the persistence and the persistence-engine could optimise them. By contrast, they require to expose modelling engineers to the persistence directly, and this is error prone and requires to learn each used persistence: which besides learning language syntax it requires also to know things like how the information is persisted (e.g. learn about the data-schema if a relational database is used) or best practises of the persistence.

C3: Type. It specifies the type of the query language:

- **Imperative**. These are the query languages where the code describes exactly how to perform the query. Hence, the program in a imperative language executes query statements exactly as they have been written. Imperative queries explicitly describe the steps that will be executed during query execution to obtain the results.
- **Declarative**. Queries specified with a declarative language describe the logic of the query, but they do not specify how the query has to be executed. In this sense, declarative query languages use higher abstraction in the specification and they allow the engineer to focus on the domain problem.

C4: Modification Support. It specifies if the query language supports modifying models. Thus, possible options of this characteristic are *yes* or *no*.

3.1.2 Existing Query Languages

Main features of the identified query languages are resumed below:

EMF API is a pure Java API that is provided by EMF, and it is the simplest solution for querying EMF models [EP14, p. 12]. EMF API supports querying EMF models which includes read-only and modification queries. It supports querying and modifying different models at same time. The used language is Java, a imperative programming language. EMF API provides a dynamic implementation that is able to query models, independently of the domain that they conform to. Additionally, EMF provides generation of a domain-specific implementation of the EMF API. EMF uses the domain metamodel for generating automatically the source-code that composes the API. Both dynamic and generated APIs require to load in-memory at least all the model elements that are involved in the executed operation or query. In the case of a model query that fully traverses a model, EMF API requires to load in-memory the entire model.

Object Constraint Language (OCL) [Gro16] is an OMG standarised declarative model query language for specification of formal expressions within UML models. OCL queries are executed over models without modifying it and they are used to specify invariants, pre-conditions, post-conditions guards or to obtain information from models. Main limitation of OCL is that it does not support features like: statement sequencing with variables, accessing concurrently to different models and modifying models. Although OCL was designed for UML models, it can be executed over EMF models using solutions such as OCLinEcore [Wil12]. OCLinEcore is integrated with the EMF API and it can be used in a transparent way in EMF-based tools. Therefore, OCL queries can be used in the same scenarios where models are queried using the EMF API. OCL queries are executed over model elements that have been previously loaded in-memory. Similar to EMF API, if executed queries fully traverse the model, it requires to previously load in-memory all model elements.

There are persistence approaches that provide persistence-specific execution of OCL queries. This is the case of CDO, which supports the execution of OCL queries at server-side, and instead of loading required model elements in the client memory, they are loaded in the server-side. Therefore, query is not evaluated directly over the database where models are persisted, and it requires first loading queried elements in the servers' memory. Then, the query is executed at server-side over the loaded information [EP14] and finally results are returned to the client-side.

Epsilon Object Language (EOL) [Kol06] is a OCL-like query language that is the core of Epsilon [Kol08], a family of tools and languages for models. Epsilon family languages, and concretely EOL, provide to the model engineers ability to perform operations over models with concepts that are close to their domain-specific knowledge. EOL provides integration with EMF and it can be used to query models in the EMF-based tools. EOL is an imperative model query language based on OCL that combines features of JavaScript: on the one hand, it provides imperative features of JavaScript such as statement sequencing, the use of variables and loops (for and while); on the other hand, it provides OCL features like collections and operations for querying collections (i.e. select or collect). Therefore, EOL supports a set of features that are not supported (at-least natively) in other declarative languages for model query such as OCL [Kol08]: (i) accessing multiple models,(ii) sequence statements, (iii) integrate simple programming idioms; and (iv) modify models. EOL queries are executed over model elements and it requires to previously load in-memory at least involved model artifacts. For queries that fully traverse a model EOL requires to load in-memory the entire model.

EMF-IncQuery [Ujh15] is a model query language that focuses on the incremental execution of queries. The language is declarative and it is based on graph pattern concepts that allow specifying complex queries in an easy way. It has been created using XText [Eff06], thus it provides a query editor with features such as validation, code assist or syntax highlighting. IncQuery is integrated with EMF and EMF API. IncQuery provides incremental graph pattern matching techniques to increase performance of queries when they are re-executed. Therefore, the incremental execution of queries does not require executing already queries that have been evaluated before. The incremental execution uses a set of indexes and calculations that are initialized in the first execution of the query. Thus, queries are executed in-memory over model elements that have been previously loaded.

EMF Query [Emf15] is an EMF-based API for querying EMF models. It is integrated with the EMF API and uses concepts of SQL queries like *select*, *from* or *where*. However, it has not any relation with SQL databases and executes queries over model elements which have to be previously loaded in-memory. While other solutions such as the previously described OCL or EOL provide a language for specifying queries, this approach provides an API. It is a Java API and consequently, it has imperative nature. However, as previously described, the classes provided by the API are based on concepts of SQL, and SQL is declarative. The query is specified using SQL-like concepts. Thus, it has been classified in the group of the declarative languages. EMF Query supports specification of queries that modify the model. However, modification support is similar to the provided by updates in SQL.

EMF Query 2 [Emf12] is an EMF-based API and an extension of EMF Query which is focused on scalability of queries. It provides two different syntax for writing queries: a SQL-based syntax; and an AST-based syntax. The SQL-based syntax is declarative. The AST-based syntax is an imperative Java API. However, the query is expressed in a declarative way. Thus, both syntaxes have been considered as declarative. EMF Query 2 aims to minimize loading of models by providing an indexing mechanism that avoids loading them wherever needed. The use of indexes is required and although the loading mechanism is improved, queries are executed over the information that has been previously loaded in memory. It has not been found information about query expressions for modifying models.

MorsaQL [EP14] is a persistence-specific query language that supports querying models that have been persisted in Morsa repositories. MorsaQL tries to minimize the amount of data that is transferred between client and server when queries are executed. This approach allows to take advantages of Morsa (e.g. load on demand) when querying models, and queries require less memory to be executed [EP14, p. 618]. Queries are specified in a declarative way using a syntax that similar to SQL and uses the Select-From-Where schema. There has not been found information about support model modifications.

Structured Query Language (SQL) [Sql16] is a query language that is specific for relational databases. The language is declarative and it is the standard language that is used to query and operate information that is persisted in relational databases. The SQL queries are executed directly over the database, and the database-engine is responsible for optimizing and resolving them. SQL supports read-only query expressions, and also expressions to modify persisted information.

Hibernate Query Language (HQL) [RHM04] is a Hibernate-specific query language. This language is declarative and the syntax is similar to SQL. While SQL operates over tables and columns, HQL is object-oriented and it operates directly over persistent objects and properties. HQL executes queries at server-side and using Hibernate. HQL supports both read-only and modification query expressions.

Cypher [Cyp15] is a graph query language that is specific for Neo4J databases. This language is declarative and its design has been inspired in SQL. Queries are executed at server-side over a Neo4J database. Cypher supports specification of queries that modify the information persisted within the database.

Neo4J Core API is a Java API that is used to query with Java code the information that has been persisted in a Neo4J database. It is executed directly over the Neo4J database and it provides support for specifying queries that modify the information persisted in the database.

MongoDB Query Language [Mon15] is the language that is used for querying the information or documents that are persisted within MongoDB databases. The language is declarative and queries are executed directly over the database. The queries can be read-only or also can perform modifications in the persisted information.

3.1.3 Query Language Classification

Table 3.1 resumes main characteristics of the previously described query languages:

Table 3.1: Characteristics of different query languages.

	Execution	**Abstraction-Level**	**Type**	**Modif.**	**Supported by**
EMF API	Memory	API	Imperative	Yes	EMF-based
OCL [Gro16]	Memory	MQL	Declarative	No	EMF-based
OCL (server-side) [Cdo16]	Persistence	MQL	Declarative	No	CDO
EOL [Kol06]	Memory	MQL	Imperative	Yes	EMF-based
EMF IncQuery [Ujh15]	Memory	MQL	Declarative	No	EMF-based
EMF Query [Emf15]	Memory	API	Declarative	Partial	EMF-based
EMF Query 2 [Emf12]	Memory	MQL/API	Declarative	No	EMF-based
MorsaQL [EP14]	Persistence	API	Declarative	Yes	Morsa
SQL [Sql16]	Persistence	PQL	Declarative	Yes	CDO+Hibernate, CDO+DBStore, Teneo
HQL [RHM04]	Persistence	PQL	Declarative	Yes	CDO+Hibernate, Teneo
Cypher [Cyp15]	Persistence	PQL	Declarative	Yes	NeoEMF/Graph
Neo4J Core API [Neo16]	Persistence	API	Imperative	Yes	NeoEMF/Graph
MongoDB QL [Mon15]	Persistence	PQL	Declarative	Yes	MongoEMF, CDO+MongoDB

From the identified APIs, EMF API, MorsaQL and Neo4J Core API provide support for full-modification of the persisted information. In the case of EMF Query, it partially supports modifications, but they are not supported in EMF Query 2. Type of the language in the APIs is imperative in the case of EMF API and Neo4J Core API, but EMF Query, EMF Query 2. MorsaQL uses imperative Java code extended with a SQL-like declarative syntax. EMF Query, EMF Query 2 APIs are executed in-memory, and MorsaQL and Neo4J Core API are executed at server-side.

All the model query languages support querying EMF models. However, EOL language also supports to query information that is outside the EMF ecosystem. From these languages, only EOL provides modification support. All the model query languages are executed in-memory, with the exception of OCL that can be executed at server-side using CDO.

In the case of the persistence-specific query languages, all are declarative and each one is specific of a different database.

3.2 Transformation of Query Languages

There are approaches that provide transformation from a query language into a different query language. This section, first provides different factors that have been used to analyse and classify these approaches. Then, identified approaches are described and finally they are classified and compared.

3.2.1 Classification Factors

Different classification factors have been identified for the analysis of approaches that transform queries expressed with a concrete query language into a different language. Classification factors are described in Table 3.2.

Table 3.2: Description of factors for classifying persistences.

Factor	Description
Model Type	Specifies the modelling technology (e.g. EMF, UML,etc.)
Input Language	The query language that is transformed.
Intermediate Results	Intermediate artifacts that are produced by the approach during the transformation process (if they are produced).
Output	Format of the transformed query.
Target	Specifies the target where the transformed query is executed.
Supports Mapping	Specifies if the also provides mapping of models between different persistences.
Incremental Execution	Specifies if the generated queries are executed incrementally.
Lazy Execution	Specifies the ability of the approach for executing generated queries lazily.
Evaluation	There is evidence in the literature that the approach has been evaluated.

3.2.2 Query Language Transformation Approaches

Following main features of identified approaches for query language transformation are resumed:

[Mar99] presents an approach for checking OCL constraints over repositories for UML models that are based on relational databases. OCL constraints are used to check the validity of the UML models, and also maintaining consistency of the application data.

The approach is able to map the UML metamodel with the database-schema, and the OCL invariants defined at the UML metamodel with SQL queries to be executed over the database. To generate SQL queries corresponding with OCL constraints, the approach first generates an intermediate graph representation. This representation contains two kind of nodes: one for specifying the SQL code generation algorithm and the other for providing information about the queried UML model and the mapping with the corresponding data-schema.

Query Code Generation Framework [Hei07] provides mapping of UML models in arbitrary data-schemas, and mapping of OCL queries in the corresponding declarative query languages. The framework is composed by three modules: the first module is responsible for generating abstract syntax model from UML/OCL models; the second module (Model Transformation Framework) is responsible for mapping UML model with the target data-schema; and the third module (OCL Transformation Framework) is responsible for mapping OCL constraints to declarative query languages.

The queries that are generated by the OCL Transformation Framework are specific for a target query language. In addition to the OCL query to be transformed, this framework uses the information about mapping between UML model to be queried and selected target data-schema. Two different prototypes that use Query Code Generation Framework are presented at [Hei07]. One for transforming OCL queries into SQL queries that are executed over a relational database. And other for transforming OCL into XQuery and execute over models persisted in XMI files.

UMLtoCSP [Cab07] is an approach focused on automatically checking correctness properties over UML/OCL models. UMLtoCSP uses the Constraint Logic Programming and ECL^iPS^e constraint solver.

The approach inputs UML class diagram, OCL constraints and property to be verified. Using the Dresden OCL toolkit, the OCL constraints and properties to be verified are transformed into a Constraint Satisfaction Problem (CSP) file. CSP file is the input of ECL^iPS^e which returns if the property holds or not, and if it does, a model instance that certifies it is depicted.

[Win08] presents an approach that translates a subset of OCL constraints into graph constraints. Supported OCL constraints are these that express equality, size and attribute operations. The generated graph constraints specify properties that have to be satisfied by graphs. This solution extends an approach that provides a graph grammar for automatically generating instances from a given UML metamodel [Ehr05]. This way, the approach adds support for checking constraints during the instance generation process.

The transformation process is performed in two steps. In the first step, the OCL constraints are transformed into graph constraints. Then, in the second step, graph constraints are transformed into application conditions [Ehr04], and consequently, constraints are considered during the instance generation process.

OCL2Trigger [AJ08] supports the transformation of OCL constraints into triggers specific for a target Database Management System (DBMS). Additionally, the approach performs verification of the generated trigger with the purpose of ensuring its execution.

The tool extends Rational Rose with a three-step transformation of constraints. The first phase consist on the definition of OCL constraints, where some constraints are specified directly using OCL and others are specified at the graphical model. Constraints of the second type are automatically transformed into corresponding OCL constraints. In the second phase, first OCL constraints are transformed into standard SQL triggers, and then they are transformed into DBMS-specific triggers. Finally, in the third phase the generated triggers can be completed by the stakeholders and trigger execution is verified using a sequence diagram.

Dresden OCL [Dem09] is a tool that provides OCL support for UML and EMF tool builders. The first version of the tool supported syntax and type checking of OCL constraints specified at UML models and generation of Java and SQL (OCL2SQL [Dem01]) code corresponding with OCL queries [Dem04]. In a posterior version of the tool, a intermediate model (pivot model) has been used during the transformation process. Pivot model provides an intermediate abstraction layer, and it makes the tool independent from specific repositories and meta-models [Brä07].

Dresden OCL architecture is composed by three layers: back-end, base and tools layers. Back-end layer specifies the used repository and metamodel. Base-layer contains: the pivot model; the essential OCL which implements the OCL

Standard library by extending the pivot model; and model bus which loads, manages and provides access to metamodel, models and model instances [Dem09, p. 2]. And tools layer contains the tools used by the second layer to load, verify and check OCL constraints. The tool is able to check OCL constraints using both interpretative and generative approaches. [Dem09] includes description of different interpretative and generative use cases for the Dresden OCL tool.

MySQL4OCL [Ege10] is an approach that generates SQL queries from OCL expressions. Generated queries are executed over MySQL databases where the information of UML models is persisted. The approach supports a subset of OCL queries, and it excludes generation from: OCL operations on sequences and ordered-sets; operations on collections of collections; operations on types; and user-defined operations.

Generated MySQL code is for a specific data-schema provided by the approach. This data-schema is composed by three different types of tables: one table per class type, containing attributes of the class type; and one table per each type of association between classes.

The generation mechanism is recursively defined over OCL expressions structure, and the generated SQL code is different depending on the type of the OCL expression. MySQL4OCL generates common SQL queries for non-iterator OCL expressions. By contrast, the approach generates stored procedures for OCL iterator expressions.

[Ege10] provides a preliminary evaluation of the approach where the time required for executing OCL expressions using MySQL4OCL and Eye OCL Software (EOS) [Cla08] are compared. Results show that MySQL4OCL requires more time for small-medium size scenarios. However, MySQL4OCL shows better results than EOS in a large scenario.

[Kol13] proposes transformation of a relational dataset into a EMF model. This way, the information can be queried using OCL-like query languages. However, the naive way of evaluating OCL-like queries on relational datasets can dramatically degrade performance [Kol13, p. 4]. Therefore, this work presents a solution improving performance when executing EOL (OCL-like) queries over a relational dataset.

Proposed solution has two main features: it uses lazy collections that allow to load on demand from database only the model elements that are required by the query; and it performs the query translation at run-time using a multi-step mechanism. This mechanism translates one-by-one and at runtime the different expressions within the query. Hence, at the end of the process the complete SQL query is obtained and executed over the database.

[Ber14] presents an approach that constructs EMF-IncQuery [Ber10] graph patterns from a subset of OCL expressions. EMF-IncQuery graph patterns provide support for incrementally execute queries over EMF models. While OCL expressions are typed functions, graph patterns evaluate to match sets that are mathematical relations [Ber14, p. 7]. Thus, the approach identifies equivalent OCL expressions and maps them with equivalent graph patterns.

The approach uses structural recursion: first, maps each OCL sub-expression to graph pattern; and then, all the produced patterns are used to generate the pattern that specifies whole expression. The approach is evaluated using the Train Benchmark Case [Ujh15]. The execution of generated queries is compared with the execution of the query using other languages.

[Ori15] presents an approach that translates OCL constraints into SQL queries which return database elements that violate the constraint. The approach is focused on the incremental checking of constraints, and generated SQL queries are only re-executed when a data update could violate the corresponding constraint, and the query only queries data related to the update.

The queries are translated at compilation-time and using a two-step translation mechanism. In the first step OCL constraints are translated in Event Dependency Constraints (EDC). EDC specify events that describe states of the data that cause violation of a constraint. In the second step, EDCs are translated into SQL queries. SQL queries are specific of a data-schema that is provided by the approach, and which is able to persist in a relational database UML classes and associations, structural events being applied, and aggregated values. Generated SQL queries join structural events, current data and current aggregated values, and it allows to: (1) only check constraints that can be violated by the update; (2) focus only in the related data and avoid checking entire database; and (3) avoid recomputing unnecessary aggregations. The scalability of the approach is also evaluated in [Ori15], obtaining time metrics for constraint checking and for updating materialized aggregates.

[Kal16] presents an approach that generates Java meta-programs from OCL constraints that are defined at metamodel-level. Meta-programs allow to check specified constraints at run-time.

The approach provides a three-step process for the generation. In the first step, OCL constraints are rewritten in order to make them more accurate and concrete. This step is not mandatory, and it is performed when the abstractions in the UML metamodel have not equivalence in Java. Then, in the second step, the OCL constraint specified over the UML metamodel is migrated to specify the metamodel using Java. Abstract Syntax Tree (AST) that correspond with the constraint can be used in the query generation performed at third step. Thus, the third step generates source-code that corresponds with the OCL constraint. Generated code can be used to check constraints at run-time.

3.2.3 Classification of Approaches

Table 3.4 summarizes classification factor values for identified approaches. As table shows, most of the approaches provide transformation of queries that are executed over UML models (8 from total of 11). Three approaches are valid for EMF models and only one approach is for models persisted with Rational Rose.

Focusing on the language of the queries to be transformed, most of approaches input queries expressed with OCL (10 from total of 11). Only one of the approaches inputs queries expressed in a different language (EOL).

Seven approaches produce intermediate results during the transformation process. And SQL is the most used query language for generated queries. Thus, the generated queries are executed in most cases over relational databases.

Three of the approaches provide a mechanism for mapping queried models into alternative persistences: [Mar99], [Ege10] and [Kol13]. Only two approaches provide incremental execution of the generated queries and they are [Ber14] and [Ori15]. And [Kol13] supports lazy execution of the generated queries.

Two of the analysed studies provide also an evaluation of the corresponding approach ([Ber14] and [Ori15]).

Table 3.4: Overview of the identified approaches for model query language transformation.

	Model Type	**Input Lang**	**Interm. Results**	**Output**	**Target**	**Supports mapping**	**Increm. exec.**	**Lazy exec.**	**Eval.**
[Mar99]	UML	OCL	graph	SQL	RDBMS	✓			
[Hei07]	UML	OCL	-	SQL Xquery	RDBMS XMI				
[Cab07]	UML	OCL	-	CSP	Eclipse				
[Win08]	UML	OCL	graph constraints	graph app. cond.	graph grammar				
[AJ08]	Rational Rose	OCL	SQL Triggers	RDBMS triggers	RDBMS				
[Dem09]	UML EMF	OCL	Pivot Model	SQL Java	RDBMS Java[1]				
[Ege10]	UML	OCL	-	SQL	MySQL DB	✓			
[Kol13]	EMF	EOL	-	SQL	RDBMS	✓		✓	
[Ber14]	EMF	OCL	part. graph patt.	graph pattern	EMF-based		✓		✓
[Ori15]	UML	OCL	EDC	SQL	RDBMS		✓		✓
[Kal16]	UML	OCL	AST	Java	Java[1]				

[1] instead of executing it, java code is added to the existing source-code

3.3 Critical Analysis

Different studies have analysed and compared EMF model persistence approaches [EP13b, Ben14, Bar12], concluding that each persistence approach is appropriate for an specific scenario. For example, XMI is appropriate persistence for small-size models as do not cause memory problems.

However, modelling scenarios in industry can be really complex [Kär09], with large models of 100MB and beyond, and with millions of model elements. This is the case of embedded system domains such as wind-power or railway, where systems can comprise a large number of elements such as sensors, actuators and control units [Bag14]. To effectively support such domains, alternative persistence mechanisms for large-scale models are required [EP13b].

Most recent approaches for EMF model persistence that have been identified use database back-ends for model persistence. These approaches provide persistence-specific implementation of the EMF resource, and consequently, they can be integrated with the EMF-based modelling tools. Most of these database-based solutions use NoSQL database back-ends (Morsa, NeoEMF/Graph, NeoEMF/Map, MongoEMF, EMF Fragments, CDO). However, there are solutions that use relational databases for model persistence (Teneo and CDO).

Among all the activities, model queries are intensively used in modelling scenarios. Therefore the impact of query performance on tool performance and user experience is significant [Ber10]. In practise, queries that traverse entire model are the most commonly used type of queries [Góm15b].

All the analysed persistence approaches provide support for executing queries using the EMF API, and consequently, they are also able to execute EMF-based model query languages (e.g. OCL, EOL or IncQuery) and APIs (e.g. EMF Query or EMF Query 2). Model query languages are commonly used by model engineers since they are closer to their knowledge, and are focused on interacting with models using abstractions. Moreover, they are persistence-agnostic and they do not imply to know how the information is persisted. Model query languages are commonly executed in-memory and they entail to load first model elements before executing queries. In the case of model traversal queries, entire models has to be loaded in-memory.

Some persistences also provide a persistence-specific way for executing queries over persisted models. In these cases, persistence-specific query languages (e.g. SQL, HQL, MorsaQL, etc.) are used to specify queries. These query languages are executed at persistence-side and they leverage capabilities of persistence. For example, they are executed directly over the database and they do not require to load intermediate results in memory. However, persistence-specific languages require engineers: to be aware of the way the information is persisted; and to learn persistence-specific concepts and languages. This increases programming effort to get complex queries correct.

Using native queries over the underlying persistence back-end provides significant gain in performance [Góm15b, p. 16]. This fact has been validated with a preliminary study which compared performance and memory usage for executing a complex query (GraBaTs case study query[2]) using OCL (model query language, executed at client- and server-sides) and SQL (persistence-specific language executed at server-side). Models (Set0-Set4) had been persisted in a relational CDO repository with the default mapping strategy, and they contain abstractions that specify Java source-code. Models size increasingly grows from 15MB of Set0 to 1.17GB of Set4. Table 3.5 illustrates the results obtained in one execution of queries using both languages and over each model. As results show, queries executed at server-side require less time and memory. Moreover, time and memory results in SQL are better than OCL as the size of the model increases.

Table 3.5: Memory and execution time results using OCL and SQL.

	OCL (client-side)		**OCL (server-side)**		**SQL (server-side)**	
	Time(s)	**Mem(MB)**	**Time(s)**	**Mem(MB)**	**Time(s)**	**Mem(MB)**
Set0	21	322	5	73	4	68
Set1	50	758	5	73	4	69
Set2	463	2955	15	343	6	150
Set3	1028	5644	33	525	9	289
Set4	1102	5973	36	584	9	289

Several approaches have proposed solutions for generating persistence-specific query languages from model query languages. However, most of the solutions provide support for querying UML models. In the case of these approaches, OCL is the query language to be transformed.

Only three approaches provide support for EMF models. [Dem09] presents DresdenOCL tool which is able to translate queries into Java or SQL queries. [Ber14] presents an approach that translates OCL queries into graph patterns. These two approaches are specific for OCL queries. The third approach is described at [Bar14] and it transforms EOL queries into SQL queries. Generated queries are specific for a specific data-schema and persistence.

[2]Information about the case study and query is extended at Chapter 7

Current state of the art motivates the challenge of providing an extensible approach that transforms queries from model query languages into persistence-query languages. The approach aims to resolve the following research questions:

- Are the database-specific query languages more efficient than model query languages in terms of performance and memory usage when models are persisted in databases?
- Which mechanism can be used to provide a solution that transforms queries from a model query language to a persistence-specific query language in an extensible way?

Part II

Contribution

4
Theoretical Framework

This chapter presents theoretical aspects of this dissertation: followed research methodology, hypotheses, goals and case studies used to validate implemented solution.

4.1 Methodology

This work has been completed following a previously defined research methodology. Figure 4.1 depicts different stages of the selected methodology and they are described below:

1. **Problem Awareness.** Analyse the problem and motivate it by answering questions such as: *Why is important? What solves it?* Context of the problem has been analysed and described (Chapter 2).

2. **Methodology.** Define methodology to be followed: proposed work plan, identify literature and technical sources, etc.

3. **State of The Art.** Identify existing approaches and solutions related to the research problem. This stage has been addressed performing an analysis of the state of the art (Chapters 2 and 3). Different approaches for model persistence and query have been identified and compared.

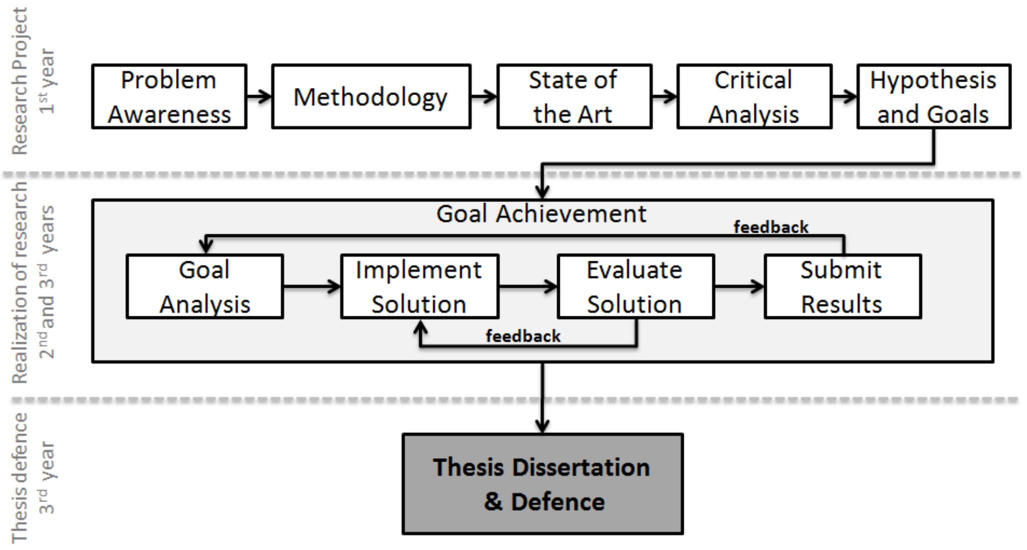


Figure 4.1: Overview of the followed research methodology .

4. **Critical Analysis.** Perform a critical analysis of the state of the art and identify: what is done, existing limitations and weak points, points to contribute, etc. Critical analysis of the existing work is described at the end of Chapter 3.

5. **Hypothesis and Goals.** Formulate the hypothesis of the thesis and set goals to be performed. These goals will be used to validate or reject the hypothesis. They are described in Sections 4.2, 4.3 and 4.4 of this chapter.

6. **Goal Achievement.** Goal achievement implies (for each setted goal): (i) analysing the goal problem; (ii) implementing a solution; (iii) evaluating the solution; (iv) submitting results; and (v) re-factorizing solution from obtained results and feedback. This phase is covered by different chapters of this dissertation: Chapters 5 and 6 describe the approach that has been designed and implemented. The approach has been validated with two experimental evaluations in Chapters 7 and 8. Finally, conclusion and hypothesis validation is described in Chapter 9.

7. **Thesis Dissertation.** Write the thesis and perform the thesis defence.

4.2 Hypothesis

Hypotheses of this research are all related with MDD and more concretely with persistence and query of models:

- **Hypothesis A**: *if models are persisted using a database, querying them using a database-specific query language is more efficient (in terms of performance and memory usage) than using a model query language.*

- **Hypothesis B**: *transformation of queries from a model query language to a persistence-specific query language provides model engineers the ability to query models using a language that is closer to their knowledge, but with the efficiency (execution time and memory) of a persistence-specific query language.*

4.3 Goal

Goal A. The main goal of this dissertation is to provide an approach that verifies (or rejects) the previously formulated hypotheses. This work aims the implementation of a framework that provides transformation of queries from model query languages to a persistence-specific query language (*Goal A*). The framework will perform the following activities:

- Query transformation. Transform the query expressed with an input model query language into a query expressed with a target persistence-specific query language.

- Query execution. Execute the generated query into the corresponding persistence and using a native and persistence-specific mechanism.

- Process results. Process the results and adapt them to fit with the expected type.

Goal B. The framework will provide a design that facilitates the inclusion of new model query languages and persistence-specific query languages.

Goal C. Evaluate the framework and compare results with other approaches for model query.

4.4 Operative Goals

This work has been focused on design, implementation and evaluation of the framework that achieves resolution of the previously described goals A,B and C. The work has been executed following a set of operative goals:

- **Operative Goal A.** Design of the framework. The framework will perform transformation and execution of queries, and processing of the obtained results. The design will facilitate inclusion of query languages.
- **Operative Goal B.** Implement a prototype. Implement a prototype of the framework with support for a model query language and for a persistence-specific query language. This prototype will be able to execute queries expressed with the model query language over models persisted using the selected persistence.
- **Operative Goal C.** Perform experimental evaluation of the implemented prototype. Experimental evaluations will be performed using existing case studies. Experimental evaluation case studies are introduced in the following paragraphs.

4.5 Case Studies

Two different case studies have been used to perform the experimental evaluation of the approach presented at this dissertation. Both case studies are widely used by approaches that are focused on model persistence and query, for evaluating their scalability and performance.

Each case study provides a different scenario for the experimental evaluation:

- **Reverse Engineering Case Study**: An academic case study that was proposed at Graph-Based Tools 2009 (GraBaTs 2009) [Gra09, Sot09]. The case study is based on the reverse engineering domain, where models specify Java source code. Moreover, a complex query is proposed with the aim of providing a scenario that compares performance and memory usage metrics of querying models. This query extracts all the singleton classes existing within the queried models. This case study has been previously used to evaluate several approaches related to the model persistence and query [Bar14, EP14, Sha14].

- **Train Benchmark Case Study**: A case study that is close to an industrial domain, railway domain concretely. The train benchmark case study proposes the use of models that specify railway systems [Szá15]. These models contain abstractions of the railway domain such as routes, segments, switches, semaphores, etc. Moreover, the case study proposes a set of queries and transformations that could be used in real scenarios. With this ecosystem composed by models, queries and transformations, the Train Benchmark Case aims to provide a scenario that evaluates the performance of approaches for executing queries. The train benchmark case has been used to evaluate an approach for incremental model query [Ujh15].

Both case studies are suitable for the evaluation of approaches that provide mechanisms for persistence or query of EMF models. They have been selected for performing two independent experimental evaluations of the approach presented at this dissertation. Both experimental evaluations are complementary, and each one provides different scenarios and results that complete the evaluation of the proposed approach:

- Different domain complexity: the reverse engineering case study uses a domain metamodel composed by a huge amount of classes (more than 100 different classes). By contrast, the railway metamodel contains around ten different classes.

- Different nature of models: the reverse engineering models are previously defined and they specify code of selected java project. Train benchmark case models are generated automatically and they provide mechanisms that break symmetry and prevent efficiently storing and caching models [Szá15].

- Different nature of queries: the reverse engineering case proposes one complex query that combines different types of subquery expressions. By contrast, train benchmark provides a set of queries with a different complexity level.

A more detailed description about each case study is provided within Chapters 7 and 8 of this dissertation.

5 Foundations of MQT-Engine Framework

The MQT-Engine framework inputs model queries expressed with a model query language, transforms them automatically to a database-specific query language and then executes the generated query. The approach makes possible to query models using a model query language that is closer to model engineers, but with the efficiency (in terms of performance and memory usage) of a persistence-specific query language that is closer to the persistence mechanism.

This chapter is organized as follows: first section provides a detailed overview of the MQT-Engine framework where execution process, involved roles and architecture of the framework are described; the chapter continues with the description of the QLI Metamodel that is used during the query transformation; the chapter finishes with a lower-level description of the design and execution of the framework.

5.1 MQT-Engine Overview

MQT-Engine transforms input queries in a model query language into equivalent queries in a persistence-specific query language, and then executes them over the persistence.

Figure 5.1 depicts the overview of the query transformation and execution

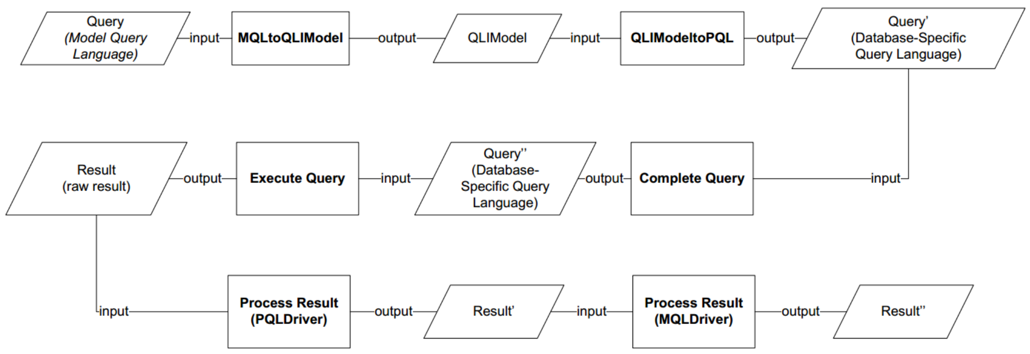

Query (Model Query Language)
input
MQLtoQLIModel
output
QLIModel
input
QLIModeltoPQL
output
Query' (Database-Specific Query Language)
input
Complete Query
output
Query'' (Database-Specific Query Language)
input
Execute Query
output
Result (raw result)
input
Process Result (PQLDriver)
output
Result'
input
Process Result (MQLDriver)
output
Result''

Figure 5.1: Overview of the query transformation and execution process of MQT-Engine.

process: it starts with *'MQLtoQLIModel'* that inputs the query that is expressed with a model query language (*Query*) and transforms it into a query language-agnostic model (QLI Model). This model conforms to a metamodel (QLI Metamodel) containing abstractions that are independent of the query language. The generated *QLI Model* is the input of *QLIModeltoPQL* which transforms the QLI Model into a query (`Query'`) that is expressed with a persistence-specific query language. Next, the generated query is modified (*Complete Query*) for adding information obtained from the model to be queried (e.g. add version information) and it is executed (*Execute Query*) over the persistence. The query mechanism of the persistence returns raw results that have to be processed for selecting the results that correspond to the executed query. This processing task is performed by the *PQLDriver*. And finally, results are processed by the *MQLDriver*. This second processing is depends on the model query language, and it formats the obtained results to the data-types provided and required by the model query language.

5.1.1 Involved Roles

Stakeholders of different types participate in the use of the MQT-Engine framework. They are depicted in the use-case diagram of Figure 5.2:

Model Engineers. The framework is addressed to model engineers. Model engineers are the stakeholders that operate models. Therefore, among other activities (e.g. query definition, model edition, etc.), they are responsible for

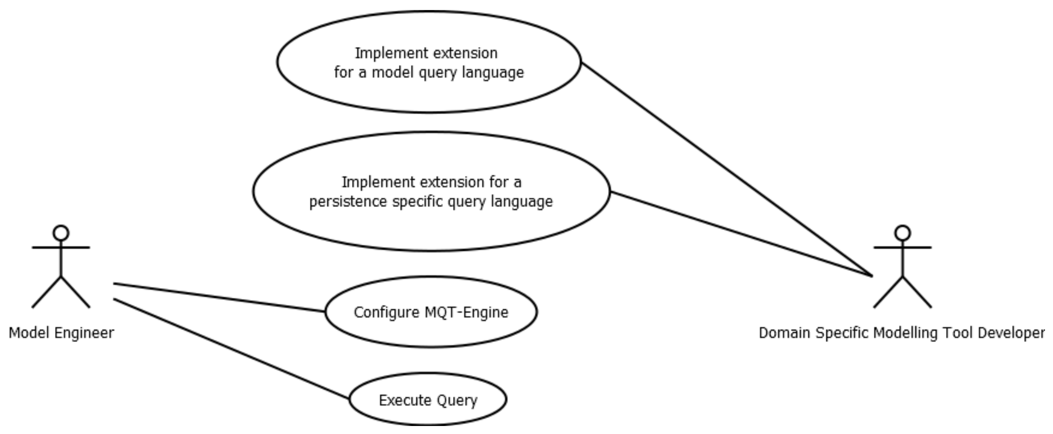

Implement extension for a model query language
Implement extension for a persistence specific query language
Model Engineer
Configure MQT-Engine
Execute Query
Domain Specific Modelling Tool Developer

Figure 5.2: Use case diagram of MQT-Engine framework.

executing the queries using MQT-Engine. Model engineers are also responsible for configuring MQT-Engine before querying.

Domain-Specific Modelling Tool Developers are the stakeholders that handle the development of domain-specific modelling tools. Therefore, domain-specific modelling tool developers are responsible for implementing extensions that add support to the MQT-Engine framework for concrete model query languages and persistence-specific query languages.

5.1.2 Architecture

Figure 5.3 illustrates a package diagram where different parts of the MQT-Engine architecture are depicted.

Domain package contains domain-specific abstractions. It provides support for interacting with domain-specific models and it is used by *MQT-Engine-Core*.

MQT-Engine-Core.QLIMetamodel implements abstractions used for specifying model queries in a query-language independent way. This metamodel is conformed by QLI Models that are used to: (1) specify the model queries with the query language-agnostic abstractions; and (2) generate persistence-specific queries.

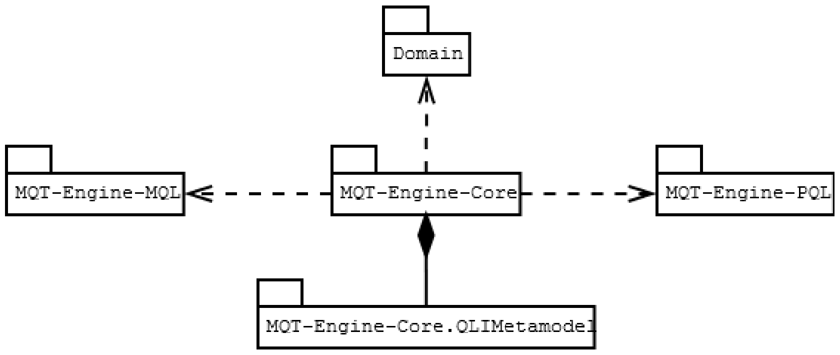


Figure 5.3: Package diagram specifying architecture.

MQT-Engine-Core package is the core part of the framework which contains a set of classes that execute the query transformation and execution. This package contains implementation of the *QLI Metamodel*. With extensibility in mind and to facilitate inclusion of new languages and persistences this package contains a set of abstract classes to be extended by packages that add support for model query languages (*MQT-Engine-MQL*) and persistence-specific query languages (*MQT-Engine-PQL*).

MQT-Engine-MQL provides support for executing queries using different model query languages. Each implementation provides an extension that is able to parse a query in a specific model query language and transform it into a QLI Model. Moreover, it also provides support for processing results and adapt them to the format required by the query. A different *MQT-Engine-MQL* is provided for each supported model query language.

MQT-Engine-PQL provides support for executing queries in different database-based persistence mechanisms. Each implementation of this package provides an extension for a different persistence mechanism. This package is able to generate a persistence-specific query language from a given QLI Model and then execute it directly over the persistence. After execution obtained raw results are processed in order to return adequate results. A different *MQT-Engine-PQL* is provided for each supported persistence-specific query language. Is important to note that each implementation is specific for a concrete persistence mechanism.

5.2 QLI Metamodel

The framework uses *QLI Metamodel* to specify queries using abstractions and in a language-agnostic way. Following sections provide: description of different artifacts existing within the metamodel; sample mapping between different model query languages and artifacts within the QLI Metamodel; and generation of different persistence-specific queries from a QLI Model instance.

A figure that depicts overview of the QLI Metamodel has been included in Appendix A.

5.2.1 QLI Metamodel Artifacts

Next, the different artifacts that compose the QLI Metamodel are described one-by-one. Each artifact is grouped by the type of the specified expressions:

Model-Traversal Query Expressions.

Expressions of this type traverse the entire model and output a collection containing model elements (see Figure 5.4). *ModelTraversalQuery* class implements the *Collection* interface, since all classes that extend it specify a collection of model elements. Classes that extend *ModelTraversalQuery* are described below:

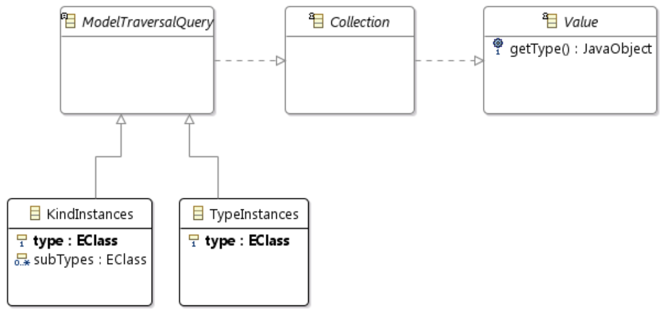


Figure 5.4: QLI Metamodel fragment related to model-traversal query expressions.

- **TypeInstances**. Searches for instances that are of the specified type (*type* attribute).

- **KindInstances**. Searches for instances that are of the specified kind (*type* attribute) and also instances that include the specified kind as a supertype (*subTypes* attribute).

Filtering Query Expressions.

Expressions of this type iterate an input value containing model objects and return a value (that could be a collection, an object or a primitive value) that is obtained after applying filtering operations over the input collection (e.g. find objects satisfying a condition, check if some element satisfies a condition, etc.). Classes that specify this type of query expressions implement the *FilteringQuery* class.

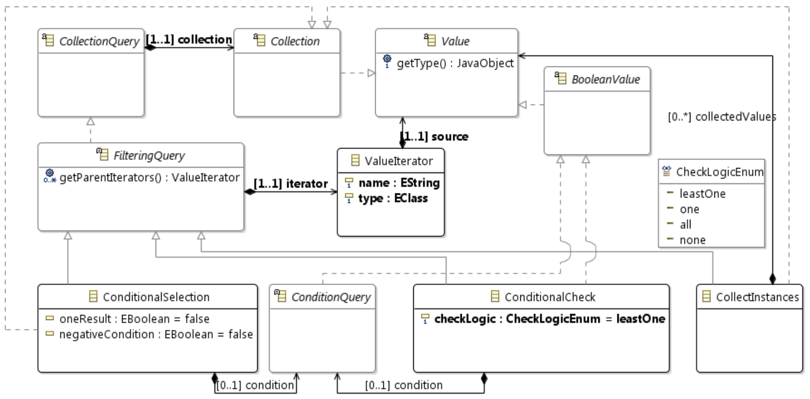


Figure 5.5: QLI Metamodel fragment related to filtering query expressions.

Figure 5.5 illustrates a fragment of the metamodel that is related to the *FilteringQuery* instances. All the instances contain a *ValueIterator* instance (at *iterator* reference) that specifies the element which iterates objects within collections. *ValueIterator* contains two attributes (*name* of the iterator and *type* of the iterated values) and *source* reference pointing to the iterated value.

All classes that extend *FilteringQuery* should implement the *getParentIterators()* method that obtains iterators specified within parent *FilteringQueries*.

FilteringQuery class is extended by the following three classes that are used to specify filtering query expressions:

- **ConditionalSelection**. Specifies query expressions that return a collection including model objects that satisfy a set of conditions (*condition* reference). This class contains two attributes: *oneResult*, if value is true only one value has to be returned; and *negativeCondition*, true value indicates that the selected objects should not satisfy the conditions. *ConditionalSelection* instances contain a *ConditionQuery* instance that specifies the condition to be evaluated. Each *ConditionalSelection* specifies a collection of values, consequently it is an implementation of the *Collection* class.

- **ConditionalCheck**. Returns a boolean value that specifies if the condition has been satisfied. *checkLogic* specifies the logic followed by the condition for checking elements and possible values are specified by *CheckLogicEnum*: *leastOne* (default value), the condition has to be satisfied at least by one of the elements; *one*, the condition has to be satisfied just by one element; *all*, the condition has to be satisfied by all the elements; and *none* the condition has not to be satisfied by any element.

  ConditionalCheck instances contain a *ConditionQuery* instance that specifies the condition to be evaluated. This class implements the *BooleanValue* interface since instances of this type return a boolean value.

- **CollectInstances**. Specifies query expressions that return a collection derived from feature values of the objects within the values contained by the *collectedValues* reference. This class implements the *Collection* interface since instances of this type specify a collection of values.

Query Expressions Specifying Conditions.

Expressions of this type specify conditions that are evaluated by *ConditionalSelection* and *ConditionalCheck* instances. As Figure 5.6 illustrates, this type of query expressions extend the *ConditionQuery* class. Classes that are used to specify different types of conditions are described below:

- **LogicalCondition**. Evaluates logically boolean values returned by one or two query expressions (*ConditionQuery* instances). Attribute *operator* contains a *LogicalOperatorEnum* value which specifies the logical operator (*AND*, *OR*, *NOT*, *XOR*, *IMPLY*). In the case of NOT operator, right attribute contains the single expression to be evaluated. In other cases, *left* and *right* attributes contain the two expressions to be evaluated.

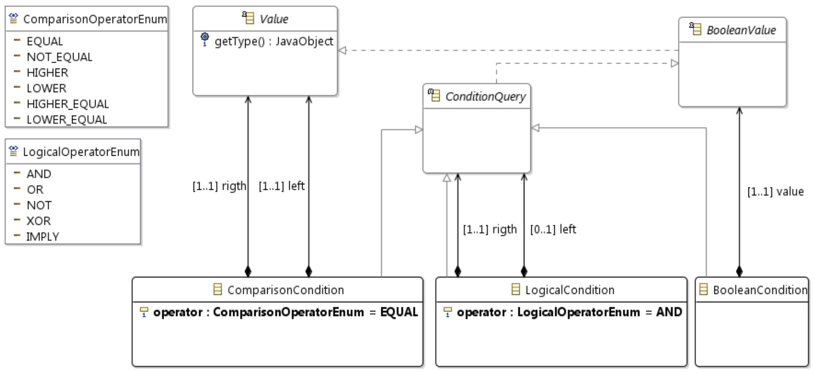


Figure 5.6: QLI Metamodel fragment related to condition query expressions.

- **ComparisonCondition**. Query expressions that compare values returned by sub-query expressions. Both values must be of same type and abstract specification of the evaluated query expressions are contained in the *left* and *right* references of this class. Attribute *operator* specifies comparison operator with a `ComparisonOperatorEnum` value. This enumerator contains literals specifying different types of operators that are used to compare values (*EQUAL*, *NOT_EQUAL*, *HIGHER*, *LOWER*, *HIGHER_EQUAL*, *LOWER_EQUAL*).

- **BooleanCondition**. For query expressions that return a boolean value that indicates if the condition is satisfied. Boolean value is obtained from *value* reference.

Query Expressions returning Values from Collections.

Figure 5.7 illustrates this group that contains classes used to specify query expressions that return a value from an input collection. Classes within this group implement the *CollectionQuery* interface. The queried *Collection* is contained in the *collection* reference.

- **Flatten**. Specifies query expressions that return the flattened input collection. *Flatten* outputs a collection of values and consequently, it implements *Collection*.

- **Size**. Specifies expressions that return size value of the input collection.

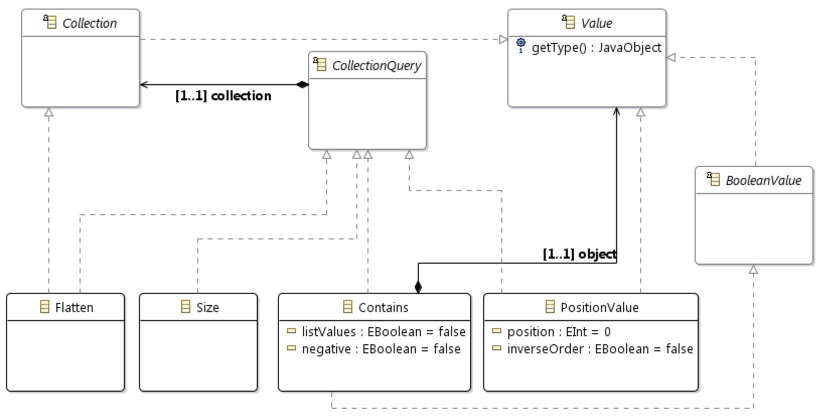


Figure 5.7: QLI Metamodel fragment related to query expressions returning values from collections.

- **Contains**. Specifies query expressions that return a boolean value indicating if the input collection contains one or more input values. The *listValues* attribute is only applicable when the input value specified by the *object* reference is a list. A true value in the *listValues* attribute indicates that the existence of all the elements within the list has to be checked one-by-one. By contrast, false indicates that the existence of the list (as an element) has to be checked. A true value in the *negative* attribute indicates that the query expression checks if the input values are excluded within the collection. And false (default value) indicates that the query expression checks if the input values are included. *Contains* outputs a boolean value, and consequently, it implements *BooleanValue*.

- **PositionValue**. Specifies query expressions that return the value of the input collection that is located on a specific position. The *position* attribute specifies position of the element to be returned, and the *inverseOrder* boolean attribute indicates if the position has to be calculated starting from the last element of the collection.

Query Expressions specifying model Objects.

This group contains classes that are used to specify model objects. These classes implement *ModelObject* as depicted in Figure 5.8:

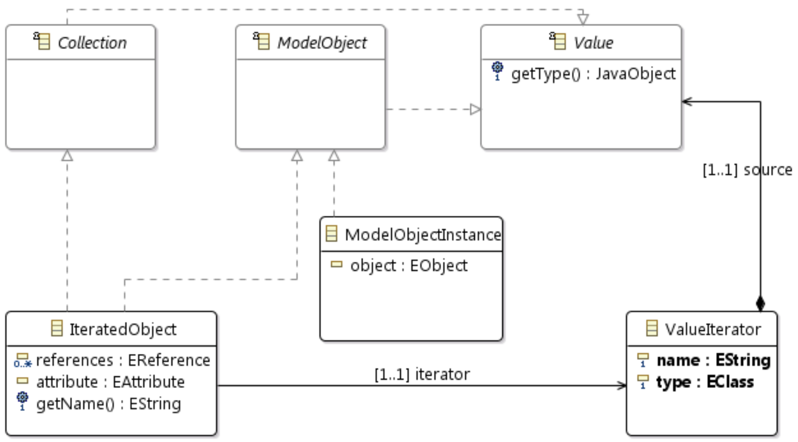


Figure 5.8: QLI Metamodel fragment related to query expressions specifying model objects.

- **ModelObjectInstance**. Specifies an object within the model. The attribute `object` contains the specified object.
- **IteratedObject**. Specifies a model object iterated by a *ValueIterator* instance (*iterator* reference). Each instance contains *attribute* and *references* attributes specifying features that are navigated by the query expression within the iterated object. *getName* operation returns an string containing the iterator name + navigated features. Depending on the navigated features, instances of this type return one or more values, and it implements `Collection`.

Query Expressions returning Values from Objects or Features.

This group contains classes specifying query expressions that return a value from an input object. Figure 5.9 illustrates the metamodel fragment that depicts abstractions of this type. Classes within this group implement *ObjectQuery* and the *object* reference contains the queried *Object*.

- **TypeValue**. Specifies query expressions that return type of a model object.
- **Defined**. Specifies query expressions that check if a model object or a feature value exists. A true value in *negative* attribute indicates that the expression checks that the value does not exist.

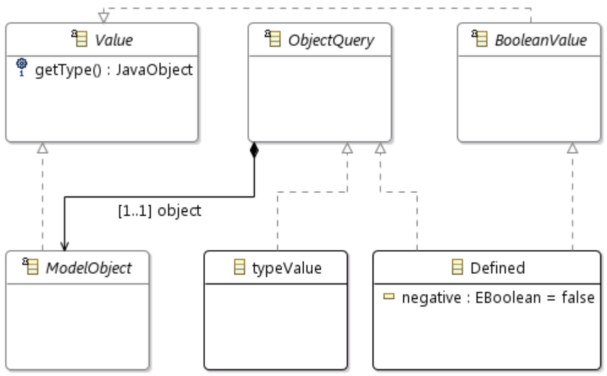


Figure 5.9: QLI Metamodel fragment related to query expressions returning values from objects or features.

Query Expressions specifying primitive values.

PrimitiveValue (see Figure 5.10) class is used to specify primitive values within query expressions. Primitive value is contained in the *value* attribute.

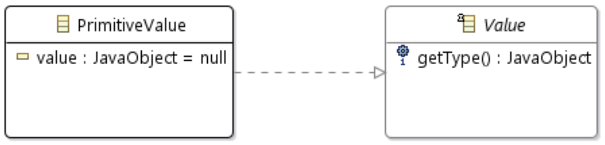


Figure 5.10: QLI Metamodel fragment related to query expressions returning values from objects or features.

5.2.2 Specifying Model Queries with QLI Models

MQT-Engine framework generates QLI Models for queries specified using a model query language. In the following paragraphs we provide a sample mapping between an EOL query and the QLI Model generated by MQT-Engine. Additional mappings between QLI Models and other model query languages have been included in Appendix A.

Listing 5.1 shows the EOL query that searches `Sensor` instances, that are associated with `Switch` instances that are part of a `Route` instance, and `Route` is different of the `Route` associated with the `Sensor` instance. This query is a simplification of the query proposed in the *Train Benchmark Case*[1].

```
1 Route.all
2   .collect(route |
3     route.follows.collect(swP |
4       swP.‘switch‘.collect(sw |
5         sw.sensor.select( sensor:Sensor | route.definedBy.
              excludes(sensor)
6       )
7     )
8   )
9 );
```

Listing 5.1: Query expressed with EOL.

Figure 5.11 illustrates the QLI Model that specifies the EOL query of Listing 5.1. The mapping between EOL expressions and QLI Model artifacts is described below:

- *Line 1*: EOL expression that collects all the `Route` instances within the model. It is specified in the model by *KI1*, a *KindInstances* instance. `Route` type does not have any specialization. Consequently, only the type attribute of *KI1* is setted with *Route EClass*.

- *Line 2*: collects the previously obtained `Route` values at the `route` variable. It is specified by a *CI1*, a *CollectInstances* instance. *CI1* contains a *ValueIterator* (*IT1*) that specifies the `route` variable. This variable contains `Route` instances and hence, it will contain the previously generated *KI1*.

- *Line 3*: `swP` variable collects the values contained by the `follows` reference, and for each route within the `route` variable. Therefore, another *CollectInstances* instance is created (*CI2*). *CI2* contains *IT2*, a *ValueIterator* instance that specifies the values contained by the `swP` variable. *IT2* contains *IO1*, an *IteratedObject* instance that specifies the values of the *follows* reference for routes specified by *IT1*.

[1]For more information refer to Chapter 8

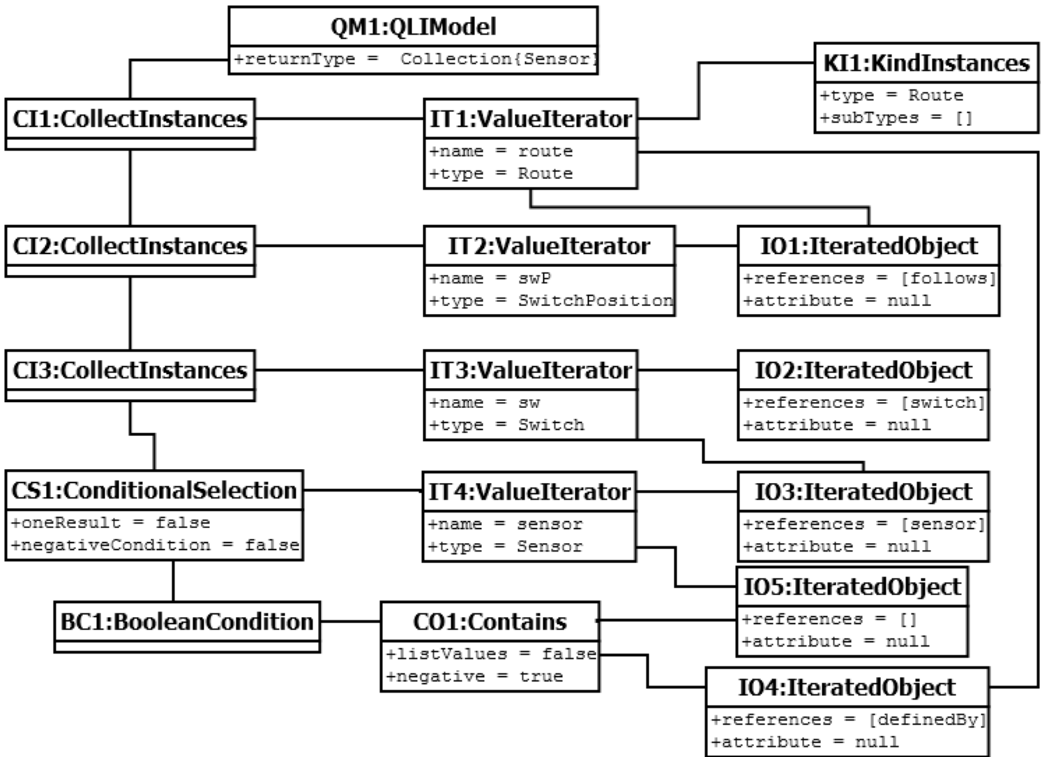


Figure 5.11: QLI Model that specifies the EOL query.

- *Line 4*: `sw` variable collects the values contained by the `switch` reference in the elements specified by the `swP` variable. It is specified by *CI3*, a *CollectInstances* instance. *CI3* contains a *ValueIterator* (*IT3*) for specifying values within the *sw* variable. Therefore, *IT3* contains *IO2* (*IteratedObject* instance), an abstraction for values of the *switch* reference for elements specified by *IT2*.

- *Line 5*: first, collects the `Sensor` instances that are contained by the *sensor* reference for each element collected by the `sw` variable. Then, selects only the elements that are not included in the `definedBy` reference of the routes collected by the `route` variable. These EOL expressions are specified by *CS1*, a *ConditionalSelection* instance, and *oneResult* and *negativeCondition* attribute values are false in both cases. *CS1* contains *IT4* (*ValueIterator* instance), which specifies the values within the `sensor` variable. *IT4* contains *IO3* (*IteratedObject* instance) which is the abstraction for values of the `sensor` reference for elements specified by IT3. Moreover, *CS1* contains *BC1* (`BooleanCondition` instance) which specifies the select condition. *BC1* includes a *Contains* instance (*CO1*) specifying the condition to be checked, and it contains: *IO4 IteratedObject* that specifies values at the `definedBy` reference of each elements collected by route variable (*IT1*); and *IO5*, that specifies the elements collected by `sensor` variable (*IT4*). Two attribute values are setted for *CO1*: *listValues* attribute value is *false*, since the input element that is checked is an object (*sensor* variable at the EOL query); and *negative* attribute value is true, since the EOL query searches for elements that exclude the input object.

5.2.3 Generating Persistence-Specific Queries from QLI Models

MQT-Engine framework transforms a QLI Model into a query in a persistence-specific query language. Following paragraphs describe this transformation through a sample mapping between a QLI Model and a SQL query. The sample model corresponds with the QLI Model of Figure 5.11. Mappings between QLI Models and other persistence-specific query languages have been included in Appendix A.

The generated SQL will be executed in CDO repositories using *DBStore* with the horizontal mapping strategy. Figure 5.12 illustrates a subset of the data-schema used to persist queried model in CDO. *ROUTE*, *SWITCHPOSITION*, *SWITCH* and *SENSOR* tables persist model artifacts (Route, SwitchPositon, Switch or Sensor instances). All these tables contain: an identifier of each element in the database (*CDO_ID*), information related to version and branching (*CDO_VERSION*, *CDO_BRANCH*, *CDO_CREATED* and *CDO_REVISED*) and information related to parent elements and resources (*CDO_RESOURCE*, *CDO_CONTAINER* and *CDO_FEATURE*). Each table contains also feature values: attribute values (e.g. *ID* in *ROUTE* table), single-value reference values (e.g. *ENTRY* or *EXIT0* in *ROUTE* table), and in the case of multi-value references, quantity of referenced elements (e.g. *FOLLOWS* and *DEFINEDBY* in *ROUTE* table). The rest of the tables (*ROUTE_FOLLOWS_LIST*, *ROUTE_DEFINEDBY_LIST* and *SENSOR_ELEMENTS_LIST*) contain the elements referenced by multi-value references. Tables of this type contain: identifier of the source instance (*CDO_SOURCE*), information related to version and branch (*CDO_VERSION* and *CDO_BRANCH*), order of the referenced value (*CDO_IDX*) and the identifier of the referenced element (*CDO_VALUE*).

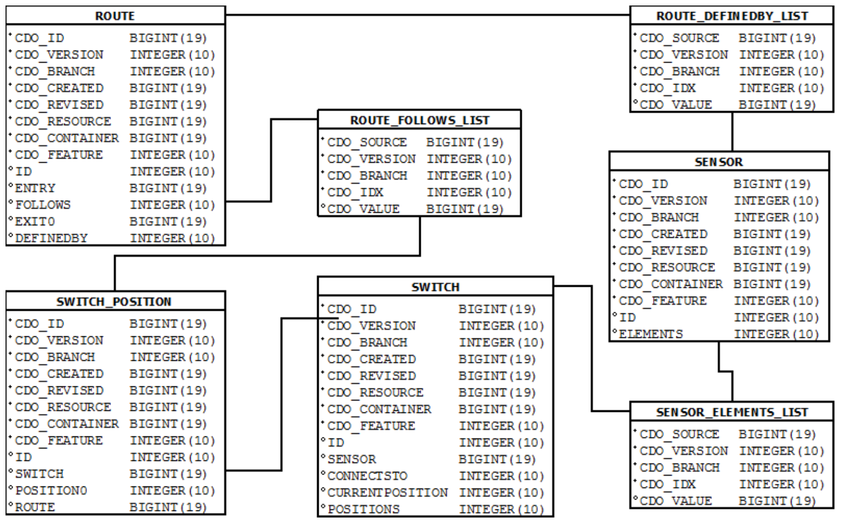


Figure 5.12: Database tables related to the generated SQL query.

Listing 5.2 illustrates the generated SQL query that corresponds with QLI Model of Figure 5.11:

- *Line 1*: *IT4* specifies the elements collected by sensor variable, and sensors are the output of the query. *Name* attribute value of *IT4* is used to complete the select expression of the generated SQL query: `SELECT sensor.CDO_ID`.

- *Line 2*: The `FROM` expression starts collecting all the route instances at *line 2*. This line is generated using *IT1* and *KI1* instances contained by *CI1*: *KI1* is used to generate the SQL that obtains all route instances (`SELECT * FROM Route WHERE ...`); and *IT1* is used to add the route alias to the previous expression (`AS route`).

- *Line 3*: *CI2* is used to generate the SQL fragment of *line 3*. *CI2* includes *IO1* and *IT2* and they are used for generating the SQL expression that obtains *SwitchPosition* values contained by the *follows* reference for each route element obtained at *line 2* of the SQL query. The *follows* feature is a multi-value reference and the `Route_follows_LIST` table is used to obtain the information. Values from this table are joined with the values of the `SwitchPosition` table. The SQL is completed with an alias that corresponds with *swP*, the value of the name attribute in *IT2*. An `INNER JOIN` expression is used to join this SQL with the expression of *line 2*.

```
1 SELECT sensor.CDO_ID
2 FROM (SELECT * FROM Route WHERE CDO_VERSION>0 AND (CDO_BRANCH = 0 AND
    CDO_REVISED=0)) AS route
3 INNER JOIN (SELECT Route_follows_LIST.CDO_SOURCE AS PARENT_ID,
    Route_follows_LIST.CDO_VERSION AS PARENT_VERSION, Route_follows_LIST.
    CDO_BRANCH AS PARENT_BRANCH, route_follows.* FROM Route_follows_LIST
    INNER JOIN (SELECT * FROM SwitchPosition WHERE  CDO_VERSION>0 AND (
    CDO_BRANCH = 0 AND CDO_REVISED=0)) AS route_follows ON
    Route_follows_LIST.CDO_VALUE = route_follows.CDO_ID AND route_follows.
    CDO_REVISED = 0 AND route_follows.CDO_BRANCH = 0) AS swP ON swP.
    PARENT_ID = route.CDO_ID AND swP.PARENT_VERSION = route.CDO_VERSION AND
    swP.PARENT_BRANCH = route.CDO_BRANCH
4 INNER JOIN (SELECT * FROM Switch WHERE  CDO_VERSION>0 AND (CDO_BRANCH = 0
    AND CDO_REVISED=0)) AS sw ON swP.switch=sw.CDO_ID
5 INNER JOIN (SELECT Sensor_elements_LIST.CDO_VALUE, sw_sensor.* FROM (SELECT
    * FROM Sensor WHERE  CDO_VERSION>0 AND  (CDO_BRANCH = 0 AND CDO_REVISED
    =0) ) AS sw_sensor INNER JOIN Sensor_elements_LIST ON sw_sensor.CDO_ID =
     Sensor_elements_LIST.CDO_SOURCE  AND sw_sensor.CDO_BRANCH =
    Sensor_elements_LIST.CDO_BRANCH AND sw_sensor.CDO_VERSION =
    Sensor_elements_LIST.CDO_VERSION ) AS sensor ON sw.CDO_ID=sensor.
    CDO_VALUE
6 WHERE NOT EXISTS (SELECT route_definedBy.CDO_ID FROM Route_definedBy_LIST
    INNER JOIN (SELECT * FROM Sensor WHERE CDO_VERSION>0 AND (CDO_BRANCH = 0
     AND CDO_REVISED=0)) AS route_definedBy ON Route_definedBy_LIST.
    CDO_VALUE = route_definedBy.CDO_ID AND Route_definedBy_LIST.CDO_SOURCE=
    route.CDO_ID AND Route_definedBy_LIST.CDO_BRANCH=route.CDO_BRANCH AND
    Route_definedBy_LIST.CDO_VERSION=route.CDO_VERSION AND route_definedBy.
    CDO_ID=sensor.CDO_ID)
```

Listing 5.2: SQL query for models persisted with CDO+DBStore.

- *Line 4*: *CI3* is used similarly for generating the SQL expressions depicted in *line 4*. *IO2* and *IT3* instances contained by *CI3* are used to generate the SQL expression. It obtains *Switch* instances contained by *switch* reference for values collected by `swP` at *line 3*. The *switch* feature is a single-value reference and this expression joins `Switch` table values with the switch column values of `swP` alias (`swP.switch`).

- *Line 5*: it is generated from the *CS1* instance. *Line 5* corresponds with the iterated values, and *IT4* and *IO3* are used to generate it. *IO3* specifies values of the sensor reference for elements collected by `sw` query alias at *line 4*. These values are obtained in a SQL expression that joins `Sensor` and `Sensor_elements_List` tables. The SQL expression is completed with the `sensor` alias (using *IT4*) and it is joined with *line 4*.

- *Line 6*: it is generated from the *CS1* instance, and concretely from the *CO1* instance contained by it. *CO1* specifies the condition of

CS1 and consequently, this expression completes the `WHERE` statement of the SQL query. *CO1* contains *IO4* and *IO5* instances. First, *IO4* is used to generate a SQL expression that joins values of `Route_definedBy_LIST` and `Sensor` tables. Then the expression is completed with `route_definedBy_LIST.CDO_ID = sensor.CDO_ID`, checking if the sensor specified by *IO5* is contained. The previously generated SQL expression is contained within by a `NOT EXISTS()` expression, since the *CO1* has the *negative* attribute with true value.

5.3 MQT-Engine Design

MQT-Engine Framework is divided into three main packages: *MQT-Engine-Core* is the query language independent part that orchestrates the query transformation. It also provides a set of abstract classes to be extended by the other parts that are specific of the query languages and persistences; *MQT-Engine-MQL* package provides implementations for model query languages; and *MQT-Engine-PQL* package provides implementations for persistence-specific query languages.

This design facilitates the inclusion of new query languages in MQT-Engine. Figure 5.13 depicts a class diagram where different parts and artifacts of the framework are illustrated.

MQT-Engine-Core package

MQT-Engine-Core is the language-agnostic package of the framework. It contains *Engine-Core*, the main class which orchestrates the query transformation and execution. This package also includes classes that implement the QLI Metamodel and abstract classes to be extended by the query language-specific parts.

Engine-Core class is the main class of the framework. Each *Engine-Core* instance contains one *MQLDriver* implementation (*mqlDriver* feature) that supports executing queries of an specific model query language. Similarly, *pqlDriver* feature contains a *PQLDriver* implementation that supports transforming and executing queries for a specific persistence. *Engine-Core* instances reference *DomainEPackage* instances (*domainMetamodels* feature).

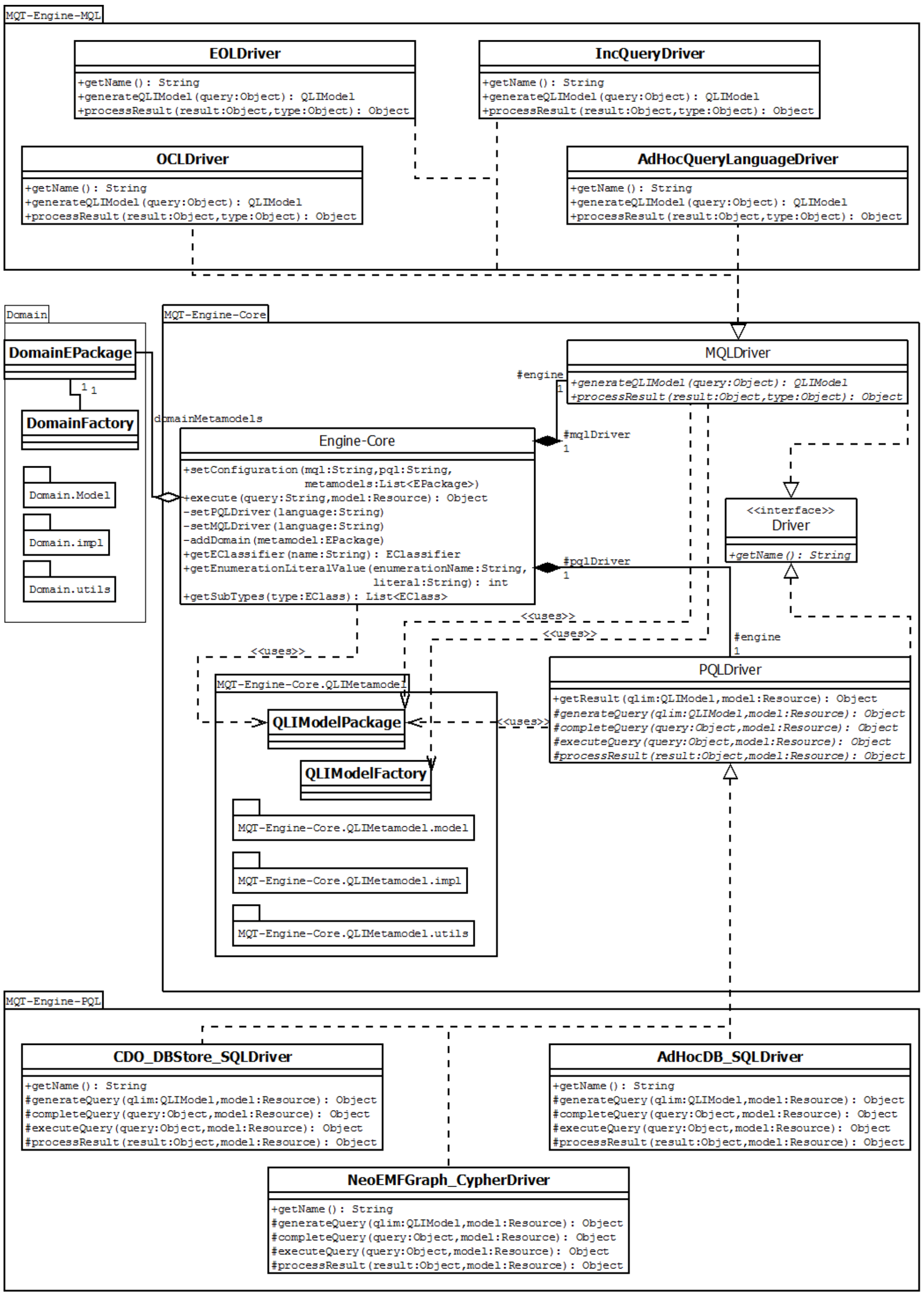

MQT-Engine-MQL
EOLDriver
+getName(): String
+generateQLIModel(query:Object): QLIModel
+processResult(result:Object,type:Object): Object
IncQueryDriver
+getName(): String
+generateQLIModel(query:Object): QLIModel
+processResult(result:Object,type:Object): Object
OCLDriver
+getName(): String
+generateQLIModel(query:Object): QLIModel
+processResult(result:Object,type:Object): Object
AdHocQueryLanguageDriver
+getName(): String
+generateQLIModel(query:Object): QLIModel
+processResult(result:Object,type:Object): Object
Domain
DomainEPackage
1
1
DomainFactory
Domain.Model
Domain.impl
Domain.utils
MQT-Engine-Core
domainMetamodels
MQLDriver
#engine
1
+generateQLIModel(query:Object): QLIModel
+processResult(result:Object,type:Object): Object
Engine-Core
+setConfiguration(mql:String,pql:String,
metamodels:List<EPackage>)
+execute(query:String,model:Resource): Object
-setPQLDriver(language:String)
-setMQLDriver(language:String)
-addDomain(metamodel:EPackage)
+getEClassifier(name:String): EClassifier
+getEnumerationLiteralValue(enumerationName:String,
literal:String): int
+getSubTypes(type:EClass): List<EClass>
#mqlDriver
1
#pqlDriver
1
<<interface>>
Driver
+getName(): String
<<uses>>
<<uses>>
<<uses>>
<<uses>>
#engine
1
PQLDriver
+getResult(qlim:QLIModel,model:Resource): Object
#generateQuery(qlim:QLIModel,model:Resource): Object
#completeQuery(query:Object,model:Resource): Object
#executeQuery(query:Object,model:Resource): Object
#processResult(result:Object,model:Resource): Object
MQT-Engine-Core.QLIMetamodel
QLIModelPackage
QLIModelFactory
MQT-Engine-Core.QLIMetamodel.model
MQT-Engine-Core.QLIMetamodel.impl
MQT-Engine-Core.QLIMetamodel.utils
MQT-Engine-PQL
CDO_DBStore_SQLDriver
+getName(): String
#generateQuery(qlim:QLIModel,model:Resource): Object
#completeQuery(query:Object,model:Resource): Object
#executeQuery(query:Object,model:Resource): Object
#processResult(result:Object,model:Resource): Object
AdHocDB_SQLDriver
+getName(): String
#generateQuery(qlim:QLIModel,model:Resource): Object
#completeQuery(query:Object,model:Resource): Object
#executeQuery(query:Object,model:Resource): Object
#processResult(result:Object,model:Resource): Object
NeoEMFGraph_CypherDriver
+getName(): String
#generateQuery(qlim:QLIModel,model:Resource): Object
#completeQuery(query:Object,model:Resource): Object
#executeQuery(query:Object,model:Resource): Object
#processResult(result:Object,model:Resource): Object

Figure 5.13: MQT-Engine framework class diagram.

DomainEPackage is part of the *Domain* package that contains a set of classes that implement a domain-specific metamodel.

Engine-Core implements *setConfiguration* method. This method configures *Engine-Core* instances specifying: name of the model query language that will be used (*mql*); name of the persistence-specific query language that will be used (*pql*); a list of *EPackage* implementations where the domain metamodels conformed by queried models are implemented (*metamodels*). It calls *setMQLDriver*, *setPQLDriver* and *addDomain* methods that are also implemented by *Engine-Core*: *setMQLDriver* initializes *mqlDriver* with the configured model query language; *setPQLDriver* intializes *pqlDriver* with the configured persistence-specific query language; and *addDomain* adds a domain metamodel in the *domainMetamodels* feature.

The query transformation and execution is orchestrated within the execute method that is implemented by *Engine-Core*. This method performs different steps:

- calls the model query language driver for transforming the query expressed with a model query language into a *QLIModel* instance.
- obtains the generated model, and calls the persistence-specific driver for the execution of the following tasks: transform the *QLIModel* into a persistence-specific query; complete the generated query; execute it over the database; and process obtained raw results;
- the result is obtained, and it calls again model query language driver for processing it.

Engine-Core class implements a set of methods that provide domain-specific information using the domain metamodels referenced at *domainMetamodels* feature (*getEClassifier*, *getEnumerationLiteralValue* and *getSubtypes*). These methods are used during the transformation process by the *MQLDriver* and *PQLDriver* implementations: *getEClassifier* inputs a name and searches for the corresponding *EClassifier* instance within the metamodels; *getEnumerationLiteralValue* inputs an enumeration name and a literal name and searches the corresponding enumeration literal value within the metamodels; and *getSubtypes* operation inputs an `EClass` instance and returns a list containing all the `EClass` instances that extend it (sub-types).

***Driver* interface.** This interface is implemented by *MQLDriver* and *PQLDriver* classes. It contains the *getName* abstract method to be implemented by the classes that extend *MQLDriver* or *PQLDriver*. This method should return name of the model query language (for *MQLDriver* implementations) or persistence (for *PQLDriver* implementations).

***MQLDriver* abstract class.** This class is extended by classes that add support for model query languages. Each *MQLDriver* instance is responsible for parsing and transforming queries from a specific model query language into a *QLI Model* instance. This task is performed by the *generateQLIModel* method. The *generateQLIModel* implementation must include the parsing of the query specified in the model query language and the transformation of each query to a *QLI Model*.

Each model query language has its own data-types. In the cases where queries return or expect results in a data-type that is specific of a model query language, the *MQLDriver* instances are responsible for processing the results obtained from the persistence and format them into the expected data type. This task is performed by the *processResults* method. This method is abstract and it has to be implemented by model query language specific specialization of the *MQLDriver*.

***PQLDriver* abstract class.** This abstract class is extended by classes that add support for persistences and persistence-specific queries. *PQLDriver* implements the `getResult` method. It inputs a *QLI Model* (*qlim*) and a model to be queried (*model*), and outputs the result of the query.

Listing 5.3 illustrates the implementation, and as the code shows `generateQuery`, `completeQuery`, `executeQuery` and `processResult` methods are called. These methods are abstract and they are implemented by each `PQLDriver` specialization.

MQT-Engine-Core.QLIMetamodel package. This package contains classes that implement the *QLI Metamodel* conformed by *QLI Models* that are used by *MQLDriver* and *PQLDriver* specializations for query transformation. *QLIMetamodelEPackage* class is one of the main classes of the package and it is used to load, navigate and edit *QLI Models*. This class is used by *MQLDriver* instances during the QLI Model generation and by the *PQLDriver* instances for parsing the *QLI Model*. *QLIMetamodelFactory* is the class that is used by the

```
public Object getResult(QLIModel qliModel, Resource model){
  Object generatedQuery = this.generateQuery(qliModel,model);
  Object completedQuery = this.completeQuery(model,
      generatedQuery);
  Object result = this.executeQuery(completedQuery, model);
  Object processedResult = this.processResult(result, model);
  return processedResult;
}
```

Listing 5.3: Implementation of getResult method.

MQLDriver for creating new *QLI Model* artifacts. This package is composed by other sub-packages that contain the implementation of the different *QLI Model* artifacts that have been previously described in Section 5.2.

MQT-Engine-MQL package

This package contains specializations of the *MQLDriver* class. For example: *EOLDriver* for EOL queries; *OCLDriver* for OCL queries; *IncQueryDriver* for queries that have been written using *IncQuery*; or *AdHocQueryLanguageDriver* for queries in an ad-hoc query language.

Each specialization implements the abstract methods of the *MQLDriver*: *generateQLIM* and *processResult*. The implementation of these methods is specific for the model query language supported by each specialization. In the case of the implementation of *generateQuery*, it must support interaction with the model query language engine. Next, the method uses the parsed query for generating a *QLI Model* that specifies the query with language-independent abstractions. In the case of the *processResult*, if the query expects a data-type that is specific of the model query language, the method implementation provides the feature of re-factoring the input result into an output result that fully matches with the data-type expected by the query.

MQT-Engine-PQL package

This package contains the specializations of *PQLDriver*. For example: *CDO_DBStore_SQLDriver*, for persistence based on CDO with *DBStore*; *NeoEMFGraph_CypherDriver* for NeoEMFGraph persistence; or *AdhocDB_SQLDriver*, for an ad-hoc persistence.

Each specialization is specific for: (i) a persistence mechanism; and (ii) a persistence-specific query language supported by the persistence. For example: *CDO_DBStore_SQLDriver* supports generation and execution of SQL queries over relational databases that are generated using the default configuration of the DBStore store[2]; *NeoEMFGraph_CypherDriver* supports querying models persisted with NeoEMFGraph (which uses a Neo4J database) using Cypher, a Neo4J specific query language; and *AdhocDB_SQLDriver* supports transforming queries to SQL and execute them in an ad-hoc persistence that uses relational databases with metamodel-agnostic data-schemas.

Therefore, each specialization provides a persistence-specific and persistence-query language specific implementation of the *generateQuery*, *completeQuery*, *executeQuery* and *processResult* methods. The *generateQuery* method implementation supports transformation from a QLI Model into a persistence-specific query language supported by the persistence. The *completeQuery* method implementation provides support for completing the generated query with additional information that is obtained from the model that will be queried (e.g. information about the version of the model). The completed information is different depending on the supported persistence and persistence-specific query language. The implemented *executeQuery* method supports the execution of the generated query using a querying mechanism that is provided by the persistence approach. Then, obtained results are processed by the *processResult* method. It processes raw results obtained directly from the persistence and returns only results that are required by the query.

5.4 MQT-Engine Execution

Figure 5.14 illustrates a sequence diagram that depicts the execution of MQT-Engine. First part illustrates the configuration process, where model engineer executes *setConfiguration*. Then, *Engine-Core* instance invokes *setMQLDriver* which creates an instance of the *MQLDriver* specialization corresponding with the input model query language. The instance is stored in the *mqlDriver* variable of *Engine-Core*. Similarly the *setPQLDriver* is invoked and it returns a *PQLDriver* instance that corresponds with the selected persistence-specific query language (and persistence). It is stored in the *pqlDriver* variable of the *Engine-Core*. Configuration process ends with the *addDomain* method execution, where the

[2]for more information about CDO and DBStore please go to Chapter 2.

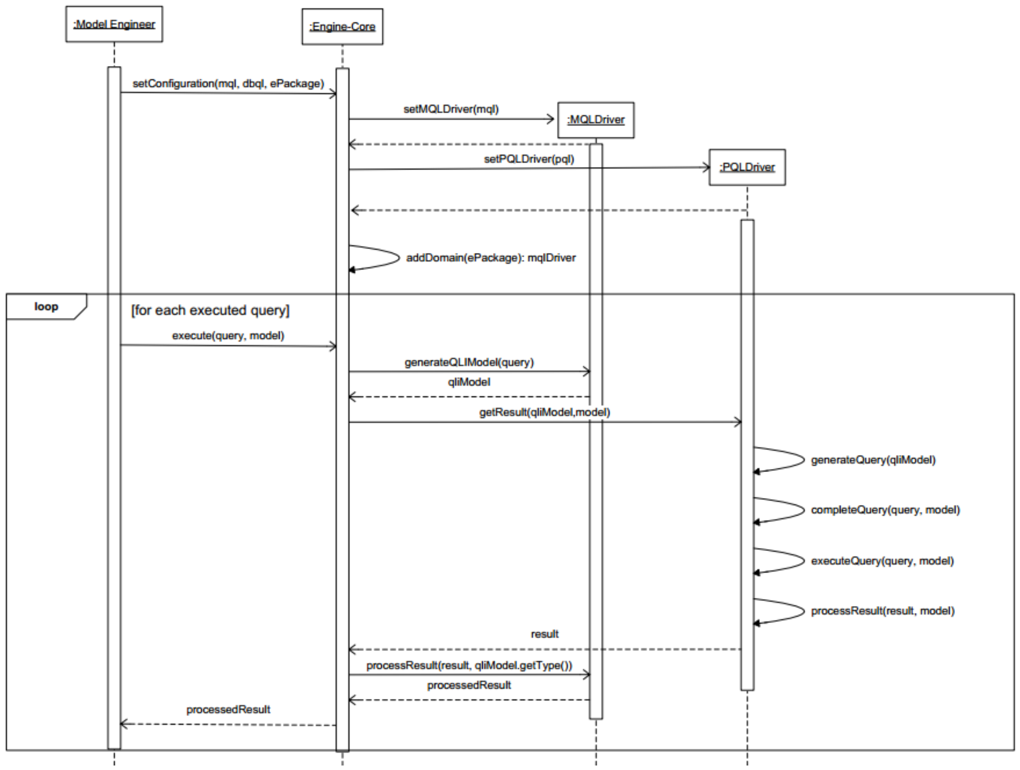

:Model Engineer
:Engine-Core
setConfiguration(mql, dbql, ePackage)
setMQLDriver(mql)
:MQLDriver
setPQLDriver(pql)
:PQLDriver
addDomain(ePackage): mqlDriver
loop
[for each executed query]
execute(query, model)
generateQLIModel(query)
qliModel
getResult(qliModel,model)
generateQuery(qliModel)
completeQuery(query, model)
executeQuery(query, model)
processResult(result, model)
result
processResult(result, qliModel.getType())
processedResult
processedResult

Figure 5.14: Sequence diagram of the query transformation and execution.

EPackages specified by the model engineer are stored in the *Engine-Cores*' *domainMetamodels* variable.

MQT-Engine is ready to execute queries when the configuration step is completed. The query execution is started by the *model engineer* when it calls the *execute* method of the *Engine-Core* instance. At this point, first, the *Engine-Core* instance executes *generateQLIModel* method of the `MQLDriver` instance. In this method, first the input query expressed with a model query language is parsed and then transformed into a *QLI Model*. The *QLI Model* is returned to the *Engine-Core* instance and then it executes the *getResult* method of the *PQLDriver* instance.

getResult executes a set of methods that obtain the result directly from the corresponding persistence: (1) *generateQuery* generates a query expressed with the persistence-specific query language and from the previously generated QLI Model; (2) the generated query is completed by the *completeQuery* method adding information that is obtained from the queried resource; (3) *executeQuery* method executes query directly over the persistence and using a query mechanism provided by the corresponding persistence; (4) the obtained raw results are processed by *processResult* of the *PQLDriver* instance; and (5) results are returned to the *Engine-Core* instance.

Finally, the *Engine-Core* instance executes the `processResult` method of the `MQLDriver` to adapt the query results to the data-types supported by the model query language.

6 Implementation of MQT-Engine Framework

MQT-Engine Framework has been implemented in a prototype that supports executing EOL queries over models that are persisted with CDO repositories. The CDO repositories are configured with the DBStore store. This prototype uses the horizontal mapping strategy, default model-database mapping strategy provided by DBStore.

The MQT-Engine implementation inputs model traversal queries that are expressed using EOL and automatically transforms them into SQL. Generated SQL queries are specific for the tables of the data-schema generated by the DBStore, and they are executed at server-side over CDO repositories .

The prototype implements the `EOLDriver` to support queries expressed with EOL. The `EOLDriver` is a specialization of the `MQLDriver` abstract class provided by MQT-Engine Framework. The `CDO_DBStore_SQLDriver` is implemented to support querying at server-side with SQL models that are persisted into CDO Repositories formed by relational databases. `CDO_DBStore_SQLDriver` is a specialization of `PQLDriver` provided by MQT-Engine Framework.

6.1 Support for EOL Model Query Language

`EOLDriver` adds support for executing queries expressed with EOL in the MQT-Engine framework. `EOLDriver` is responsible for: parsing and transforming EOL queries into a `QLI Model` instances; and processing results obtained from the persistence, adapting them to the data-types supported by EOL. Figure 6.1 illustrates the classes related to the *EOLDriver*.

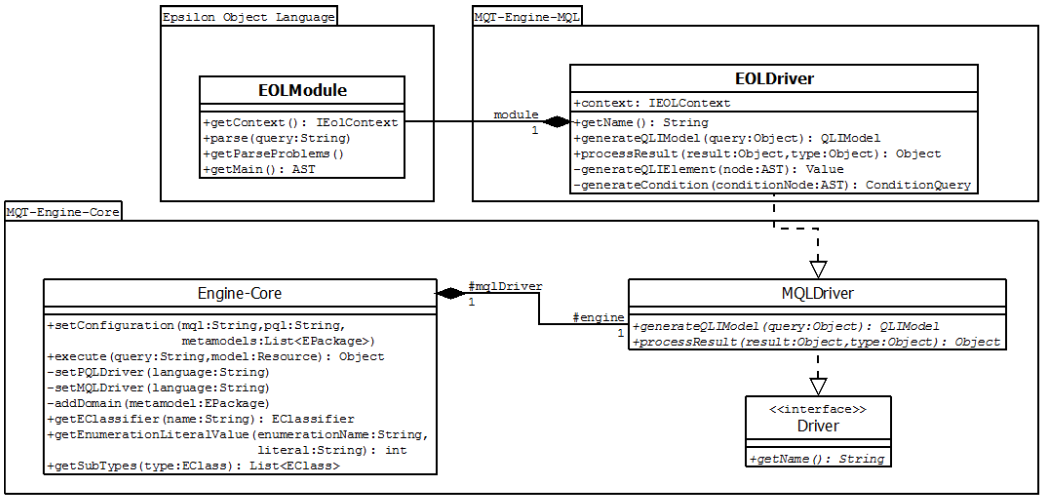

Epsilon Object Language
EOLModule
+getContext(): IEolContext
+parse(query:String)
+getParseProblems()
+getMain(): AST
MQT-Engine-MQL
EOLDriver
+context: IEOLContext
+getName(): String
+generateQLIModel(query:Object): QLIModel
+processResult(result:Object,type:Object): Object
-generateQLIElement(node:AST): Value
-generateCondition(conditionNode:AST): ConditionQuery
module
1
MQT-Engine-Core
Engine-Core
+setConfiguration(mql:String,pql:String, metamodels:List<EPackage>)
+execute(query:String,model:Resource): Object
-setPQLDriver(language:String)
-setMQLDriver(language:String)
-addDomain(metamodel:EPackage)
+getEClassifier(name:String): EClassifier
+getEnumerationLiteralValue(enumerationName:String, literal:String): int
+getSubTypes(type:EClass): List<EClass>
#mqlDriver
1
#engine
1
MQLDriver
+generateQLIModel(query:Object): QLIModel
+processResult(result:Object,type:Object): Object
<<interface>>
Driver
+getName(): String

Figure 6.1: EOLDriver class diagram.

EOLDriver is able to interact with the EOL queries through the `EOLModule`. `EOLModule` provides a set of methods: *getContext* returns an EOLContext that is useful for setting and getting variables that are used in the EOL query; *parse* method is responsible for parsing a String that contains EOL query; *getParseProblems* indicates if any problem has been appeared during the query parsing; and *getMain* returns the AST node that corresponds with the root EOL expression of the query. These methods are used by the *EOLDriver* during the query transformation process.

EOLDriver is a specialization, of the *MQLDriver* class and it implements the *getName*, *generateQLIModel* and *processResult*: *getName* method is used to identify the model query language supported by the *MQLDriver* specialization, which in this case returns the 'EOL' string value; *generateQLIModel* is responsible for generating the QLI Model from EOL queries; and *processResult* method is responsible for adapting the obtained results to data-types supported by EOL.

Two additional private methods are implemented by the *EOLDriver*: *generateQLIElement* and *generateCondition*. The first one, *generateQLIElement*, is used by *generateQLIModel* for the creation of the corresponding QLI Model artifacts. Regarding *generateCondition* method, it is used by the *generateQLIElement* for the creation of QLI Model artifacts that specify conditions.

The previously presented *generateQLIModel* and `processResult` methods are key artifacts in the query transformation and execution process, and following paragraphs provide a lower-level description about the implementation and execution process of these methods:

6.1.1 EOL to QLI Model Transformation: generateQLIModel

generateQLIModel implementation of the EOLDriver transforms an input EOL query into a language-agnostic QLI Model. Figure 6.2 depicts the activity diagram that illustrates the execution process, and it is described below:

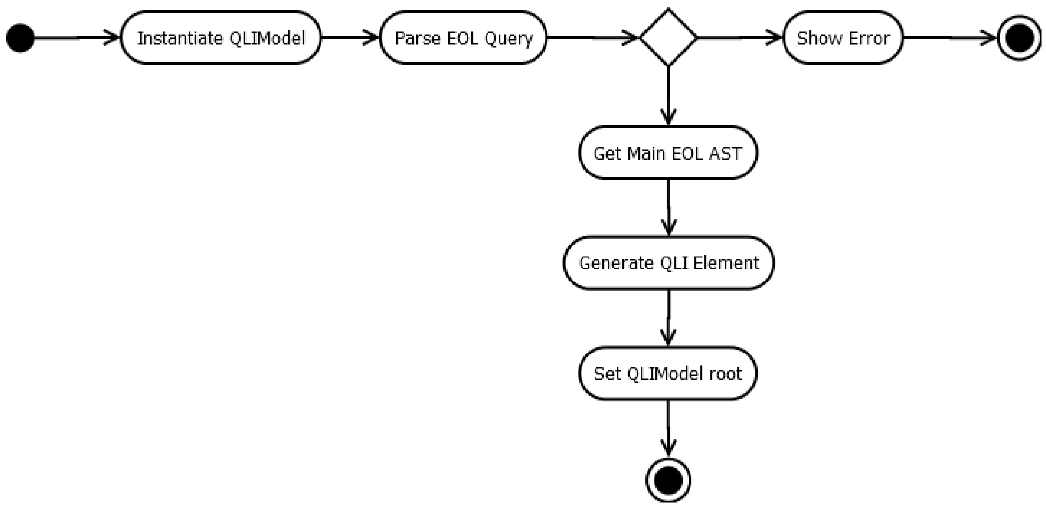


Figure 6.2: Activity diagram of generateQLIModel.

- *Instantiate QLIModel.* The first step is the instantiation of a new QLI Model. This QLIModel will contain the root artifact (*Value* instance) that specifies the EOL query.

- *Parse EOL Query.* Next, the input EOL query is parsed: an EOL Module instance is created, and then *parse*, *getParseProblems* and *getMain* methods are used to parse the query.
- *Show Error.* If the parsing process produces any error the execution is stopped and an error is shown to the model engineer.
- *Get main EOL AST.* If the parsing is correct, an AST node that corresponds with the root expression of the EOL query is obtained.
- *Generate QLI Element.* The AST node is the input of the *generateQLIElement* method which is responsible for creating QLIModel artifacts that correspond with the input query. The `generateQLIElement` method is called recursively until all nodes are visited.
- *Set QLIModel root.* The root QLI Model artifact returned by *generateQLIElement* is setted as the root of the generated QLI Model, and the model is returned to the *Engine-Core* instance, which orchestrates the query transformation and execution.

The transformation algorithm that orchestrates mapping between EOL and QLI Metamodel is implemented within the `generateQLIElement` method. Table 6.1 resumes mapping between QLI Metamodel and the EOL query expressions supported by the *EOLDriver* implementation, and they are described below:

- EOL model traversal queries:
  - *type.allOfType()*: returns a collection containing all the model elements that are instances of the type. Each expression of this type is transformed into a `TypeInstances`. This instance will contain the `EClass` instance corresponding with the type in the `type` attribute.
  - *type.allOfKind(),type.allInstances(), type.all()*: returns a collection containing all the model elements that are instances either of the type itself or of one of its subtypes. All these expressions generate a `KindInstances` instance within the `QLI Model`. Each instance will contain `EClass` instance corresponding with the type in the `type` attribute, and `superTypes` attribute contains `EClasses` that specify all the types that extend the input type. `DomainUtil` provides an operation that returns all supertypes for a given type and it is used by the `EOLDriver` for this purpose.

Table 6.1: Generated QLI Elements for EOL expressions.

EOL Expression	**QLI Metamodel Element**
EOL model traversal queries	
allOfKind(), allInstances(), all()	KindInstances
allOfType()	TypeInstances
EOL Filtering queries	
select(), selectOne(), reject()	ConditionalSelection
one(), exists(), forAll()	ConditionalCheck
collect()	CollectInstances
EOL Expressions returning Values from Collections	
at(), first(), last()	PositionValue
size()	Size
includes(), excludes(), includesAll(), excludesAll()	Contains
flatten()	Flatten
EOL Expressions returning Values from Objects	
.reference, .attribute	IteratedObject
type()	TypeValue
isDefined(), isUndefined()	Defined

- EOL filtering queries:

  – *collection.select(iterator:Type | condition)*: returns all the elements of the input collection that satisfy the condition. A `ConditionalSelection` instance is created per each EOL expression of this type. The expression returns all the elements that satisfy the condition. Consequently, `oneResult` and `negativeCondition` attributes are `false`. `condition` reference will contain a `ConditionQuery` instance that corresponds with the condition of the select expression.

  – *collection.selectOne(iterator:Type | condition)*: returns first element of the input collection that satisfies the condition. The algorithm creates one `ConditionalSelection` instance per each EOL expression of this type. `oneResult` attribute is `true` and `negativeCondition` is `false`. `condition` reference contains a `ConditionQuery` instance that corresponds with the condition.

- *collection.reject(iterator:Type | condition)*: returns all the elements of the input collection that do not satisfy the condition. EOL expressions of this type are transformed into `ConditionalSelection` instances. `oneResult` attribute value is `false`, `negativeCondition` is `true` (only elements that do not satisfy the condition will be selected) and a `ConditionQuery` instance that corresponds with the condition is contained by the `condition` reference.
- *collection.one(iterator:Type | condition)*: returns a boolean value that indicates if the collection contains just one value satisfying the condition. The algorithm generates one `ConditionalCheck` instance per each EOL expression of this type. In this case, the `checkLogic` attribute value is `one`.
- *collection.exists(iterator:Type | condition)*: returns a boolean value that indicates if at least one element within the collection satisfies the condition. The algorithm generates one `ConditionalCheck` instance per each EOL expression of this type. Value for `checkLogic` is the default one (`leastOne`).
- *collection.forAll(iterator:Type | condition)*: returns a boolean value that indicates if all the elements within the collection satisfy the condition. EOL expressions of this type are transformed into `ConditionalCheck` instances with the `checkLogic` attribute setted with `all`.
- *collection.collect(iterator:Type | expression)*: returns a collection containing values of the specified expression for each element of the input collection. Expressions of this type are transformed into a `CollectInstances` instance which contains a `Value` instance that corresponds with the collect expression.

• EOL expressions returning values from Collections:

- *collection.at(position)*: returns the element of the input collection that is located at a specific position. Expressions of this type are specified through `PositionValue` instances. Value of `position` attribute is the specified position and `inverseOrder` false.
- *collection.first()*: returns the first element of the input collection. Expressions of this type are specified through `PositionValue` instances. Value of `position` attribute is 1 and `inverseOrder` false.

  - *collection.last()*: returns the last element of the input collection. Expressions of this type are specified through `PositionValue` instances. Value of `position` attribute is 1 and `inverseOrder` true.
  - *collection.size()*: returns how many elements are contained in the collection. One `Size` instance will be instantiated per each expression of this type, and it contains the input collection (`collection` reference).
  - *collection.includes(object)*: returns a boolean value that indicates if the object is included in the input collection. `Contains` instance is used to specify expressions of this type. The `listValues` and `negative` attribute values are `false` in both cases.
  - *collection.excludes(object)*: returns a boolean value that indicates if the object is not included in the input collection. These expressions are specified by a `Contains` instance which has `listValues` attribute with `false` value and `negative` attribute with `true` value.
  - *collection.includesAll(objectList)*: returns a boolean value that indicates if all the objects are included in the input collection. These expressions are specified by a `Contains` instance which has `listValues` attribute with true value and `negative` attribute with false value.
  - *collection.excludesAll(objectList)*: returns a boolean value that indicates if all the objects are not included in the input collection. `Contains` instance is used to specify expressions of this type. The `listValues` and `negative` attribute values are true in both cases.
  - *collection.flatten()*: returns a flattened collection that contains values of the input collection. This type of expressions are specified through the `Flatten` instance. Each instance will contain the input collection in the `collection` reference.

- EOL expressions returning values from Objects:

  - *object.reference*. Returns the value of the specified reference of the input element. The input object of the expression is previously specified by an `IteratedObject` instance. Expressions of this type add an `EReference` instance that corresponds with the reference within the `references` attribute of the `IteratedObject` instance.

  - *object.attribute*. Returns the value of the specified attribute of the input element. Similar to the previous one, but it adds a `EAtribute` instance in the attribute reference.
  - *object.type()*. Returns type of the input object. The algorithm generates a `TypeValue` instance that contains the input object (`ModelObject` instance).
  - *object.isDefined*. Returns a boolean value that indicates if the input object or feature is defined. This type of expressions are specified through a `Defined` instance which has the `negative` attribute with false value.
  - *object.isUnDefined*. Returns a boolean value that indicates if the input object or feature is not defined. This type of expressions are specified through a `Defined` instance which has the `negative` attribute with true value.

- Primitive values existing within EOL queries are specified with `PrimitiveValue` instances. Each instance of this type will contain the primitive value within the `value` attribute.

Figure 6.3 illustrates activity diagram of the algorithm followed by the *generateQLIElement*. Execution starts checking if the AST node specifies an EOL traversal expression. If it is, and the expression searches only types, `TypeInstances` is instantiated and returned. If it also searches sub-types a `KindInstances` is created and returned.

For not EOL traversal expression, the algorithm checks if the AST contains childs (EOL subqueries). If there is not any children a set of checks are performed over the AST node: if it specifies a primitive value or a enumerator value, a `PrimitiveValue` instance is created; if it specifies a model object, a new `IteratedObject` instance is generated; and if the previous conditions are not satisfied a exception is thrown.

If the AST node has at least one child, the process continues getting the first child AST node (which corresponds with left EOL subquery). Child node is the input of a recursive execution of the *generateQLIElement* method, and the transformation algorithm is re-executed again for it, getting as output the QLI Model artifact that corresponds with the EOL subquery. Next, the second child (right EOL subquery) is obtained, and checked:

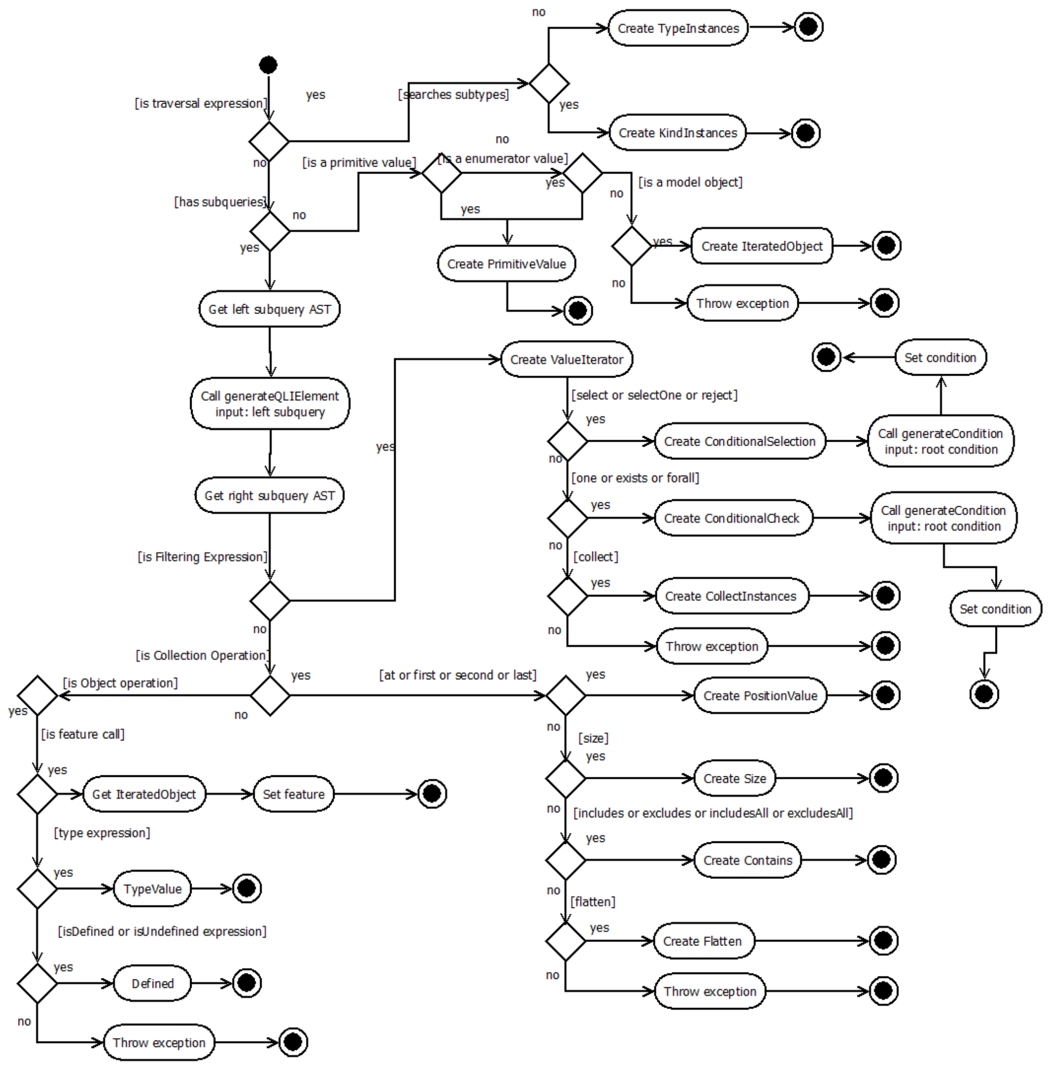


Figure 6.3: Activity diagram of the QLI Model artifact generation.

(a) If the second child specifies an EOL filtering expression, first a `ValueIterator` instance is created, and then, the algorithm performs different actions depending on the specified filtering expression:

- *select*, *selectOne* or *reject*. A `ConditionalSelection` instance is created and the AST node corresponding with the EOL condition is obtained. Condition AST node is the input of the *generateCondition* method, which outputs a `ConditionQuery` (abstraction for the EOL condition). Next, `ConditionQuery` instance is setted in the `condition` feature of the previously created `ConditionalSelection` instance.
- *one*, *exists* or *forall*. A `ConditionalSelection` instance is created, and the *generateCondition* method is executed, obtaining the `ConditionQuery` instance.
- *collect*. A `CollectInstances` is created and returned.
- If the filtering expression does not correspond with any of the previous types, the execution is finished and an exception is thrown.

(b) If the second child specifies an EOL operation over a collection, the algorithm performs different actions depending on the specified expression:

- *at*, *first*, *second*, *last*: A `PositionValue` instance is created.
- *size*: A `Size` instance is created.
- *includes*, *excludes*, *includesAll* or *excludesAll*: A Contains instance is created.
- *flatten*: A `Flatten` instance is returned.
- If the filtering expression does not correspond with any of the previous types, the execution is finished and an error is shown.

(c) If the second child specifies an EOL expression that operates a model object, the algorithm checks the type:

- if the EOL expression is a feature call: first the `IteratedObject` instance is obtained; and then the instance is completed with the feature to be navigated.
- if is a type expression, a `TypeValue` instance is created.

- if it is an *isDefined* or *isUndefined* EOL expression, a `Defined` instance is created.
- If the query does not correspond with any of the previous types, the execution is finished and an exception is thrown.

Some of the previous steps require the execution of the *generateCondition* method for transforming EOL expressions which specify conditions. Figure 6.4 illustrates the activity diagram of this method.

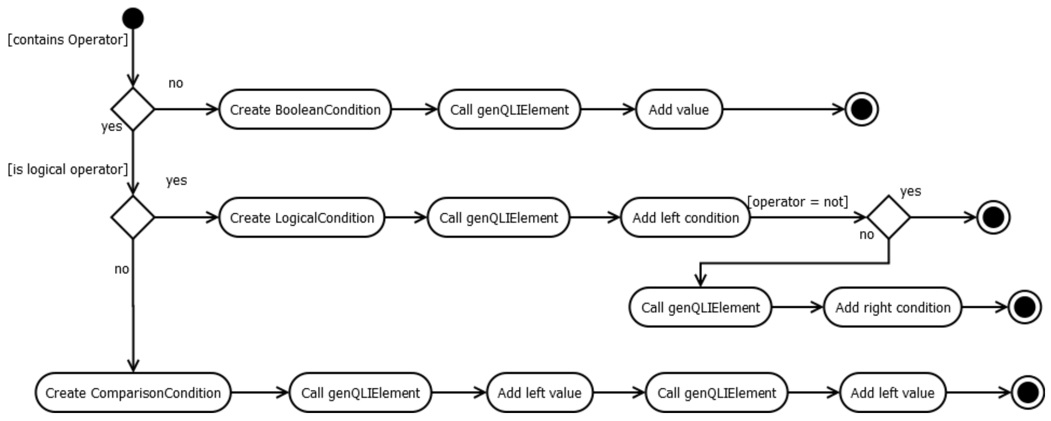


Figure 6.4: Activity diagram of the QLI Model artifact generation.

As the diagram shows, *generateCondition* inputs an AST node that specifies the root element of the condition, and checks if it contains an operator. If there is not any operator specified, a *BooleanCondition* instance is created, and then the *genQLIElement* method is called. This method outputs a QLI Model artifact (*Value* instance) and it is setted as the value of the *BooleanCondition*.

By contrast, if the root element contains an operator, first the operator type is checked. Different actions are performed depending on the operator type:

- If the operator is logical (e.g. 'and', 'or', 'not'), first, a *LogicalCondition* instance is created. Next, the first element to be evaluated logically is obtained (first child of the node), and it is the input of *genQLIElement* method which is executed. This method outputs a *Value* instance that corresponds with the left value to be evaluated by the *LogicalCondition*, and it is setted as the left condition in the *LogicalCondition* instance. Following step checks if the operator is an EOL 'not', and if it is the case, the execution is finished returning the *LogicalCondition* instance. If the logical operator is different of 'not', the second child is obtained and the *genQLIElement* is executed again, obtaining the QLI Model artifact that corresponds with the second condition. It is added in the right condition of the `LogicalCondition`, and this latter is returned.

- If the operator is not logical, a *ComparisonCondition* instance is created. Next, *genQLIElement* method is executed, obtaining the *Value* instance that corresponds with the first child. It is added at the left feature of the *ComparisonCondition* instance. The process is repeated for the second child, and the Value instance is added at the right value. Finally the *ComparisonCondition* is returned.

Listing 6.1 completes this description showing code fragment of the *generateQLIElement* implementation. The shown code fragment is related to the creation of a `ConditionalSelection` type instance[1] and it is executed when visiting a node that specifies an EOL select expression.

```
public Value generateQLIElement(AST n) throws Exception{
   ...
else if (n.getType() == EolParser.POINT){
AST left = n.getFirstChild();
Value source = genQLIElem(left);
AST right = n.getSecondChild();
  if(isFilteringExpression(right)){
    if(right.getText().equals("select")){
    //SELECT EXPRESSION
    ConditionalSelection select = createConditionalSelection(
       right, source);
    return select;
    else ...
```

Listing 6.1: Fragment of the genQLIElem method.

6.1.2 Processing results: processResults

Data-type of the result expected by queries expressed with EOL could be specific of EOL. In these cases, the results obtained from the persistence should be adapted to match with the expected data type. This task is performed by the

[1]For more information about QLI Metamodel classes please return to Section 5.2

`processResult` method, which is implemented by the `EOLDriver`. The data-type is specified by the *returnType* feature value of the QLIModel instance (root model of the generated QLI Model). The type value in *returnType*, and the results obtained from the persistence are the input parameters of *processResults* method. Then, the method re-factors results, adapting them to the expected data-type.

6.2 Support for SQL Queries over Relational CDO Repositories

The MQT-Engine Framework prototype provides the implementation of the *CDO_DBStore_SQLDriver* that supports querying with the framework models persisted with relational CDO repositories. *CDO_DBStore_SQLDriver* is a specialization of the `PQLDriver` abstract class provided by the MQT-Engine framework. *CDO_DBStore_SQLDriver* provides support for transforming QLI Models into SQL queries that are executed over a CDO repository that uses DBStore.

DBStore provides a common-schema that is used within the relational databases. This data-schema contains domain-agnostic and dedicated tables that store the information related with the change history, branches, commits or user access. DBStore generates automatically one data-schema for models of each different domain. The strategy for the generation of the domain-specific data-schemas can be customized by the stakeholders, and the default mechanism for generation used by the DBStore is the horizontal mapping.

SQL is a mature and widely used declarative query language for relational databases. The generated SQL queries are for data-schemas that are generated using the horizontal mapping strategy (default mapping strategy used by DBStore). In the horizontal mapping strategy two different types of tables could be distinguished:

- **Object Tables**: these tables contain information about all instances of an specific type. The name of each table corresponds with name of the type.

- **Many-Value Reference Tables**: these tables contain objects referenced by a many-value reference of an specific type. The name of the table follows this format: ObjectTypeName_FeatureName_List .

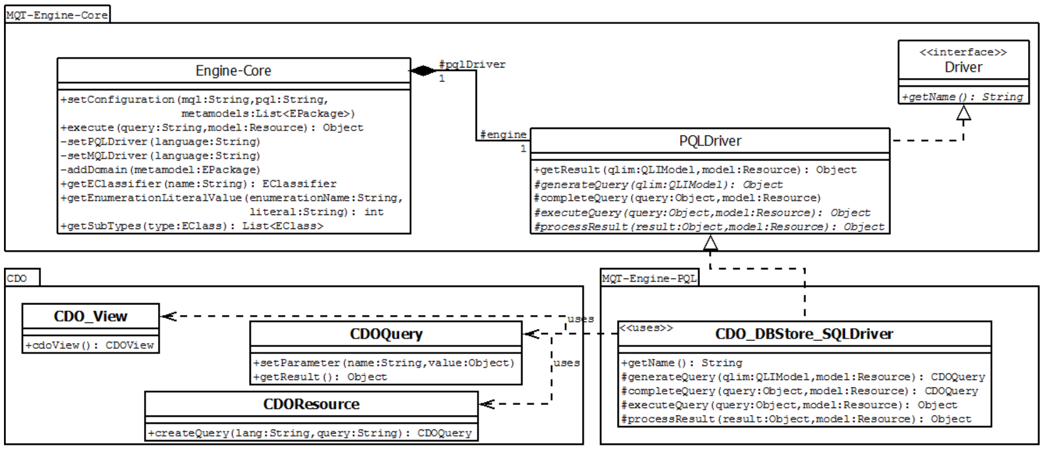

MQT-Engine-Core
Engine-Core
+setConfiguration(mql:String,pql:String, metamodels:List<EPackage>)
+execute(query:String,model:Resource): Object
-setPQLDriver(language:String)
-setMQLDriver(language:String)
-addDomain(metamodel:EPackage)
+getEClassifier(name:String): EClassifier
+getEnumerationLiteralValue(enumerationName:String, literal:String): int
+getSubTypes(type:EClass): List<EClass>
#pqlDriver
1
#engine
1
<<interface>>
Driver
+getName(): String
PQLDriver
+getResult(qlim:QLIModel,model:Resource): Object
#generateQuery(qlim:QLIModel): Object
#completeQuery(query:Object,model:Resource)
#executeQuery(query:Object,model:Resource): Object
#processResult(result:Object,model:Resource): Object
CDO
CDO_View
+cdoView(): CDOView
CDOQuery
+setParameter(name:String,value:Object)
+getResult(): Object
CDOResource
+createQuery(lang:String,query:String): CDOQuery
uses
uses
MQT-Engine-PQL
<<uses>>
CDO_DBStore_SQLDriver
+getName(): String
#generateQuery(qlim:QLIModel,model:Resource): CDOQuery
#completeQuery(query:Object,model:Resource): CDOQuery
#executeQuery(query:Object,model:Resource): Object
#processResult(result:Object,model:Resource): Object

Figure 6.5: CDO_DBStore_SQLDriver class diagram.

SQL queries generated by *CDO_DBStore_SQLDriver* use the previously described *Object Tables* and *Many-Value Reference Tables*.

Figure 6.5 illustrates the class diagram of *CDO_DBStore_SQLDriver*. As it is depicted in the figure, it uses the *CDOQuery*, *CDOView* and *CDOResource* classes of CDO for interacting with models. *CDOResource* is the class used by CDO for representing EMF models. This is the class that instantiates the model to be queried. The *CDOResource* contains the *cdoView* method and it is called by the *CDO_DBStore_SQLDriver*. *CDOView* is able to create a *CDOQuery* class, which provides support for executing SQL queries at server-side.

CDO_DBStore_SQLDriver implements the abstract methods of the *PQLDriver* class. The *getName* method returns the string value 'CDO_DBStore_SQL' which identifies the *PQLDriver* specialization. The rest of the implemented methods deal with the different tasks for query generation and execution that are performed by `PQLDriver` specializations:

- *generateQuery* method that is responsible for transforming `QLI Model` into a `SQL` query.
- *completeQuery*, which completes the SQL query with information about the version and branch of the model that will be queried.
- *executeQuery*, responsible for executing the SQL query over a CDO Repository using the CDOQuery natively provided by CDO.

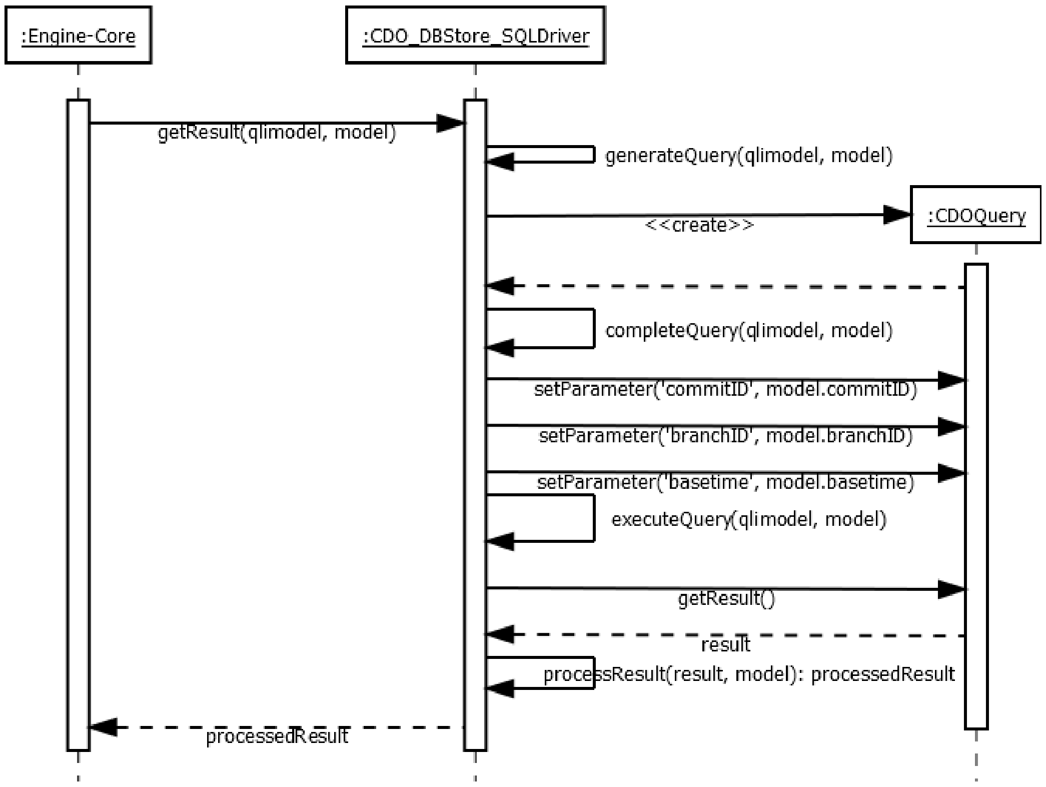

:Engine-Core
:CDO_DBStore_SQLDriver
:CDOQuery
getResult(qlimodel, model)
generateQuery(qlimodel, model)
<<create>>
completeQuery(qlimodel, model)
setParameter('commitID', model.commitID)
setParameter('branchID', model.branchID)
setParameter('basetime', model.basetime)
executeQuery(qlimodel, model)
getResult()
result
processResult(result, model): processedResult
processedResult

Figure 6.6: CDO_DBStore_SQLDriver sequence diagram.

- *processResult*, post-processes results of the query obtained from the CDO Repository, and returns only the results that are related to the queried model (or resource).

Figure 6.6 depicts a sequence diagram where the query transformation and execution performed by the *CDO_DBStore_SQLDriver* is resumed. The diagram shows the execution order of the different implemented methods that have been introduced in previous paragraphs, and it is described below: after generating a QLI Model that specifies a query, the *Enginer-Core* instance calls the `CDO_DBStore_SQLDriver` through the `getResult` method. This method inputs the QLI Model and the queried model. In this case models are persisted in CDO repositories and consequently the model is specified by a *CDOResource* class.

From QLI Model to SQL: generateQuery

The first task performed by the `getResult` method is the generation of the SQL query from the input QLI Model. This is done by the `generateQuery` method. This implementation for CDO supports querying CDO Repositories that could contain different branches and versions of a same model. The generated SQL queries contains checks and parameters related to branching and versioning. The parameters are setted later in the execution of the completeQuery method. Different parts of the generated SQL queries that contain checks and parameters of versioning and branching are:

- ***WHERE* statements that obtain information from an Object-Table.** Listing 6.2 illustrates the parameters that are included in the where SQL statement when an object table is queried: (1) `commit`, specifies the timestamp of the commit corresponding with the model version; (2) `branchID`, specifies the identifier of the branch that is being queried; (3) `hasBase`, boolean value that specifies if the branch is based in another branch; (4) `baseID`, specifies the identifier of the base branch; and (5) `baseTime`, specifies the timestamp of the corresponding version of the base branch.

```
CDO_VERSION>0
AND (
  (CDO_BRANCH =:branchID AND CDO_CREATED <= :commit AND (
      CDO_REVISED=0 OR CDO_REVISED>:commit))
  OR (:hasBase AND CDO_BRANCH =:baseID AND CDO_CREATED<=:
      basetime AND (CDO_REVISED=0 OR CDO_REVISED>:basetime))
)
```

Listing 6.2: Branching and versioning parameters in object tables.

- **INNER JOIN statements that join an object table with a many-value reference table.** Listing 6.3 illustrates the SQL expressions checking that the versions and branches of objects and references correspond.

Table 6.2 describes SQL queries that are generated from each `QLI Model` element. Therefore, a SQL string that corresponds with the QLI Model is obtained within the `generateQuery`. However, this string is not the output

Table 6.2: SQL queries generated for each QLI model element.

QLI Element	Generated SQL

KindInstances	`(SELECT KindFeatures.* FROM KindTable WHERE ...)`
	`UNION ALL (SELECT KindFeatures.* FROM SuperType1Table WHERE ...)`
	`UNION ALL (SELECT KindFeatures.* FROM SuperType2Table WHERE ...) ...`

TypeInstances	`SELECT * FROM TypeTable WHERE ...`
LogicalCondition	`(rightStatementSQL (AND | OR | ...) leftStatementSQL)`
ArithmeticalCondition	`(rightStatementSQL ( = | < | ...) leftStatementSQL)`
ValueCondition	`valueSQL`
PrimitiveValue	strings: *'value'*; other types: *value*

ConditionalSelection	method-specific SQL. Ex.: oneResult=false, negativeCondition=false
	if is contained by another filtering query
	`INNER JOIN (VariableIteratorSQL) AS iteratorName`
	`ON ... AND condSQL`
	else
	`SELECT iteratorName.*`
	`FROM (VariableIteratorSQL) AS iteratorName`
	`WHERE ... AND condSQL`

ConditionalCheck	method-specific SQL. Ex.: checkLogic = leastOne
	if is contained by another filtering query
	`EXISTS (SELECT iteratorName.*`
	`FROM (VariableIteratorSQL) AS iteratorName`
	`WHERE condSQL)`
	else
	`SELECT COUNT(iteratorName.*)>0`
	`FROM (VariableIteratorSQL) AS iteratorName`
	`WHERE condSQL)`
	`LIMIT 1`

CollectInstances	if is contained by another filtering query
	`SELECT collectedValue.CDO_ID`
	`FROM ...`
	`INNER JOIN (VariableIteratorSQL) AS iteratorName ON ...`
	else
	`SELECT collectedValue.CDO_ID`
	`FROM (VariableIteratorSQL) AS iteratorName`

PositionValue	if inverseOrder=false
	`SELECT * FROM collectionSQL LIMIT 1 OFFSET position-1`
	if inverseOrder=true
	`SELECT * FROM collectionSQL`
	`ORDER BY ROWNUM DESC LIMIT 1 OFFSET position-1`

Size	`SELECT COUNT (*)`
	`FROM collectionSQL`

Defined	method-specific SQL. Ex.: negative=false EXISTS (collectionSQL)
Contains	method-specific SQL. Ex.: listValues=false, negative=false EXISTS (collectionSQL AND collectionAlias.CDO_ID=containedObject.CDO_ID)
IteratedObject	specific SQL. dependening on contained features: EX. no features: iteratorName.CDO_ID EX. with attribute and without references: iteratorName.attributeName EX. with single-value reference: iteratorName.referenceName EX. with many-value reference: SELECT FeatureType.CDO_ID FROM FeatureType INNER JOIN ParentType_Feature_List ON ...

```
objectTable.CDO_VERSION = referenceTable.CDO_VERSION
AND objectTable.CDO_BRANCH = referenceTable.CDO_BRANCH}
```

Listing 6.3: Branching and versioning parameters in object tables.

of the method. `generateQuery` method creates a CDOQuery class instance that contains the SQL query string. CDOQuery is a class provided by CDO and it is able to execute queries using SQL and over the database. The method execution ends when the CDOQuery is returned.

SQL query completion: completeQuery

The SQL query specified within the CDOQuery returned by `generateQuery` contains a set of parameters to be specified before executing it. These parameters are setted by the `completeQuery` method.

Listing 6.4 illustrates the code of `completeQuery` method. It inputs the queried model and a generated query, and it first checks that the queried model corresponds with a CDOResource instance (lines 2-4) and that the query is specified by a CDOQuery instance (lines 5-7). `CDOQuery` contains the `SQL` query generated in the previous steps. Parameter values are obtained from the queried model (lines 8-20), and next, the obtained values are setted to the generated SQL through the `CDOQuery` instance (lines 21-25). `completeQuery` finishes the execution returning the CDOQuery instance that contains the completed query (line 26).

SQL query execution: executeQuery

After setting parameters, *CDO_DBStore_SQLDriver* proceeds with the `SQL` query execution through the `executeQuery` method. This method uses the `CDOQuery` and it is able to execute query directly against the `CDO Repository` and at server-side. The `SQL` query is executed against the entire repository. Listing 6.5 depicts the code of the execute query method.

SQL Query Result processing: processQuery

SQL query results obtained from the database correspond to all models (`CDOResource`) of the repository. However, target of queries expressed with model query languages commonly is a single-model. To provide query results for

```
public CDOQuery completeQuery(Object query, Resource model){
 if (!(query instanceof CDOQuery))
   throw new UnsupportedOperationException("query is not
       instance of CDOQuery");
 CDOQuery cdoQuery = (CDOQuery) query;
 if (!(model instanceof CDOResource))
   throw new UnsupportedOperationException("model is not
       instance of CDOResource");
 CDOResource cdoModel = (CDOResource) model;
 CDOView view = model.getView();
 long commitID = view.getTimeStamp();
 if(commitID == 0){
  commitID = view.getLastUpdateTime();
 }
 long branchID = view.getBranch().getID();
 long parentBranchID=-1;
 boolean existsBase = false;
 if(view.getBranch().getBase().getBranch() != null){
  parentBranchID = view.getBranch().getBase().getBranch().
      getID();
  existsBase = true;
 }
 long basetime = view.getBranch().getBase().getTimeStamp();
 cdoQuery.setParameter("commitID", Long.toString(commitID));
 cdoQuery.setParameter("branchID", Long.toString(branchID));
 cdoQuery.setParameter("parentBranchID",Long.toString(
     parentBranchID));
 cdoQuery.setParameter("existsBase", existsBase);
 cdoQuery.setParameter("basetime",Long.toString(basetime));
 return cdoQuery;
}
```

Listing 6.4: Implementation of the completeQuery method.

```
protected Object executeQuery(Object query, Resource model) {
  if (!(query instanceof CDOQuery))
    throw new UnsupportedOperationException("query is not
        instance of CDOQuery");
  return ((CDOQuery)query).getResult();
}
```

Listing 6.5: Implementation of the executeQuery method.

a specific model only, the *CDO_DBStore_SQLDriver* processes them using the `processResult` method.

The `processResult` method filters and analyses the SQL results returned by the query executed at server-side, and only returns results that are part or are related to the queried model. It has been decided to do this post-process, because including it in the transformation would require complex SQL queries that could have impact in performance.

The performed result processing is different depending on the results expected by the query: if the query expects a collection of model objects, this method filters them and selects only those that are part of the model. Model objects are specified in CDO using CDOObject instances, and these instances provide cdoResource method which returns the parent CDOResource of the model object. This way, the `processResult` method compares that the parent resource of the model object is equal to the queried model (CDOResource instance). By contrast, if the query expects a boolean value (e.g. check if a certain element type exists within the model), the boolean value to be returned by the query is chosen in the `processResult` method: to check if a result exists in a model, obtained results are analysed (e.g. `while(res.hasNext()){ if (res.getNext().cdoResource == resource) return true;} return false;`).

Part III

Validation

7 Experimental Evaluation: Reverse Engineering Case

This chapter describes an experimental evaluation based on the academic reverse engineering case study proposed at Graph-Based Tools 2009 (a.k.a. GraBaTs'09) [Gra09,Sot09]. This case study identifies a set of models that specify source code of Java projects, with the aim of comparing execution time and memory usage metrics to execute a complex query over them. Proposed query extracts all the singleton classes that are specified within the models.

This experimental evaluation addresses the following questions:

- **Question 1**: Which is the performance and memory usage of MQT-Engine for querying models persisted in a CDO repository and using a model query language (concretely EOL)?
- **Question 2**: Obtained results are better or worse (in terms of memory usage and execution time) than using other approaches (Plain EMF, MDT OCL, CDO_OCL, SQL) for querying CDO models?

This chapter is structured as follows: first, the experiment scenario is presented describing the context, the execution environment and metrics, the queried models and the executed queries. Then, obtained results are shown and analysed.

7.1 Experiment Scenario

This section describes the scenario used to perform this experimental evaluation, and it provides: overview of the case study, explanation of the execution environment and obtained metrics, queried models, and executed queries.

7.1.1 Context

The Graph-Based Tools 2009 (a.k.a. GraBaTs'09) Reverse Engineering Case Study proposes a set of tasks for the benchmark of model queries and transformations [Gra09, Sot09]. The first task of the case study focuses on the scalability of approaches in terms of performance and memory usage.

This case study has been used in a large amount of works: to evaluate persistence of models in graph databases [Bar14]; to evaluate persistence of models using Morsa [EP11]; at works that are focused on model query: Hawk [Bar13] or MorsaQL [EP14]; and to evaluate a framework to benchmark scalability of NoSQL data-stores [Sha14]. Although the used metamodel is different, GraBaTs'09 query has been also used in the evaluation of NeoEMF/Graph [Ben14] and NeoEMF/Map [Góm15b].

Reverse Engineering Case Study models conform to the JDTAST domain metamodel. This metamodel contains abstractions of the Java source code. The case study provides the implementation of JDTAST metamodel in the *org.amma.dsl.jdt* project. The metamodel is composed by three packages (EPackage instances): *Core* package, which contains abstractions related to the creation, edition and build of java programs; *DOM* package, which contains abstractions for specifying java source code as a structured document; and *PrimitiveTypes* package, containing abstractions for primitive types existing in the domain (String, Boolean and Integer).

Figure 7.1 depicts a simplified fragment of the metamodel extracted from [Esp13, p. 61]. As the figure shows, models that conform to the metamodel have a *IJavaModel* root instance. This instance contains one *IJavaProject* instance for each existing Java project, and *IJavaProject* instances contain *IPackageFragmentRoot* instance that specify root package fragments. The *IPackageFragmentRoot* class extends the *IPackageFragment* class which contains a set of *ICompilationUnit* instances. All these classes are contained by the Core package.

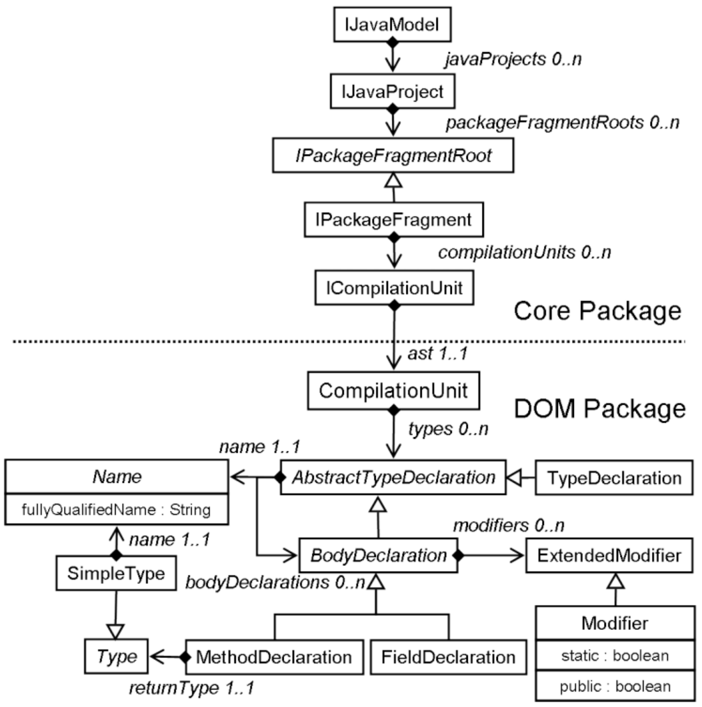


Figure 7.1: Simplified version of the JDTAST metamodel.

extracted from [Esp13, p. 61]

Each *ICompilationUnit* instance contains a *CompilationUnit* class instance, which specifies the source code of the corresponding compilation unit. Compilation units could be of different types, and it is specified by the *types* feature which contains an *AbstractTypeDeclaration* instance. This class contains a *Name* instance which will specify the fully qualified name of the type. *AbstractTypeDeclaration* class is extended by different classes, one for each existing compilation unit type. Existing types include:

- *TypeDeclaration*. This class specifies types declared within the source code.
- *BodyDeclaration*. *BodyDeclaration* instances contain modifiers (*ExtendedModifier* instances). Modifiers can be of different types,

and they are specified by classes that extend the *ExtendedModifier* class. Modifier instances are one of these specializations, and they contain two attributes with boolean values that indicate if the declaration is static (*static* feature) and if the declaration is public (*public* feature). *BodyDeclaration* class has different specializations which includes a specialization for declaration of methods (MethodDeclaration) or a specialization for declaration of fields (FieldDeclaration). In the case of *MethodDeclaration* instances, the reference *returnType* contains a *Type* instance that specifies the type of the returned object. Type instance is extended by different classes, and *SimpleType* is one of them.

7.1.2 Execution Environment and Metrics

All the experiments of this experimental evaluation have been executed as a standalone application over a Microsoft Azure[1] virtual machine configured with a 4 Core processor, 14GB of RAM, 200GB SSD, and running 64-Bit Windows Server 2012 and Java SE v1.8.0. We have used Eclipse Mars with CDO 4.4. CDO repositories have been executed in embedded mode[2] to measure total memory usage and avoid the uncertainty of connections in the execution time. Repositories run on top of H2 v1.3.168, using the DBStore with its default mapping, caching and pre-fetching values, and supporting audits and branches.

Evaluated queries have been expressed using different query languages:

- **Plain EMF**. CDO provides support for executing queries using the Plain EMF API. These queries are executed at client-side and they require to load the entire model in-memory.

- **OCL**. CDO supports executing OCL queries at both client- and server-sides. The client-side execution of OCL (a.k.a. MDT OCL) is similar to Plain EMF and requires to load the entire model in-memory. By contrast, CDO also supports the server-side execution of the OCL queries (a.k.a. CDO-OCL) using the *OCLQueryHandler*.

- **SQL**. CDO configured with DBStore provides support for the server-side execution of the SQL queries. SQL queries are directly executed over the database and they do not require to load models in advance.

[1]Azure: `https://azure.microsoft.com/en-us/services/virtual-machines/`

[2]Read more about CDO Embedded at `https://wiki.eclipse.org/CDO/Embedded`

- **EOL**. The MQT-Engine prototype presented at this dissertation provides support to execute EOL queries over relational CDO repositories (configured with DBStore). MQT-Engine transforms queries into SQL and executes them in the server-side, and over the database. Hence, query does not require to load models in advance.

In order to get reliable numbers, each query has been executed 5 times for each evaluation case and Java Virtual Machine has been restarted for each execution. Correctness of query-results has been ensured by automatically comparing the results of each query using different languages.

Results have been evaluated against the following quantitative metrics:

- **M1**: Query Result Size. M1 specifies the number of results returned by each query.

- **M2**: Average Execution Time (in seconds). The average time is calculated from the five executions of the same query over same model and using same query language.

- **M3**: Maximum Memory Usage (in MB). M3 includes memory used by the CDO Client and Server. The maximum time is calculated from the five executions of the same query over same model and using same query language.

An in-house Java profiler has been used to collect performance and memory usage results that correspond with M2 and M3 metrics.

7.1.3 Models

The GraBaTs'09 Reverse Engineering Case Study provides five different models that conform to the JDTAST domain metamodel. The size of models ranges from 8.8MB, containing 14 java classes and more than 70k model elements, to 646MB, containing almost 6k java classes and 5M model elements.

The case study models are natively persisted using the XMI persistence. All the experiments of this evaluation execute queries over models persisted in CDO repositories and CDO persistence is not natively supported by the case study. Consequently a CDO-specific implementation of the domain metamodel has been provided using the CDO Model Migrator[3]. Therefore, XMI models have been

[3]More information at `https://wiki.eclipse.org/CDO/Preparing_EMF_Models`

migrated to conform the CDO-specific implementation of the metamodel, and then they have been persisted in CDO repositories. The migration mechanism is based on a migration utility implemented to evaluate NeoEMF/Graph and NeoEMF/Map [Góm15b].

Table 7.1 depicts details of the models used in this evaluation. Models Set0 to Set4 are provided by the case study[4], and each model is persisted in a separated CDO repository. By contrast, Set5 to Set9 are CDO repositories that contain several copies of the migrated Set4 model. The number of copies persisted in each repository is described in 'Number of models' column of the table.

Table 7.1: Main characteristics of queried models.

	XMI size	**CDO Repo. size**	**Numb. of models**	**Model Elem.**	**Numb. of Java Classes**
Set0	8,8	15.3MB	1	70,447	14
Set1	27	43.8MB	1	198,466	40
Set2	271	307MB	1	2,082,841	1,605
Set3	598	784MB	1	4,852,855	5,314
Set4	646	1.17GB	1	4,961,779	5,984
Set5	n/a	2.01GB	2	9,923,558	11,968
Set6	n/a	2.88GB	3	14,885,337	17,952
Set7	n/a	3.67GB	4	19,847,116	23,936
Set8	n/a	4.45GB	5	24,808,895	29,920
Set9	n/a	5GB	6	29,770,674	35,904

The experiments have been grouped by two different configuration factors (F) that may impact:

- **F1, Size of the model**: This factor measures how the increasing size of the model may influence on the performance and memory usage for the query execution. The size measure is the number of model elements that contains each model (see Table 7.1). Experiments grouped by this factor use models that have been persisted in independent CDO repositories (models from Set0 to Set4).

[4]More details described at `http://web.emn.fr/x-info/atlanmod/index.php?title=GraBaTs_2009_Case_Study`

- **F2, Size of the repository**: CDO is able to store many models in the same repository. The experiments that are part of F2 measure how the increasing size of the repository influences the performance and memory usage. The size of the repository specifies the number of models and elements within the repository. For this factor, different model copies have been persisted in the same CDO repository (models from Set4 to Set9).

7.1.4 Queries

Three different queries are executed over models in this experimental evaluation. All the selected queries traverse the entire model but with increasing complexity. They are described following:

- **Q1**: Number of classes (*TypeDeclaration* instances) that exist within the model. Listing 7.1 illustrates the query expressed with EOL.

```
1 TypeDeclaration.all;
```

Listing 7.1: Q1 query expressed with EOL.

- **Q2**: Number of private methods (*MethodDeclaration* instances) that exist within the model. Listing 7.2 illustrates the query expressed with EOL. As the code shows, this query first obtains all the *MethodDeclaration* instances (line 1), and then selects only those that contain a *Modifier* with the private feature valued as true (line 2).

```
1 MethodDeclaration.all
2   .select(md | md.modifiers.exists(mod: Modifier|mod.
       private=true));
```

Listing 7.2: Q2 query expressed with EOL.

- **Q3**: Number of singleton classes (*TypeDeclaration* instances) that exist within the model. Listing 7.3 illustrates the query expressed with EOL. As the code shows, first all the *TypeDeclaration* instances are collected (line 1). Then, it selects all the instances that contain a *MethodDeclaration* instance (line 2) which satisfies following conditions: (1) contains a modifier with the public feature value as true (line 3); (2) contains a modifier with the static feature value as true (line 4); (3) the returnType is a SimpleType instance (line 5); and (4) the returnType is the class itself (line 6).

```
1 TypeDeclaration.all
2   .select(td | td.bodyDeclarations.exists(md:
        MethodDeclaration |
3     md.modifiers.exists(mod: Modifier|mod.public=true)
4     and md.modifiers.exists(mod: Modifier|mod.static=true)
5     and md.returnType.isTypeOf(SimpleType)
6     and md.returnType.name.fullyQualifiedName = td.name.
          fullyQualifiedName));
```

Listing 7.3: Q3 query expressed with EOL.

Q3 is the complex query that is proposed by the GraBaTs 2009 Case Study. The evaluation has been extended with two queries that have a lower complexity level (Q1 and Q2). Table 7.2 shows the result size returned by queries when they are executed over each model.

Table 7.2: Result size of queries for each model.

	Set0	Set1	Set2	Set3	Set4	Set5	Set6	Set7	Set8	Set9
Q1	14	40	1,605	5,314	5,984	5,984	5,984	5,984	5,984	5,984
Q2	4	38	1,793	9,275	10,086	10,086	10,086	10,086	10,086	10,086
Q3	1	2	41	155	164	164	164	164	164	164

7.2 Results

This section shows and describes results obtained in the experiments. Results are grouped by the two previously described configuration factors (F1 and F2).

7.2.1 F1: Size of the Model

Figures 7.2 to 7.7 illustrate execution time and memory usage results required by the queries using the different query languages. Additionally, a table with the results is included in Appendix B.

As the results show, size of the queried model has a great impact over the time and memory required in Plain EMF and MDT OCL, and three queries show

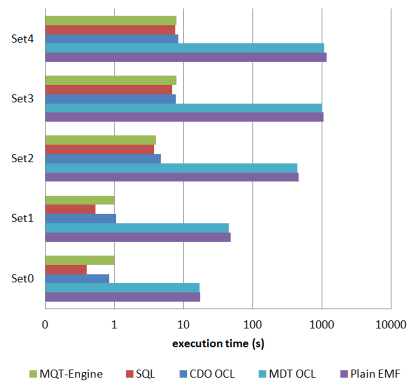


Figure 7.2: Execution time results for Q1 in F1.

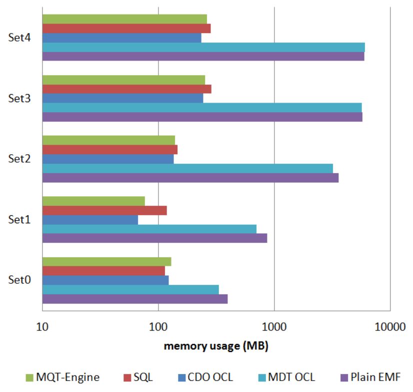


Figure 7.3: Memory usage results for Q1 in F1.

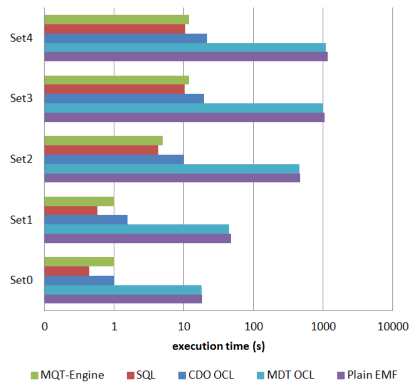


Figure 7.4: Execution time results for Q2 in F1.

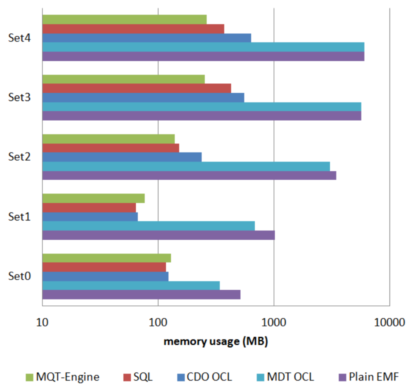


Figure 7.5: Memory usage results for Q2 in F1.

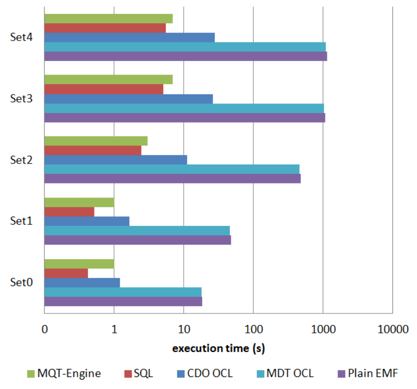


Figure 7.6: Execution time results for Q3 in F1.

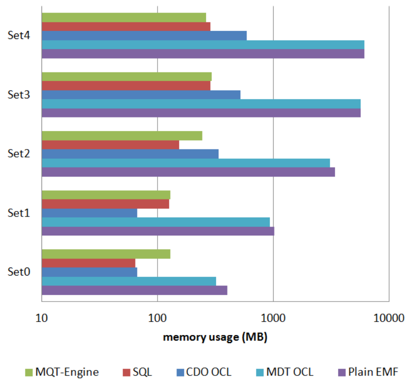


Figure 7.7: Memory usage results for Q3 in F1.

similar values (entire model is always loaded in memory). In Set4, these client-side solutions require more than 6000% of time of Set0 and more than 1100% of memory. *Plain EMF* requires 17-18 s and 396-513 MB for querying the smallest model (Set0) and 1140-1166 s and 6-6.1GB for the largest (Set4). Model size impact is slightly lower for *MDT OCL* as it requires 17-18s and 322-342MB for Set0 and 1090-1101 s and 6-6.1 GB for Set4.

The impact of the model size is lower if queries are executed at server-side. In Set4, these solutions require up to 2400% of time of Set0 and up to 800% of memory. However, increase values are much lower than on client-side solutions.

CDO-OCL is more than 17 times faster than Plain EMF and MDT OCL, and it only requires 1 s to execute queries in Set0. Memory usage is also reduced to 123 MB (Q1-Q2) and 67 MB (Q3). Regarding other sets, results vary depending on the query: Q1 requires less time than Q2 and Q3, and Q2 less than Q3. For example, in Set4 Q1 requires 8 s, Q2 22 s and Q3 28 s. However, Q3 is more than 38 times faster than any query in Plain EMF or MDT OCL. In terms of memory, Q1 requires less than Q2 and Q3: in Set4 Q1 needs 235 MB, Q2 636 MB and Q3 590MB. Worst memory value (636 MB) is more than 9 times lower than the best memory usage result of the client-side solutions. *SQL* shows better results: queries require less than a second and 118 MB in Set0; and less than 12 s and 375 MB in Set4. Q1 requires less time and memory than Q2 and results are similar to CDO-OCL. In the case of Q2 it is 2 times faster than CDO-OCL and memory usage is reduced by 40% for Set4. Q3 time and memory results are lower than Q1 and Q2, and it is more than 4 times faster than CDO-OCL requiring less than 50% of memory.

Performance and memory results of *MQT-Engine* for executing queries using EOL are similar to SQL. Execution time results show that MQT-Engine requires between 1 and 2 s more than SQL to be executed. The generated SQL query is the same that is used in the SQL experiments, and it indicates that the extra-time corresponds with the EOL to SQL transformation. MQT-Engine requires 1s and less than 130 MB for executing queries in Set0, and less than 8 s and 315 MB in Set4. As occurs in SQL, Q3 requires less time and memory than Q1 and Q2, and Q1 less than Q2. For example in Set4: Q1 requires 8 s and 263 MB, Q2 12 s and 315 MB, and Q3 7 s and 263 MB. MQT-Engine results are significantly better than using the other server-side solution (CDO-OCL), and much better than using a client-side solution (Plain EMF or MDT-OCL).

7.2.2 F2: Size of the Repository

Figures 7.8 to 7.13 illustrate execution time and memory usage results required by the queries using the different query languages in the experiments that are part of the F2 factor. Additionally, a table with the results has been included in Appendix B.

As the results show, time and memory results obtained querying F2 models (set4-set9) indicate that the size of the repository has not influence in queries executed in the client-side: in the case of *Plain EMF* execution time value for executing queries is between 1140-1174 s and requires around 6 GB of memory; in the case of *MDT OCL* the execution time is slightly lower (between 1081-1113 s) and also requires around 6 GB of memory.

This scenario changes in the case of the server-side solutions, where the size of the repository has influence. *CDO-OCL* results show a constant increase of the query execution time from one repository to the subsequent one (e.g. from Set5 to Set6). The increase changes according to query: between 20-28 s for Q1, 31-41 s for Q2, and 35-42 s for Q3. Memory usage increases: from 235 MB to 695 MB in Q1; from 636 MB to 1860 MB in Q2; and from 590 MB to 2171 MB of Q3. The influence of the repository size is greater in Q3, which requires more time and memory. In Set4, CDO-OCL requires around 1100% of time of Set0 and around 330% of memory. The trend is similar in *SQL*, but the time increase between repositories is lower: 4-6 s for Q1, 7-8 s for Q2, and 4-5 s for Q3. Memory values increase from 285 MB to 849 MB for Q1; from 375 MB to 1249 MB for Q2; and from 289 MB to 968 MB for Q3. In the case of SQL, repository-size influence is greater in Q2. Q3 is resolved faster and Q1 requires less memory than others. In Set4, SQL requires around 430% of time of Set0 and around 300% of memory. While increase is similar to CDO-OCL in the case of the memory, increment of the execution time is lower.

MQT-Engine results agree with those obtained in SQL and execution time and memory is also influenced by repository size. However, influence is lower than in CDO-OCL. Execution time difference between SQL and MQT-Engine is only of 1-2 s (transformation time overhead). In terms of memory, MQT-Engine uses less memory than others (including SQL): from 263 MB to 761 MB for Q1, from 315 MB to 1136 MB for Q2, and from 264 MB to 579 MB for Q3. The filtering mechanism provided by MQT-Engine could be the reason of memory usage difference between SQL and MQT-Engine. Results show that the execution time and memory usage of MQT-Engine is much lower than the required by the client-side solutions (Plain EMF and MDT OCL). Moreover,

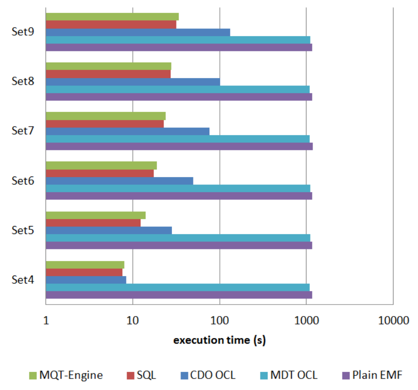


Figure 7.8: Execution time results for Q1 in F2.

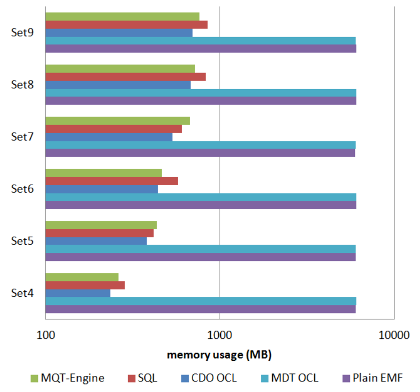


Figure 7.9: Memory usage results for Q1 in F2.

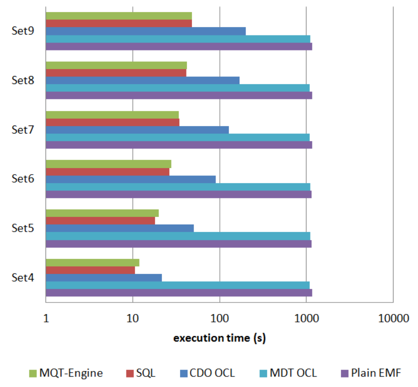


Figure 7.10: Execution time results for Q2 in F2.

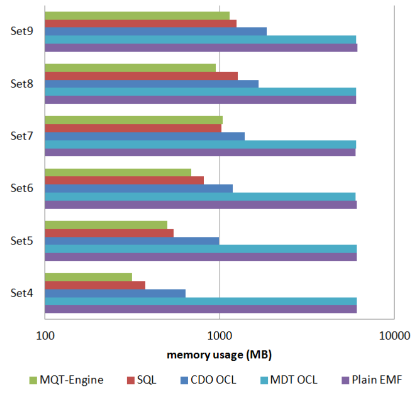


Figure 7.11: Memory usage results for Q2 in F2.

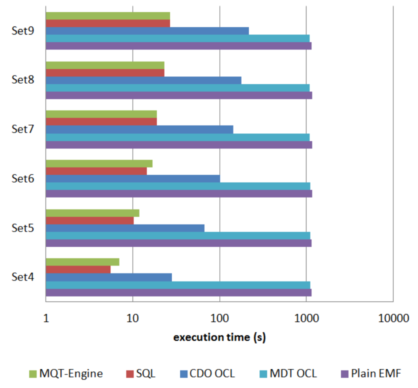


Figure 7.12: Execution time results for Q3 in F2.

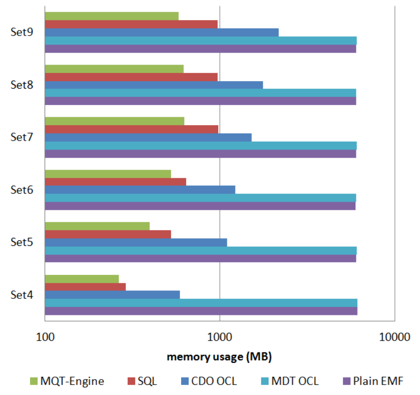


Figure 7.13: Memory usage results for Q3 in F2.

MQT-Engine also resolves queries faster than the natively provided server-side version of OCL (CDO-OCL).

7.2.3 Threads to Validity

Models used within these experiments have been generated for test-case purpose. Using industrial models and real model operations would be more realistic.

The experiments have been executed using an in-house framework. This framework provides an in-house Java profiler, and it has been used to obtain performance and memory usage metrics.

7.2.4 Conclusions

The experiments have compared the performance and memory usage results of executing different model query languages (Plain EMF, OCL at executed client- and server-side, SQL and EOL using MQT-Engine) over models persisted in CDO repositories.

Results of F1 show how the increasing size of the model impacts over the performance and memory usage when executing queries using MQT-Engine, Plain EMF, OCL (executed at client- and server-side) and SQL. The impact has great impact over Plain EMF and client-side OCL. By contrast, impact is lower in solutions executed at server-side, and especially in MQT-Engine and SQL which show best performance and memory usage results.

F2 experiments show the impact of increasing size of the model repository when executing queries. Results have shown great impact over server-side solutions (server-side OCL, MQT-Engine and SQL). In the case of client-side solutions (Plain EMF and client-side OCL) performance and memory results have been similar for all model sizes. However, server-side solutions have shown much better performance and memory usage results, and specially SQL and MQT-Engine.

Some of the obtained results address the previously described *Question 1* of this experimentation case, and shows the memory usage and execution time required by the MQT-Engine framework for transforming EOL queries into SQL, and executing them over the CDO repository.

Moreover, the results are compared with the results of other approaches. This comparison addresses *Question 2* of the experimentation case. Obtained results show that MQT-Engine is a promising alternative for making queries from EOL to CDO repositories. Results indicate that MQT-Engine is much faster and use less memory than model query languages executed in the client-side of CDO (Plain EMF and OCL). Moreover, obtained results are better than the natively supported server-side execution of the OCL queries.

8 Experimental Evaluation: Train Benchmark Case

This chapter describes an experimental evaluation based on the Train Benchmark Case [Szá15]. This benchmark case uses models and queries that are close to a real industrial domain: railway domain.

This experimentation case obtains performance metrics for executing the proposed queries over models that are persisted using different approaches (XMI, MySQL, Neo4J and CDO). Moreover, the queries are expressed using different languages (e.g. Plain EMF, OCL, cypher, EOL). These experiments include the evaluation of the MQT-Engine prototype configured to query with EOL, models persisted in CDO repositories.

This experimental evaluation addresses the following questions:

- **Question 1**: Which is the performance of MQT-Engine for executing the queries of the Train Benchmark Case over models persisted in CDO repositories and using EOL, a model query language?
- **Question 2**: How are the results compared to execution time required by other query languages (Plain EMF, OCL at client-side and OCL at server-side) for querying models in CDO repositories?

- **Question 3**: How are the results compared to performance results of querying models persisted with other approaches (XMI, MySQL database or Neo4J database)?

This chapter is structured as follows: first, the Train Benchmark Case is explained, describing used domain metamodel, models, queries and model modifications; next, the experimentation scenario is described, including evaluated approaches and used configurations; and finally, obtained results are analysed.

8.1 Experimental Scenario

8.1.1 Context

The Train Benchmark Case has been designed for evaluating the performance of batch and incremental queries, and it has been used in several scenarios: as a challenge in the 8th Transformation Tool Contest held in 2015 [Szá15]; for evaluating approaches for model query and transformations, concretely EMF-IncQuery [Ujh15]; and as a case study of the MONDO[1] project supported by the European Commission.

The Train Benchmark Case uses models of the railway domain. The implementation of the domain metamodel conformed by these models (railway metamodel) is provided by the case study in the *hu.bme.mit.trainbenchmark.emf.model* project.

Figure 8.1 illustrates the artifacts of the metamodel and their relationships (hierarchy and containment). As the figure shows, each model will contain a *RailwayContainer* instance that is the root element of the model. This root instance will contain semaphores (specified by *Semaphore* instances) and routes (specified by *Route* instances). Semaphores have a signal feature with three possible values that are specified within the *Signal* enumeration. Routes have an entry and exit semaphore, and they are defined by two or more sensors (*Sensor* instances). Sensors are the artifacts that track route segments (*Segment* instances) and railway switches (*Switch* instances). *Segment* instances have a *length* attribute containing an integer value (*EInt*) that indicates the length of the segment. *Switch* instances have a attribute that specifies the current position of the switch, and the possible values are defined at the *Position*

[1]Find more about MONDO project at http://www.mondo-project.org/

enumeration: *FAILURE*, *LEFT*, *RIGHT* and *STRAIGHT*. Switches are referenced by *SwitchPosition* instances that indicate the position (values from the *Position* enumeration) that have the switches within each route.

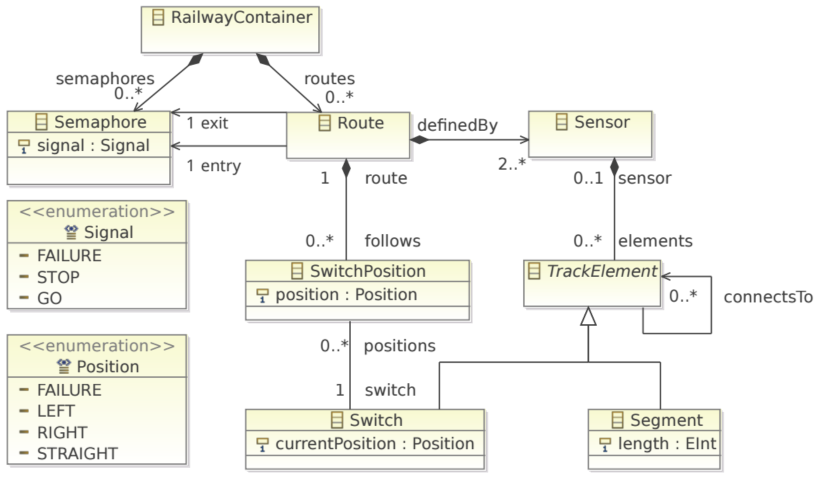


Figure 8.1: Containment and hierarchy relationships in the railway metamodel

extracted from [Szá15, p. 3]

Figure 8.2 provides a different viewpoint of the metamodel that focuses on the generalization and specialization relationships of the artifacts existing within the metamodel. As the figure shows, *Semaphore*, *Route*, *TrackElement*, *SwitchPosition* and *Sensor* classes are a specialization of the *RailwayElement* class. This latter contains an attribute that identifies each artifact (*id* feature). *Segment* and *Switch* classes are a specialization of the *TrackElement* class.

Train Benchmark Case provides an evaluation framework for assessing scalability of validating and re-validating well-formedness constraints over models that conform to the previously described domain. This benchmark presents a set of experiments composed by the typical constraints that are used on the railway domain. Each experiment contains a set of phases, and each phase is detailed below:

- **Read Phase**. In those experiments where queries are executed in-memory (e.g. using XMI), read phase encompasses process where information required by the corresponding persistence is loaded from the physical-support to the memory. In cases where databases are used (e.g. using CDO), read phase encompasses process for connecting to the database.

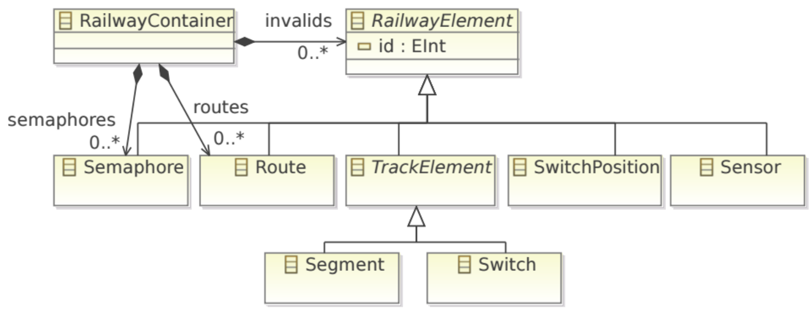


Figure 8.2: Super-type relationships within the railway metamodel

extracted from [Szá15, p. 3]

- **Check Phase**. It encompasses query-processing, execution and also returning the results.
- **Manipulation Phase**. The model is automatically changed within this phase for simulating model modifications. These modifications are performed on a subset of elements returned by the previous phase. Manipulation phase uses one of the following two strategies: fixed, a constant number of invalid model elements is modified, testing in this way efficiency for handling small change sets; and proportional, a percentage of the invalid model elements is modified, testing in this way efficiency for handling large change sets.
- **Re-Check Phase**. In this phase, the query is re-executed over the modified model.

The Train Benchmark Case presents three different scenarios for the experiments: *Batch validation* scenario, *fault-injection* scenario and *automated model-repair* scenario. *Fault-injection* scenario has been omitted from the evaluation due to problems to execute it correctly when models are persisted in CDO. Therefore, the experiments that compose this experimental evaluation are related to the batch validation scenario or to the automated model-repair scenario. Both scenarios are described below:

- **Batch validation scenario** (a.k.a. batch scenario). A set of queries are executed over models to check different constraints. Each experiment using this scenario is composed by one execution of the read phase and one execution of the check phase.

- **Automated model-repair scenario** (a.k.a. repair scenario). This scenario is similar to the fault-injection scenario. In this case, in the manipulation phase a set of model elements that break the constraint are modified, decreasing the number of broken constraints. Each experiment of this scenario executes once the read and check phases and then executes ten successive executions of the manipulation and re-check phases.

8.1.2 Execution Environment and Metrics

All the experiments have been executed on a Kernel-based Virtual Machine (KVM)[2] configured with a dual-core Xeon processor, 16GB of RAM, 120GB of storage, and running 64-Bit Ubuntu 14.04 LTS. Oracle Java SE 1.8.0_66 and Eclipse Mars.1 Release (4.5.1) had been used. Timeout value for each experiment is 25,000 seconds.

Different persistences and query languages have been used in experiments of this experimental evaluation:

- **XMI**: Natively supported by the Train Benchmark Case. XMI models are queried using the following query languages: **Plain EMF**, **IncQuery** (two configurations: local and incremental) and **OCL**.
- **MySQL**: MySQL based persistence of models is also supported by the Train Benchmark Case, and these models are queried using **SQL**.
- **Neo4J**: **Neo4J** (noSQL database-engine) is also supported. These models are queried using the **Core API** of Neo4J, and using **Cypher** (Neo4J-specific graph query language).
- **CDO repositories**: CDO repositories are not natively supported by the Train Benchmark Case framework. CDO-specific implementation has been provided extending the classes and interfaces of the framework. Models are persisted in CDO repositories that use DBStore with the default configuration and mapping strategy. CDO models are queried using **Plain EMF**, **OCL** (executed at client-side and at server-side) and with **EOL** (executed using MQT-Engine).

XMI, MySQL and Neo4J persistences use the railway metamodel natively provided by the Train Benchmark Case. However, a CDO-specific implementation

[2]More about KVM at http://www.linux-kvm.org/

of the metamodel has to be provided in order to support CDO repositories in the Train Benchmark Case Study framework. The implementation has been provided using the CDO Model Migrator[3].

In order to get reliable numbers, each experiment has been repeated 5 times for each case and for each model. A new instance of the Java virtual machine is created for each experiment, setting the maximum heap memory value to 10GB. Correctness of query-results has been ensured by automatically comparing the results of each query using different languages.

Obtained results have been evaluated against the following quantitative metrics:

- **M1**: Reading time. M1 specifies the average time (in seconds) required for performing the read phase.
- **M2**: Checking execution time. It specifies the average time (in seconds) required for executing the query over the model and returning the results.
- **M3**: Checking result size. M3 specifies the number of results returned by the query on the check phase.
- **M4**: Rechecking execution time. It specifies the average time (in seconds) required for executing the query over the model and returning the results.
- **M5**: Rechecking result size. M5 specifies the number of results returned by the query in the re-check phase.

The Train Benchmark Framework natively provides an utility that is able to collect execution time results that correspond with M1,M2 and M4 metrics.

While the read and check phases are executed in the experiments of all the scenarios, manipulation and re-check phases are only executed at the experiments that are part of the repair scenario. Therefore, M1, M2 and M3 metrics are obtained in the experiments of both scenarios. By contrast, M4 and M5 are obtained only on the experiments that are part of the repair scenario.

8.1.3 Models

Model instances are automatically generated by the Train Benchmark Case framework. The generation mechanisms takes care for breaking symmetry during

[3]More information at `https://wiki.eclipse.org/CDO/Preparing_EMF_Models`

generation, and it makes possible to get instances that are closer to real-world models and also to prevent to query tools efficiently storing and caching models [Szá15]. Models are generated using a generation factor that increases with a geometric progression of 2. The minimum value of the generation factor is 1 and the maximum 4096. As the generation factor increases, generated models contain more elements and the size grows.

The Train Benchmark Case provides a utility for generating automatically models that are persisted with XMI, MySQL and Neo4J. By contrast, generation of CDO models is not natively supported. Therefore, an implementation for generation of models persisted in CDO repositories has been provided. This project extends the classes for model generation provided by the Train Benchmark Case. Generated models are different depending on the scenario where they are executed:

- **batch models**: these are the models that are used in experiments of the batch scenario. These models do not break any well-formedness constraint and consequently, the queries that are executed against them do not return any value. Table 8.1 describes details of these models.

Table 8.1: Batch scenario models.

	Gen. Factor	XMI Size (MB)	CDO Repo (MB)	# of objects[4]
batch-1	1	0.2	0.8	1,249
batch-2	2	0.3	1.0	2,137
batch-4	4	0.4	1.8	5,587
batch-8	8	1.4	3.0	11,131
batch-16	16	3.0	6.1	24,285
batch-32	32	6.4	14.0	52,105
batch-64	64	12.6	28.1	101,633
batch-128	128	24.2	59.2	194,717
batch-256	256	49.8	103.0	399,045
batch-512	512	96.9	217.3	773,125
batch-1024	1024	194.5	446.9	1,544,029
batch-2048	2048	393.6	941.2	3,112,065
batch-4096	4096	795.7	2,000.0	6,258,711

- **repair models**: these are the models that are used on the experiments of the repair scenario. They contain a set of artifacts that violate the constraints, and consequently, the queries return a collection that contains elements where the constraints are violated. On each experiment, repair

[4]Data obtained by measuring the number of objects in the corresponding CDO repository.

models are manipulated and re-checked at ten successive iterations. Table 8.2 describes models of the experiments for the repair scenario.

Table 8.2: Repair scenario models.

	Gen. Factor	XMI Size (MB)	CDO Repo (MB)	# of objects [5]
repair-1	1	0.2	0.8	1,333
repair-2	2	0.3	1.1	2,512
repair-4	4	0.7	1.7	5,398
repair-8	8	1.5	3.2	12,111
repair-16	16	2.9	5.8	23,291
repair-32	32	6.4	13.2	52,102
repair-64	64	12.5	33.9	101,502
repair-128	128	25.4	62.0	204,814
repair-256	256	51.8	107.1	415,985
repair-512	512	100.5	223.2	803,264
repair-1024	1024	200.2	457.9	1,591,524
repair-2048	2048	397.0	950.1	3,144,511
repair-4096	4096	804.5	2,000.0	6,341,453

8.1.4 Queries and Manipulations

The five evaluated queries are proposed by the Train Benchmark Case [Ujh15]. Each query has a different complexity level and they are described following:

- **Q1: PosLength**. Returns all the *Segment* instances that have zero or negative length value. This is one of the simplest queries within the benchmark case and it only performs an attribute check.

- **Q2: RouteSensor**. Searches for *Sensor* instances that are associated with a *Switch*, the *Switch* belongs to a *Route*, and the `Sensor` is associated with a different *Route*. This is one of the type of queries that are commonly used in real case validations for searching broken cycles.

- **Q3: SemaphoreNeighbor**. The query searches *Route* instances that have (i) a exit *Semaphore*; (ii) a *Sensor* connected to another *Sensor* by two different *TrackElement* instances; and (iii) there is not other *Route* that connects the same *Semaphore* and the other *Sensor*. This is the most complex query within the benchmark case.

[5]Data obtained by measuring the number of objects in the corresponding CDO repository.

- **Q4: SwitchSensor**. Returns *Switch* instances that have not any associated *Sensor* instance. This is another simple-query that checks for missing associations of an object.

- **Q5: SwitchSet**. It searches for *Route* instances that reference a *Semaphore* with *GO* signal value, and additionally the *Route* contains a *Switch* instance with a position value that is different of the value specified by the *SwitchPosition* instance that refers to the *Switch*.

Appendix B includes description of these queries using EOL language.

Besides queries, the benchmark case proposes a set of model modifications that are executed in the repair scenario (manipulation phase). These modifications select candidates to be modified from the result of the previously executed query. Each modification repairs an artifact, and the number of elements that violate the constraint decrease. The manipulation phase of the repair scenario uses the fixed transformation strategy with 10 value constant. Thus, each execution of the manipulation phase repairs 10 artifacts, and the result size is decreased in 10 values. Modifications for each query are detailed in Appendix B.

The Train Benchmark Case provides the project that performs such modifications if models are persisted with XMI, MySQL or Neo4J. However, modifications for CDO repositories are not supported. Therefore, projects for performing CDO modifications have been implemented. Modifications are almost the same of the other persistences, and the difference is that as using CDO the model is not loaded in advance, after performing each modification is required to store changes for persisting them. This strategy requires the generation of the model on each independent experiment, and to avoid this action, a temporary model copy is used. The temporary model is a copy of the generated model, and it is automatically created before executing each experiment.

8.2 Results

This sections shows and describes the obtained results, and they have been grouped by the scenario:

8.2.1 Batch Scenario

Three different metrics have been obtained in the experiments that are part of the batch scenario: read time (M1), check time (M2) and result size (M3).

Result size (M3)

M3 metric results are zero for all the experiments, since the constraints specified by the queries (Q1-5) are not violated, and consequently, the queries do not return any result.

Query execution (M1+M2)

Execution time average results for M1 and M2 phases have been included in the Appendix B. Focusing in read phase (M1) results, they indicate that in the case of *XMI+EMF* and *XMI+OCL*, the time required is similar for all the experiments that are executed over same model, and independently of the executed query. Similarly, results show that the query has not influence on the time required for executing the read phase in MySQL+SQL, Neo4J+Core API and Neo4J+Cypher experiments. Read phase requires a similar time for all the experiments over models of the same size. This behaviour changes in the case of both configurations of *XMI+IncQuery* (local and incremental), where the read phase execution time is different depending on the executed query. *SwitchSet* is the query where the read phase is executed faster, and *SemaphoreNeighbor* results are the slowest. *XMI+IncQuery* experiments are not able to execute the *SemaphoreNeighbor* query over the largest model (*batch-4096*).

In the case of *CDO* repositories, the time required by the read phase to be executed is similar for models of the same size and independently of the executed query and query language. For example, read phase is executed in 3-8 seconds for all the *CDO* experiments over the largest model (*batch-4096*). This result is more than four times lower than the best result of the experiments in the rest of persistences (37 seconds in *RouteSensor* with *MySQL+SQL*).

Figure 8.3 illustrates the execution time average required by each query for different query languages and persistences. Illustrated values include the execution time of read phase (M1) and check phase (M2)[6].

[6]Refer to Appendix B for M1 and M2 execution time results

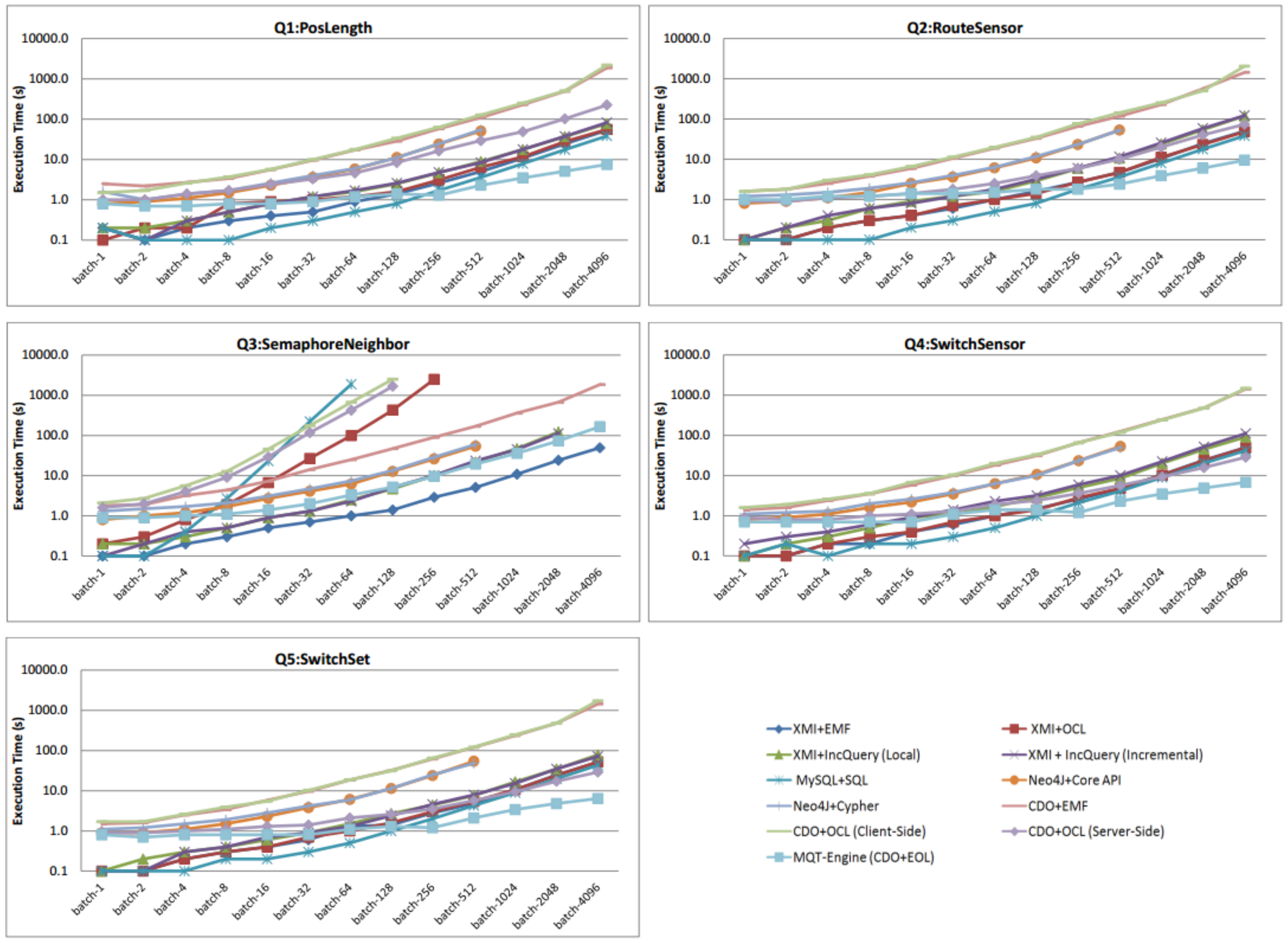


Figure 8.3: Execution time results for queries in the batch scenario.

Q1:Poslength. *MySQL+SQL* is the fastest approach from *batch-2* to *batch-128*. By contrast, from *batch-256* to *batch-4096* *MQT-Engine* is the fastest option. Difference between *MQT-Engine* results and the rest of the options increases as the size of the queried model increases. For example, *MQT-Engine* requires 7.5 seconds on *batch-4096*, and *MySQL+SQL* (the second best option) requires 38 seconds. These options are followed by query languages that are executed over XMI models. Results are similar for *XMI+EMF* and *XMI+OCL* (53-56 seconds for *batch-4096*). *XMI+IncQuery* requires in most of the cases one third more of time than *XMI+OCL*. These options are followed by server-side *CDO+OCL* which requires 225 seconds for the largest model.

The slowest options are *CDO+EMF* and client-side *CDO+OCL*. Results for the largest model increase to 1874 seconds in CDO+EMF, and to 2193 seconds in client-side *CDO+OCL*. *Neo4J+CoreAPI* and *Neo4J+Cypher* is not able to execute *Q1* from *batch-1024* to *batch-4096* (memory out of bound exception). Results for the rest of the model sizes, are better than *CDO+EMF* and client-side *CDO+OCL*, but worse than the rest of the options.

Q2:RouteSensor. Obtained results are similar for *RouteSensor* query. *MySQL+SQL* is the fastest option from *batch-1* to *batch-256* and the second best option from *batch-512* to *batch-4096*. Execution time increases from 0.1 seconds of the smallest model to 38.2 seconds of the largest one. *MQT-Engine* is the second best option from *batch-1* to *batch-256*, and the first one for larger sizes. Execution time in *MQT-Engine* increases from 1 second (*batch-1*) to 9.5 seconds (*batch-4096*). These options are followed by *XMI+EMF* and *XMI+OCL*, and in both cases results are similar: they require around 44 seconds for the largest model. Execution time is higher for local and incremental executions of *XMI+IncQuery*, and in most of the cases they require more than double of the time of *XMI+EMF*. In the case of server-side *CDO+OCL*, results are better than *XMI+IncQuery*, but worse than *XMI+EMF* or *XMI+OCL*.

CDO+EMF and client-side *CDO+OCL* are the options that require more execution time. For the largest model, *CDO+EMF* requires 1438 seconds and client-side *CDO+OCL* 2048 seconds. *Neo4J* experiments fail for models larger than *batch-512* (memory problems). For smaller sizes, results are better than *CDO+EMF* or client-side *CDO+OCL*, but worse than other options.

Q3:SemaphoreNeighbor. The obtained results are different for *SemaphoreNeighbor* query. *XMI+EMF* is the best option for all model sizes, requiring 49 seconds for the largest model. The trend is not the same for *XMI+OCL* where execution time greatly increases as the model size increases. It requires more than 2500 seconds for *batch-256*, and the execution fails for larger sizes (timeout exception). *XMI+IncQuery* results are similar in both configurations (local and incremental), and they are not able to execute query over largest model(timeout exception). *MySQL+SQL* is able to execute query only over models smaller than *batch-128*. In these cases, results are worse than previous queries (almost 1900 seconds for *batch-64*). *Neo4J* experiments are only executed from *batch-1* to *batch-512*, and they require 54-60 seconds for *batch-512*.

In CDO options, only *CDO+EMF* and *MQT-Engine* are able to execute experiments for all sizes. Results in *MQT-Engine* are much better, and it is the second best option after *XMI+EMF* for *batch-128* and larger models. In the largest model, *SemaphoreNeighbor* requires 166 seconds in *MQT-Engine* and 1859 seconds in *CDO+EMF*. Both *CDO+OCL* options only execute experiments from *batch-1* to *batch-128*. For executed model sizes, results are better for server-side *CDO+OCL*: it requires 1690 seconds for *batch-128*, while client-side *CDO+OCL* requires 2523 seconds.

Q4:SwitchSensor. *MySQL+SQL* shows the best results from *batch-1* to *batch-128*. For larger models, it is also one of the three best options. *SwitchSensor* requires 42 seconds for the largest model in *MySQL+SQL*. *MQT-Engine* results for models larger than *batch-128* show best performance. *XMI+EMF*, *XMI+OCL* and server-side *CDO+OCL* results are similar to *MySQL+SQL*. If we compare these four options for largest models, server-side *CDO+OCL* shows best results. For example, for *batch-4096*, server-side *CDO+OCL* requires 28 seconds, *MySQL+SQL* 42 seconds, *XMI+OCL* 49 seconds and *XMI+EMF* 50 seconds . Execution times are higher in *XMI+IncQuery*, where local configuration shows better performance results than the incremental configuration. For the largest model, local configuration resolves the query in 91 seconds, and incremental configuration in 112 seconds.

Worst results are shown by *CDO+EMF* and client-side *CDO+OCL*. Results are similar and they require more than 1420 seconds for executing Q4 over *batch-4096*. *Neo4J+Core API* and *Neo4J+Cypher* experiments are not able to execute the query over *batch-1024* and larger models. Both approaches show similar results (around 50 seconds for largest model) that are better than *CDO+EMF* or client-side *CDO+OCL*, but worse than other approaches.

Q5:SwitchSet . *MySQL+SQL* is the fastest option from *batch-1* to *batch-128*. *MQT-Engine* shows best performance results for larger sizes, and it requires only 6.4 seconds in *batch-4096*. Server-side CDO+OCL is the second best option for largest models, and it requires 29 seconds for *batch-4096*. It is followed by *MySQL+SQL* which requires 43 seconds for querying the largest model. *XMI+EMF* and *XMI+OCL* results are close to *MySQL+SQL*. Both approaches show similar results and they require around 47 seconds for querying *batch-4096*. Performance results are higher in the case of *XMI+IncQuery* experiments. Both local and incremental configuration results are similar, and they require around 75 seconds for *batch-4096*.

CDO+EMF and client-side execution of *CDO+OCL* are the slowest options, and *CDO+EMF* results are better than *CDO+OCL*. For example, largest model is queried in 1448 seconds with *CDO+EMF* and in 1732 seconds in the client-side *CDO+OCL*. As occurs on previous queries, *Neo4J* experiments are not executed for *batch-1024* and larger models. Both configurations (*Neo4J+Core API* and *Neo4J+Cypher*) show a similar trend, and they require around 50 seconds for executing the query over *batch-512*.

8.2.2 Repair Scenario

Five different metrics have been obtained in the experiments that are part of the batch scenario: read time (M1), check time (M2), check result size (M3), recheck time (M4), and recheck-result size (M5).

Check and Recheck result size (M3 and M5)

M3 returns the number of artifacts that violate the constraint that is specified by the executed query. Table 8.3 illustrates the size of the results returned by each query for each model of the repair scenario.

M5 returns the number of results returned by the re-execution of the query after performing a set of modifications. The size of the result obtained after each recheck execution corresponds with *previous_value - 10*. Hence, as each experiment executes the recheck phase ten times, the last result size value is the first result size (specified in Table 8.3) *minus* 100.

Table 8.3: Results obtained for each query at check phase in the repair scenario.

model	Q1	Q2	Q3	Q4	Q5
repair-1	92	7	1	3	2
repair-2	196	9	1	4	4
repair-4	411	26	0	5	11
repair-8	914	44	5	12	24
repair-16	1,769	70	8	24	52
repair-32	4,068	149	13	57	144
repair-64	7,853	311	20	126	287
repair-128	15,914	638	38	259	590
repair-256	32,168	1,248	72	516	1,217
repair-512	62,217	2,398	147	987	2,321
repair-1024	123,682	4,683	307	1,915	4,544
repair-2048	244,750	9,311	578	3,826	8,923
repair-4096	493,877	18,779	1,175	7,707	17,934

Query execution (M1+M2)

The time required by the experiments for executing the read phase (*M1*) over CDO models of the same size is similar in the different query languages. The execution time values are low (around 4 seconds for the largest model). Main reason for low results is that in the read phase in CDO approaches only encompasses the connection with the repository, and it is not required to load models in-memory.

Read phase execution times are higher in the XMI approaches, since they load in-memory the queried model. Results show that the query has impact in the read phase in local and incremental *XMI+IncQuery*. For example, in the case of local *XMI+IncQuery* over largest model, read phase requires around 68 seconds in *PosLength*, and 449 seconds in RouteSensor.

Figure 8.4 illustrates the execution time results that corresponds with the sum of the time obtained in read (*M1*) and check (*M2*) phases. Each chart of the figure shows the results for a different query (*Q1-Q5*). Additional tables that show average results for each experiment are included in Appendix B.

Q1:PosLength. *XMI+EMF*, *XMI+OCL* and *MySQL+SQL* are the options that show best results for *Q1*. Although results are similar for three approaches, MySQL+SQL is the best for *repair-512* and smaller models, and *XMI+EMF* for models larger than *repair-512*. In *repair-4096*, XMI+EMF requires less than 45 seconds, *MySQL+SQL* 45 seconds and *XMI+OCL* 49 seconds. The next fastest options are local and incremental configurations of *XMI+IncQuery* and *MQTEngine*, which show slightly higher results. *MQT-Engine* is faster than

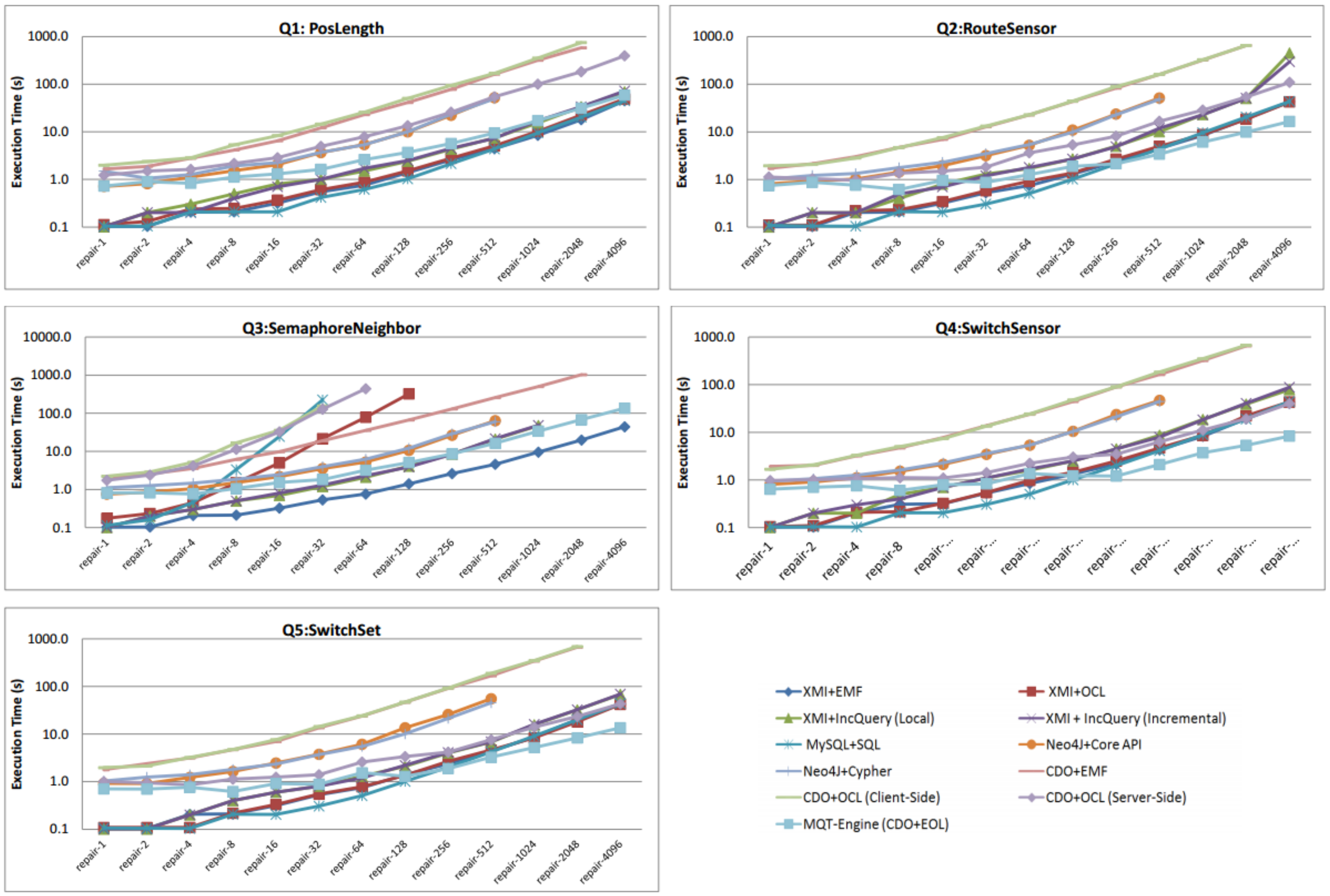


Figure 8.4: Query execution time (M1+M2) in the repair scenario.

XMI+IncQuery for the two largest models. For example, in *repair-4096* MQT-Engine requires 59 seconds, local *XMI+IncQuery* 68 seconds and incremental *XMI+IncQuery* 72 seconds.

Client-side *CDO+OCL* is the option that shows worst results, and it is not able to execute query over largest model (timeout exception). In the case of *repair-2048* it requires 747 seconds. Although they are somewhat better, *CDO+EMF* results are also high. *CDO+EMF* does not execute *Q1* over the largest model, and it requires 579 seconds for *repair-2048*. Server-side *CDO+OCL*, *Neo4J+Core API* and *Neo4J+Cypher* results are lower than in *CDO+EMF*. Three approaches show similar results for *repair-512* and smaller models. Both *Neo4J* options are not able to execute the query over models larger than *repair-512* (memory exception). Server-side *CDO+OCL* is able to execute the query for all sizes, and it requires 394 seconds for the largest one.

Q2:RouteSensor. *MySQL+SQL* shows best results from *repair-1* to *repair-128*, and results are similar for *XMI+EMF* and *XMI+OCL*. However, MQT-Engine is the best option for larger models. In the case of *repair-4096*, MQT-Engine requires 16 seconds, *XMI+OCL* 42 seconds, and *XMI+EMF* and *MySQL+SQL* 43 seconds. Obtained results are higher at the incremental *XMI+IncQuery* (e.g. almost 300 seconds for *repair-4096*). Although similar, local *XMI+IncQuery* shows higher result for most sizes (e.g. 450 seconds for *repair-4096*). In most of the cases, server-side *CDO+OCL* shows results slightly higher than *XMI+IncQuery* options, however for largest models results are much better (109 seconds).

CDO+EMF and *CDO+OCL* are the options that show worst results. They are not able to execute *Q2* over the largest model , and they require around 650 seconds for *repair-2048*. Both *Neo4J* options show similar results, and they are not the worst for *repair-512* and smaller models. However, *Neo4J* experiments are not executed over *repair-1024* and larger models.

Q3:SemaphoreNeighbor. *XMI+EMF* is the best option, and it requires 44 seconds in the largest model. Local and incremental XMI+IncQuery are the next best options from *repair-1* to *repair-1024*. However, *IncQuery* experiments are not able to execute *Q3* over *repair-2048* and *repair-4096*. *XMI+EMF* and *MQT-Engine* are the only options able to execute query over all sizes. MQT-Engine is the second option for the two largest models, requiring 136 seconds for the largest model. *CDO+EMF* executes query over all model sizes, with the exception of the largest one. *CDO+EMF* executes *Q3* query in 1024 seconds over *repair-2048*. *Neo4J+Core API* and *Neo4J+Cypher* are not able to execute the query from *repair-1024* to *repair-4096* (memory exception). Both approaches show similar results, and for *repair-512*, *Neo4J+Cypher* requires 60 seconds and *Neo4J+Core API* 63 seconds.

MySQL+SQL and Client-side *CDO+OCL* shows worst results: the query is successfully executed from *repair-1* to *repair-32*. For querying *repair-32*, client-side *CDO+OCL* requires 156 seconds and *MySQL+SQL* 323 seconds. Results are slightly better in server-side *CDO+OCL*, and it is able to execute query over *repair-64* in 439 seconds. *XMI+OCL* is only able to execute the query over *repair-128* (requiring 323 seconds) or smaller models;

Q4:SwitchSensor. *MySQL+SQL* is the fastest option from *repair-1* to *repair-128* and *MQT-Engine* is the fastest from *repair-256* to *repair-4096*. *MQT-Engine* resolves *Q4* over the largest model in 8 seconds. Results for the largest models in *MySQL+SQL* are similar to the results obtained with *XMI+EMF* and *XMI+OCL*, and they require around 44 seconds for *repair-4096*. Although execution time results are slightly higher from *repair-1* to *repair-1024*, server-side *CDO+OCL* shows better results for *repair-2048* (19 seconds) and *repair-4096* (40 seconds). *XMI+IncQuery* results are higher and both configurations (local and incremental) show similar results: local *XMI+IncQuery* requires almost 79 seconds over largest model; and incremental *XMI+IncQuery* 88 seconds.

CDO+EMF and client-side *CDO+OCL* are the worst options, and they are not able to execute query over the largest model. *CDO+EMF* resolves the query in almost 653 seconds over *repair-2048* and client-side *CDO+OCL* in almost 672 seconds. From *repair-1* to *repair-512* *Neo4J+Core API* and *Neo4J+Cypher* results are better than *CDO+EMF* or client-side *CDO+OCL*. However, *Neo4J* options are not able to execute the query over *repair-1024* and larger models.

Q5:SwitchSet. *MySQL+SQL* is the fastest option from *repair-1* to *repair-128*. However, *MQT-Engine* is the fastest solution for larger models. *MQT-Engine* resolves the query in less than 14 seconds over *repair-4096*, and *MySQL+SQL* requires almost 44 seconds. *XMI+EMF* and *XMI+OCL* results are few milliseconds higher than *MySQL+SQL* results for *repair-512* and smaller models. However, query is resolved faster in larger models using *XMI+EMF* or *XMI+OCL*: around 42 seconds for *repair-4096*. Server-side execution of *CDO+OCL* is the following faster one, and it requires 43 seconds for *repair-*

4096. *XMI+IncQuery* results are higher than in the previously described options. Both local and incremental configurations show similar results, and they require around 68 seconds for the largest model.

CDO+EMF and client-side *CDO+OCL* show similar results, and they are the slowest options. They are not able to execute query over the largest model, and *CDO+EMF* requires 668 seconds for *repair-2048* and client-side *CDO+OCL* 687 seconds. Results in *Neo4J* options are better than *CDO+EMF* or client-side *CDO+OCL* for *repair-512* (around 50 seconds) and smaller models. However, they are not able to execute *Q5* over *repair-1024* and larger models.

Query re-execution (M4)

The experiments that are part of the repair scenario also obtain metrics that describe the execution time for re-executing a query after performing a model manipulation (re-check phase). Figure 8.5 illustrates the execution time required by the experiments for executing re-check phase (M4). Each chart of the figure shows the results for a different query (*Q1-Q5*). Additionally, tables that show average results for each experiment are included in Appendix B.

Q1:PosLength. *XMI+IncQuery* (incremental) only requires 1 millisecond for rechecking all model sizes. It is followed by *XMI+IncQuery* (local) which requires up to 100 milliseconds for the largest model. *MySQL+SQL* executes the re-check in less than a second for all sizes. *Neo4J* options require less than a second in the executed model sizes. However, *Neo4J* experiments are not executed for *repair-256* and larger models. Re-execution results increase in the case of *XMI+EMF* or *XMI+OCL*. *XMI+EMF* performs it over the largest model in 2 seconds and *XMI+OCL* in almost 4 seconds. All the CDO-related options require more time for executing the re-check. *MQT-Engine* is the fastest option for re-executing the query over CDO models, and it requires 6 seconds in the largest one. Next options are client-side *CDO+OCL* and *CDO+EMF*, however execution for *repair-4096* is not performed successfully. Server-side *CDO+OCL* shows higher results, but it is able to execute query over all sizes (44 seconds in the largest model).

Q2:RouteSensor. The result scenario is similar to the scenario described at Q1. *XMI+IncQuery* options, *MySQL+SQL* and *Neo4J* options require less than a second for re-checking query over all model sizes. However, in *Neo4J* options, experiments for *repair-256* and larger models are not executed. Re-execution

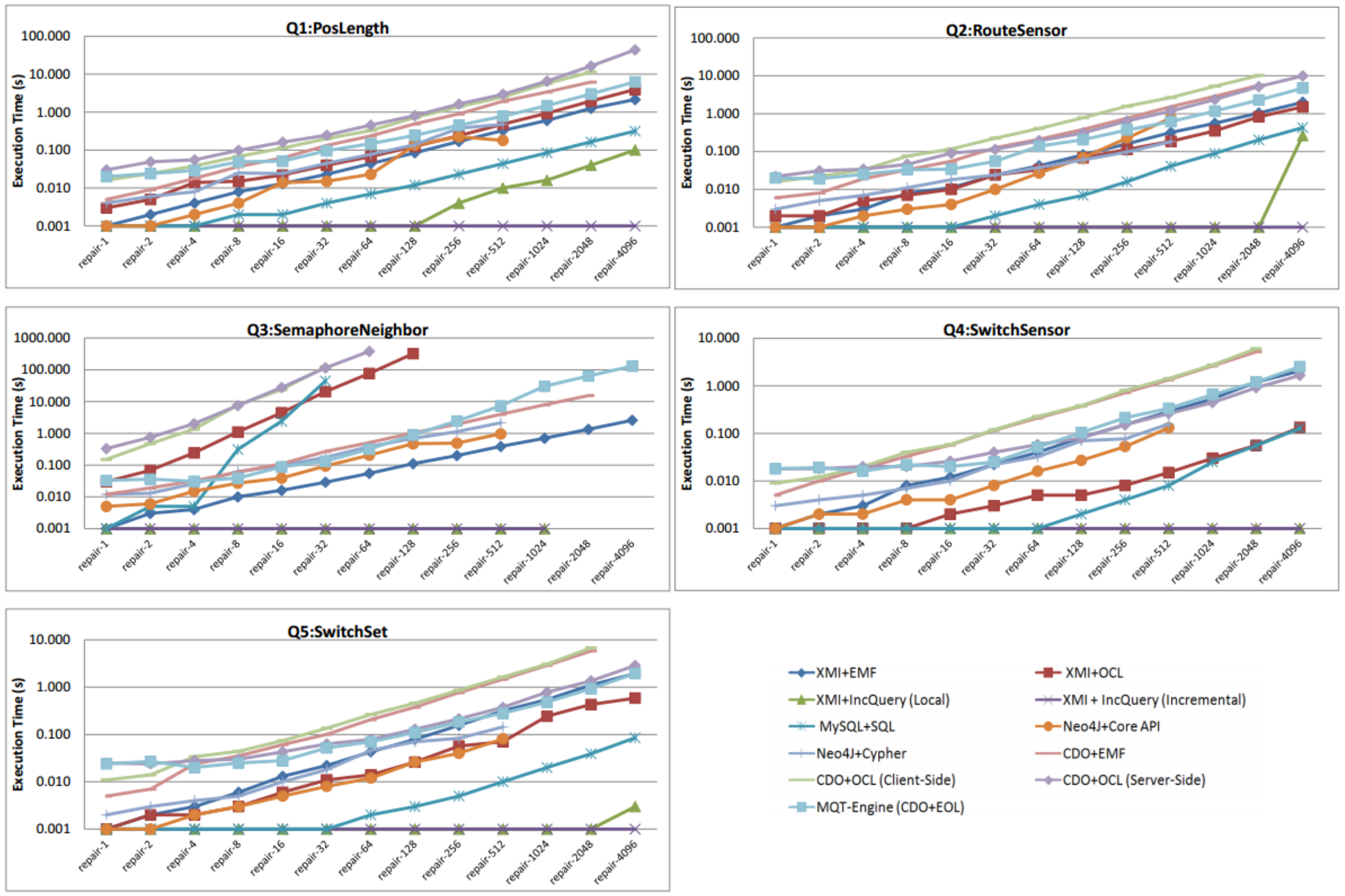


Figure 8.5: Query re-execution time (M4) in the repair scenario.

results increase in the case of *XMI+EMF* or *XMI+OCL*. *XMI+EMF* requires 2 seconds for the largest model, and *XMI+OCL* requires less than 2 seconds. CDO-related options require more time for executing the re-check, and *MQT-Engine* is the fastest option over CDO models (less than 5 seconds for largest model). *CDO+EMF* and server-side *CDO+OCL* are the next option. *CDO+EMF* does not execute experiments for largest model, and server-side *CDO+OCL* requires less than 10 seconds. Client-side *CDO+OCL* shows slower results and it does not execute re-check over the largest model.

Q3:SemaphoreNeighbor. The recheck is executed in less than a few seconds in *XMI+EMF*, *XMI+OCL*, *XMI+IncQuery* (local and incremental) and *Neo4J* (Core API and Cypher). From those options, only *XMI+EMF* executes the re-check phase for all model sizes. In the case of *MySQL+SQL*, model size has great impact and it requires more than 40 seconds for executing the query over *repair-32*. The re-check is not executed for larger models in *MySQL+SQL*. In CDO-related options, *CDO+EMF* is the option that shows better results (15 seconds for *repair-2048*). However, re-check is not executed for largest model in *CDO+EMF*. *MQT-Engine* is able to execute the recheck over all models and it requires 130 seconds in *repair-4096*. Model size has greater impact over both *CDO+OCL* options, and they requires more time for executing re-check. In the case of *repair-32* they require more than 115 seconds.

Q4:SwitchSensor. *XMI+OCL*, *XMI+IncQuery* (local and incremental) and *MySQL+SQL* require less than a second for executing the re-check over all model sizes. *Neo4J+Core API* and *Neo4J+Cypher* executed experiments show results that are lower than a second. However, they do not execute the re-check from *repair-1024* to *repair-4096* (memory exception in a previous phase). Required time is increased in *XMI+EMF*, however it only requires around 2 seconds for executing the re-check over *repair-4096*. CDO options require more time than the rest of the options. Server-side execution of *CDO+OCL* is the fastest option (almost 2 seconds for *repair-4096*) and it is followed by *MQT-Engine* (less than 3 seconds). *CDO+EMF* and client-side *CDO+OCL* show similar results, and both options execute the recheck phase for all models except for *repair-4096*. *Repair-2048* requires less than 6 seconds in *CDO+EMF* and less than 7 seconds in client-side *CDO+OCL*.

Q5:SwitchSet . The recheck phase is resolved in less than a second for *XMI+OCL*, *XMI+IncQuery* (local and incremental) and *MySQL+SQL* in all model sizes. *Neo4J* experiments are not executed from *repair-1024* to *repair-4096*. The experiments executed over smaller model sizes show results that are lower than a second. Obtained results are increased in *XMI+EMF*, however it only requires around 2 seconds for executing the recheck over *repair-4096*. In the case of the CDO options, they require more time than the previously described options. Re-check is resolved faster by MQT-Engine (around 2 seconds for *repair-4096*), and it is followed by the server-side execution of CDO+OCL (3 seconds for *repair-4096*). CDO+EMF and client-side CDO+OCL are higher, and both options show similar results and execute the recheck phase for all models except for *repair-4096*.

8.2.3 Threads to Validity

The evaluation framework provides results in terms of matches (number of solutions produced by each query) and in terms of performance (execution time). However, the memory usage metrics have not been obtained. Reasons for this omission is that the memory metrics obtained with the Train Benchmark Case framework where not accurate.

Moreover, the results of these experiments correspond with the batch and repair scenarios proposed at the case benchmark. By contrast, experiments related to the inject scenario are not included due to problems for executing modifications over CDO repositories.

8.2.4 Conclusions

Focusing on the experiments that are part of the batch scenario, can be concluded that *MQT-Engine* is the option that shows better execution results for most of the queries (*PosLength*, *RouteSensor*, *SwitchSensor* and *SwitchSet*) over the models that are larger than *batch-128*. In the case of smaller models, *MySQL+SQL* is the fastest option. Regarding *SemaphoreNeighbor*, *XMI+EMF* is the fastest one for all the model sizes. In the largest models, *XMI+EMF* is followed by *MQT-Engine*. Is important to note that only three approaches are able to execute *SemaphoreNeighbor* over all model sizes: *XMI+EMF*, *XMI+OCL*, and *MQT-Engine*. In the case of *Neo4J* query languages, they fail for memory reasons over models larger than *batch-512*.

In the case of the repair scenario experiments, *XMI+EMF* is the fastest option to execute read+check phases in *PosLength* and *SemaphoreNeighbor* queries. By contrast, *RouteSensor*, *SwitchSensor* and *SwitchSet* are performed faster using *MQT-Engine*. Comparing these results with the results obtained in the batch scenario, it can be concluded that the query result size has impact over the execution time in the *PosLength* query. This query returns a quantity of results much higher than other queries. Moreover, results show that CDO approaches require much less time for the read phase, since they do not require to load in-memory models. The trend for *Neo4J* options in the repair scenario is the same of the batch scenario: both options fail for memory reasons for models that are larger than *repair-512*.

Focusing in the recheck phase, the times are much lower for the options that query models persisted with XMI. The reason for this is that the information is already loaded in memory. The best results for the recheck phase are shown by *XMI+IncQuery* options. *IncQuery* is focused on the incremental changes of models, and consequently, it is able to resolve queries faster than other approaches. In the case of CDO options, the results show a faster re-execution of the query for *CDO+EMF* and both executions of *CDO+OCL*. The reason is that the information required by the query has been loaded in the check phase execution. However, this does not occur with *MQT-Engine*, and it has to re-execute the query again over the database. In this case, the execution times are decreased with respect to the first execution of the check phase. Is important to note, that the recheck phase is not executed by some options for the largest model sizes (Neo4J options, CDO+EMF and CDO+OCL in the client-side), and the reason for this is that these options had not been able to execute the previous phases correctly for the corresponding model size.

Obtained results address the previously described questions:

- *Question 1: Which is the performance of MQT-Engine for executing Train Benchmark Case queries over CDO repositories and using EOL?* Results show that *MQT-Engine* is able to successfully execute EOL queries in batch and repair scenarios, and for all the evaluated model sizes.

- *Question 2: How are the results in comparison to the execution time required by other query languages over CDO repositories?* Experiments show that *MQT-Engine* performs better than *CDO+EMF* and *CDO+OCL* (executed in the client- and server-side) for the first execution of the query. Results that correspond with the re-execution of the query in *MQT-Engine* are also better than *CDO+EMF* and *CDO+OCL*, with the exception of *SemaphoreNeighbor* where *CDO+EMF* performs the re-execution faster. However, *MQT-Engine* is the only CDO-based approach that is able to execute the queries over all the evaluated model sizes.

- *Question 3: How are MQT-Engine results in comparison to other query languages and persistences?* The results are different depending on the executed scenario and phase. For example, in the read phase, *MQT-Engine* is one of the fastest approaches for largest models since it only connects to the CDO repository, and does not require to load models in-memory as occurs on non-CDO options. If we focus in the first query execution (which encompasses read and check phases) in the batch scenario, *MQT-Engine* is the first option for most of the queries in the largest models, with the exception of *SemaphoreNeighbor* where *XMI+EMF* shows better results. For small sizes, *MySQL+SQL* is the best option in most of the experiments and it is followed by *MQT-Engine*. Although is not the best for all queries, *MQT-Engine* is one of the fastest options for first execution of the query in the experiments of the repair scenario. However, in the re-execution of the query, queries executed over XMI are resolved faster than in *MQT-Engine*. Main reason is that the information is already loaded in memory. In the case of *XMI+IncQuery*, incremental execution is provided and *XMI+IncQuery* experiments are resolved much faster than the others. The re-execution in *MQT-Engine* is not incremental. Therefore, the required time is slightly lower, but similar to the first query execution.

Part IV

Conclusion

9 Conclusion

This chapter concludes this dissertation providing conclusions of the performed work: first general conclusions are provided; then, the previously formulated hypotheses are validated; Next, MQT-Engine limitations are identified. This chapter continues providing a comparison of the approach with the existing work, and describing future work. The chapter ends with a list of lessons learned during the design, implementation and evaluation phases of this dissertation.

9.1 Conclusions

Model query languages are closer to model engineers, and they are more appropriate than persistence-specific query languages for them. Model query languages are expressed in languages focused on interacting with models, independently of the persistence mechanism. Persistence-specific query languages show better performance and memory usage results when querying models persisted in databases. However, in persistence-specific query languages, engineers should be aware of the way information is persisted, and learn database specific concepts and languages. Differences between model query languages and persistence-specific query languages motivated the work presented at this dissertation.

Preliminary prototypes were presented at [Car15a, Car16a]. These prototypes support transformation of EOL queries into SQL queries. Then, generated queries are executed over an ad-hoc persistence. This persistence uses relational databases with a metamodel-agnostic schema for persisting EMF models.

After validating the feasibility of transforming queries from a model query language into a persistence-specific query language, MQT-Engine framework was designed and implemented. This framework provides model engineers the ability to use a model query language with the efficiency of a persistence-specific query language. MQT-Engine transforms queries expressed with a model query language into queries expressed with a persistence-specific query language, and then executes them over the persistence and at database-side.

MQT-Engine has been designed with extensibility in mind, and it provides a set of classes to be extended by components that provide support for alternative query languages in the framework. The framework provides QLI Metamodel which is query language-agnostic and separates model query and persistence-specific query languages during transformation. Additionally, it facilitates inclusion of new languages within the framework.

In this dissertation, a prototype of the framework is provided. It supports transformation of EOL queries into SQL queries, and then executes them over a relational CDO repository containing models to be queried. The prototype has been evaluated using two different use cases: one is based on the reverse engineering domain, and the other is based on the railway domain.

Results of the evaluation show that MQT-Engine framework is able to transform queries and execute them over the CDO repository successfully. Moreover, it is one of the solutions that has shown better results when querying models persisted with CDO. If we compare MQT-Engine results with results of other evaluated query languages, they are not always the best, but they are one of the solutions with better results in the first execution of the queries.

9.1.1 Hypothesis Validation

Two hypotheses were formulated for this dissertation, and they have been validated with the design, implementation and evaluation of MQT-Engine framework:

Hypothesis A argues that *if models are persisted using a database, querying them using a database-specific query language is more efficient in terms of performance and memory usage*. Both experimental evaluations have provided performance results that indicate that, in the CDO case, queries are resolved faster using SQL, which is a persistence-specific query language of relational databases. Moreover, reverse engineering benchmark case has provided memory usage results that show that using SQL less memory is required for executing the evaluated queries.

In the case of the first case study, we had distinguished two configuration factors: F1 and F2. F1 has evaluated the impact of the increasing size of the model. Execution time difference between SQL (and MQT-Engine) and model query languages increases with model size, and results indicate that SQL is up to 162 times faster than a model query language (Q3 using Plain EMF and over Set4). The same occurs with memory usage: SQL (and MQT-Engine) requires up to 23 times less memory than model query languages executed at client-side; and more than 2 times less memory than server-side executed OCL. F2 evaluated the impact of the increasing size of the repository. Results show that execution time and memory usage difference between SQL (including MQT-Engine) and model query languages executed at server-side is reduced as the size of the repository increases. However, for the largest repository, SQL is more than 40 times faster than model query languages executed at client-side, and requires up to 10 times less memory. SQL-based solutions are up to 8 times faster than OCL at server-side and requiring up to four times less memory.

In the second case study, MQT-Engine using SQL is one of the evaluated solutions showing best results for all queries and in largest model sizes. MQT-Engine shows best results for queries executed over CDO repositories, and it is the unique CDO solution able to execute all queries over all model sizes. However, this first hypothesis is not always satisfied when a query is re-executed consecutive times. Some experiments of the second case study have shown that in this case, having the information loaded in memory is more efficient in terms of re-execution time.

Hypothesis B argues that *transformation of queries from a model query language to a persistence-specific query language provides model engineers the ability to query models using a language that is closer to their knowledge, but with the efficiency of a persistence-specific query language.* The MQT-Engine prototype validates this second fact, since it provides support for transforming EOL, a model query language, into SQL, a persistence-specific query language. Generated SQL queries are executed at server-side and over a relational CDO repository where queried models are persisted. Two experimental evaluations have shown that MQT-Engine is able to resolve queries in all the executed experiments and moreover, they have shown good results in terms of performance.

Results obtained during the MQT-Engine evaluation using the reverse engineering case study show that the execution time difference between MQT-Engine and SQL is two seconds or less. By contrast, MQT-Engine has shown memory usage results that are better than SQL. This difference in execution time and memory usage between SQL and MQT-Engine proves that the query transformation overload is constant and small.

9.1.2 Limitations

We have developed a framework that transforms and executes model queries. However, different limitations have been identified in the proposed approach, and they are discussed below:

Support for other types of queries. MQT-Engine framework is able to transform queries that fully traverse models. This type of queries start the computation by obtaining all the instances of a specific type that exists within the queried model, and covers the majority of computational-demanding queries in domains such as reverse engineering [Góm15b] or railway. However, there are other types of queries (e.g. non-traversal queries or queries that modify the model) that are not yet supported by the approach.

Support for other query languages. Although the design of MQT-Engine facilitates the inclusion of new model query languages and persistence-specific query languages, they have to be manually implemented. This requires the developer knowing about the query language to be included and about the QLI Metamodel in order to implement the mapping between them.

Support for other persistences. The evaluated MQT-Engine prototype supports querying models persisted only in relational CDO repositories using DBStore.

9.2 Comparison with other approaches

Model persistence. Several approaches provide model persistence facilities by leveraging database back-ends. Each approach persists models using a different back-end: Morsa [EP13b], MongoEMF [Hun14], NeoEMF/Graph [Ben14] and EMF Fragments [Sch13] use NoSQL database back-ends; Teneo [Ten12] uses relational databases for persistence; and CDO [Cdo16] supports persistence in both types (relational and NoSQL) of back-ends. All these approaches are focused on the persistence of models using databases. On the contrary, this dissertation is not focused directly on the persistence of models, and it provides a query transformation and execution mechanism for queries expressed with model query languages.

Query transformation. Model query transformation has been the focus of other works, but most of the identified approaches support transformation of model queries for UML models [Mar99, Hei07, Cab07, Win08, AJ08, Dem09, Ege10, Ori15, Kal16]. By contrast, MQT-Engine is a solution that supports transformation of queries for EMF models.

EMF model query transformation is also supported in [Ber14] and [Dem09], but they focus on transformation of OCL queries. At this stage, MQT-Engine framework provides implementation that transforms EOL queries. Additionally, it provides a set of classes with the aim of facilitating inclusion of new model query languages.

In [Bar14] EOL queries are transformed into SQL queries. Generated queries are specific for a concrete data-schema and persistence. In the case of MQT-Engine prototype, generated queries are for the data-schema that is automatically generated by DBStore in CDO. This way, it provides support for transforming queries of different domains. As occurs with model query languages, the design of MQT-Engine aims to facilitate the inclusion of new persistences and persistence-specific query languages using other mechanisms and schemas for storing information.

Query optimization. EMF-IncQuery provides a model query language that can be executed incrementally. This way, re-execution of queries only implies to query and operate model parts that have changed [Ujh15]. [Wei15] presents an approach which improves the efficiency of model traversal queries expressed with EOL. These two approaches provide different mechanisms for improving query execution but on the user-side. By the contrary, MQT-Engine framework transforms queries into a persistence-specific query language, and generated queries are directly executed over persistence and at server-side.

9.2.1 MQT-Engine Classification

MQT-Engine is classified as follows, using the factors specified at Chapter 2 and Chapter 3. Table 9.1 shows the values of classification factors for query languages in EOL and using MQT-Engine prototype. EOL is an imperative model query language that is supported by EMF-based modelling tools. Main difference between native EOL and EOL with MQT-Engine is that in the latter one, queries are executed directly over the persistence. However, the presented prototype of MQT-Engine does not support modification EOL queries.

Table 9.1: Query language classification for EOL+MQT-Engine.

	Execution	**Abstraction-Level**	**Type**	**Modif.**	**Supported by**
EOL+MQT-Engine	Persistence	MQL	Imperative	No	EMF-based

Next, MQT-Engine is classified using the classification factors for query transformation approaches in Chapter 3:

- **Model Type:** The approach has been designed for EMF models.
- **Input Language:** MQT-Engine aims to support transformation and execution of queries expressed with different model query languages. The prototype of the approach presented at this dissertation takes as inputs queries expressed with EOL.
- **Intermediate Results:** MQT-Engine creates a QLI Model during the query transformation process.
- **Output:** MQT-Engine targets generation of queries in different persistence-specific query languages for different persistences. But the prototype presented at this dissertation generates SQL queries.
- **Target Persistence:** As described in the previous point, the approach aims to support different persistences for EMF models. This version of the

approach supports execution of generated queries over CDO repositories configured with DBStore.

- **Supports mapping?** No. MQT-Engine focuses on query transformation and execution. However, different components have been implemented in both evaluation cases in order to obtain models persisted with CDO.

- **Incremental execution?** No. Generated SQL queries are directly executed over the persistence on each query execution.

- **Lazy Execution?** No. MQT-Engine does not provide lazy objects and collections. However, MQT-Engine avoids loading intermediate results when querying models, and it only obtains from persistence required results.

- **Evaluation?** Yes. MQT-Engine has been evaluated with two different case studies: reverse engineering case and train benchmark case. In the first evaluation case, MQT-Engine has been compared with Plain EMF, OCL (executed at client- and server-side) and SQL queries executed over CDO repositories. Second case has compared performance results of different query languages (Plain EMF, OCL, IncQuery, Neo4J Core API, Cypher, EOL) over different persistences (XMI, MySQL, Neo4J, CDO).

9.3 Future Work

A set of tasks that could extend and improve MQT-Engine Framework proposed at this dissertation have been identified:

Re-execution of generated queries. Results obtained in the experimental evaluation based on the Train Benchmark Case show that the generated SQL queries require similar time in the first and in the posterior execution of the queries. This is not the case of some of the evaluated client-side solutions, where the posterior executions require less time. Therefore, a future task is the optimization of the generated query for reducing time required by queries when they are executed more several times.

Support other types of queries. Support transformation and execution of other types of queries: queries that do not traverse models, modification queries, etc.

Support for other model query languages. Provide MQT-Engine implementations for additional model query languages (e.g. IncQuery or OCL). Each implementation will support mapping between the corresponding model query language and QLI Models that conform QLI Metamodel provided by MQT-Engine. Moreover, the approach will format the results into the data-types expected by the query in the model query language.

Support generation of SQL queries for other data-schemas. Extend MQT-Engine Framework with support for generation of SQL queries for relational databases using a different data-schema.

Support for other persistence-specific query languages. Provide MQT-Engine implementations for generating queries using query languages that are specific for other persistence back-ends. These implementations will encompass also the execution of the generated query over the corresponding persistence.

Extend evaluation. Experimental evaluation results have shown that the domain has impact over the performance of queries. Therefore, it would be interesting to extend MQT-Engine evaluation with other domains, and evaluating the approach with scenarios having different types of models and queries.

9.4 Lessons Learned

These are the main lessons learned during these years of research:

Importance of querying mechanisms. Most of the identified approaches that are focused on operating large models provide alternative persistence mechanisms for models that are database-based. However, experimental evaluation results of this dissertation show that using the correct mechanism for querying models is also a key factor that has impact over the performance of a persistence mechanism.

Relational databases for persistence. Most recent approaches promote the use of NoSQL databases for storing any type of information, which includes persistence of models. If we focus in the size of the models, most of the domains generate models with large size in terms of readability, but not large enough in terms of scalability. In these cases, using a well-structured relational database for model persistence is enough. For example, relational databases are the most mature and widely used back-end in CDO, which is one of the most used persistence approach.

Performance of XMI. In the cases where the memory is not limited, using native XMI persistence could be the solution that provides better results in terms of performance.

There is not an optimal solution for all cases. Results obtained in this dissertation, and also results obtained by other studies show that each persistence and query approach is more appropriate for a different domain and scenario. It is very complicated to provide one solution that is the best for all domains.

Part V

Bibliography and Appendix

Bibliography

[AJ08] Harith T. Al-Jumaily, Dolores Cuadra, and Paloma Martínez. OCL2Trigger: Deriving active mechanisms for relational databases using model-driven architecture. *Journal of Systems and Software*, vol. 81(12):pp. 2299–2314, 2008.

[Atk03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Softw*, vol. 20(5):pp. 36–41, Sep 2003.

[Atl15] AtlanMod. NeoEMF Graph. `http://neo4emf.com/wiki.html`, 2015. Online. Accessed January 27, 2016.

[Bag14] Alessandra Bagnato, Etienne Brosse, Andrey Sadovykh, Pedro Maló, Salvador Trujillo, Xabier Mendialdua, and Xabier De Carlos. Flexible and Scalable Modelling in the MONDO Project: Industrial Case Studies. In *Proceedings of the 3rd Workshop on Extreme Modeling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, XM@MoDELS 2014, Valencia, Spain, September 29, 2014.*, pp. 42–51. 2014.

[Bar12] Konstantinos Barmpis and Dimitrios S. Kolovos. Comparative Analysis of Data Persistence Technologies for Large-scale Models. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pp. 33–38. ACM, New York, NY, USA, 2012.

[Bar13] Konstantinos Barmpis and Dimitris Kolovos. Hawk: Towards a Scalable Model Indexing Architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pp. 6:1–6:9. ACM, New York, NY, USA, 2013.

[Bar14] Konstantinos Barmpis and Dimitris Kolovos. Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models. *Journal of Object Technology*, vol. 13(3):pp. 3:1–26, Jul 2014.

[Ben14] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, a Scalable Persistence Layer for EMF Models. In Jordi Cabot and Julia Rubin, editors, *Modelling Foundations and Applications*, vol. 8569 of *Lecture Notes in Computer Science*, pp. 230–241. Springer International Publishing, 2014.

[Ber10] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over EMF models. In *Model Driven Engineering Languages and Systems*, pp. 76–90. Springer, 2010.

[Ber14] Gábor Bergmann. Translating ocl to graph patterns. In *Model-Driven Engineering Languages and Systems*, pp. 670–686. Springer, 2014.

[Béz05] Jean Bézivin. On the unification power of models. *Software and System Modeling*, vol. 4(2):pp. 171–188, 2005.

[Brä07] Matthias Bräuer and Birgit Demuth. *Model-level integration of the OCL standard library using a pivot model with generics support*. Springer, 2007.

[Cab07] Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 547–548. ACM, 2007.

[Car14a] Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. MQT, an Approach for Run-Time Query Translation: From EOL to SQL. In *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, pp. 13–22. 2014.

[Car14b] Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. Scalable Model Edition, Query and Version Control Through Embedded

Database Persistence. In *Joint Proceedings of MODELS 2014 Poster Session and the ACM Student Research Competition (SRC) co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 28 - October 3, 2014.*, pp. 11–15. 2014.

[Car15a] Xabier De Carlos, Goiuria Sagardui, Aitor Murguzur, Salvador Trujillo, and Xabier Mendialdua. Model Query Translator - A Model-level Query Approach for Large-scale Models. In *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, 9-11 February, 2015.*, pp. 62–73. 2015.

[Car15b] Xabier De Carlos, Goiuria Sagardui, Aitor Murguzur, Salvador Trujillo, and Xabier Mendialdua. Runtime translation of model-level queries to persistence-level. In *Model-Driven Engineering and Software Development - Third International Conference, MODELSWARD 2015, Angers, France, February 9-11, 2015, Revised Selected Papers*, pp. 97–111. 2015.

[Car16a] Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. CRUD Model Operations from EOL to SQL. In *MODELSWARD 2016 - Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016.* 2016.

[Car16b] Xabier De Carlos, Goiuria Sagardui, and Salvador Trujillo. Two-Step Transformation of Model Traversal EOL Queries for Large CDO Repositories. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pp. 141–157. 2016.

[Cdo16] CDO. `http://eclipse.org/cdo/`, 2016. Online. Accessed January 27, 2016.

[Cla08] Manuel Clavel, Egea Marina, and Garcia de Dios Miguel Angel. Building an efficient component for ocl evaluation. *Electronic Communications of the EASST*, vol. 15, 2008.

[Cyp15] Cypher Query Language. `http://neo4j.com/docs/stable/cypher-query-lang.html`, 2015. Online. Accessed January 27, 2016.

[Deb16] Csaba Debreceni, István Ráth, Dániel Varró, Xabier De Carlos, Xabier Mendialdua, and Salvador Trujillo. Automated model merge by design space exploration. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pp. 104–121. 2016.

[Dem01] Birgit Demuth, Heinrich Hussmann, and Sten Loecher. OCL as a Specification Language for Business Rules in Database Applications. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, vol. 2185 of *Lecture Notes in Computer Science*, pp. 104–117. Springer Berlin Heidelberg, 2001.

[Dem04] Birgit Demuth. The dresden ocl toolkit and its role in information systems development. In *Proc. of the 13th International Conference on Information Systems Development (ISD 2004)*, vol. 7. 2004.

[Dem09] Birgit Demuth and Claas Wilke. Model and object verification by using dresden ocl. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia*, pp. 687–690. Citeseer, 2009.

[DR12] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems. In *Proceedings of the 6th International Conference on Graph Transformations*, ICGT'12, pp. 20–37. Springer-Verlag, Berlin, Heidelberg, 2012.

[Eff06] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, vol. 32, p. 118. 2006.

[Ege10] Marina Egea, Carolina Dania, and Manuel Clavel. MySQL4OCL: A Stored Procedure-Based MySQL Code Generator for OCL. *Electronic Communications of the EASST*, vol. 36, 2010.

[Ehr04] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. *Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28–October 1, 2004. Proceedings*, chap. Constraints and Application Conditions: From Graphs to High-Level Structures, pp. 287–303. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[Ehr05] Karsten Ehrig, J Küster, Gabriele Taentzer, and Jessica Winkelmann. *Automatically generating instances of meta models*. Citeseer, 2005.

[Emf12] EMF Query 2. `https://wiki.eclipse.org/EMF/Query2`, 2012. Online. Accessed January 27, 2016.

[Emf15] EMF Query. `https://projects.eclipse.org/projects/modeling.emf.query/`, 2015. Online. Accessed January 27, 2016.

[EP11] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In Jon Whittle, Tony Clark, and Thomas Kahne, editors, *Model Driven Engineering Languages and Systems*, vol. 6981 of *Lecture Notes in Computer Science*, pp. 77–92. Springer Berlin Heidelberg, 2011.

[EP13a] Javier Espinazo Pagán. Morsa. `http://modelum.es/trac/morsa/`, 2013. Online. Accessed January 27, 2016.

[EP13b] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. A Repository for Scalable Model Management. *Software & Systems Modeling*, pp. 1–21, 2013.

[EP14] Javier Espinazo Pagán and Jesús García Molina. Querying Large Models Efficiently. *Inf Softw Technol*, vol. 56(6):pp. 586–622, Jun 2014.

[Esp13] Espinazo Pagán, Javier. *A Model Repository for Scalable Access*. Ph.D. thesis, University de Murcia, 2013.

[Gar14] Antonio Garmendia, Esther Guerra, Dimitrios S Kolovos, and Juan de Lara. Emf splitter: A structured approach to emf modularity. *XM@ MoDELS*, vol. 1239:pp. 22–31, 2014.

[Góm15a] Abel Gómez, Amine Benelallam, and Massimo Tisi. Decentralized model persistence for distributed computing. *BigMDE 2015*, p. 42, 2015.

[Góm15b] Abel Gómez, Massimo Tisi, Gerson Sunyé, and Jordi Cabot. Map-based transparent persistence for very large models. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, vol. 9033 of *Lecture Notes in Computer Science*, pp. 19–34. Springer Berlin Heidelberg, 2015.

[Gra09] GraBaTs 2009, 5th International Workshop on Graph-Based Tool. `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`, 2009. Online. Accessed February 24, 2016.

[Gro16] Object Management Group. Object Constraint Language. `http://www.omg.org/spec/OCL/`, 2016. Online. Accessed January 27, 2016.

[Hei07] Florian Heidenreich, Christian Wende, and Birgit Demuth. A Framework For Generating Query Language Code From OCL Invariants. *Electronic Communications of the EASST*, vol. 9, 2007.

[Hil15] Guillaume Hillairet. EMFJSON. `http://emfjson.org/docs/`, 2015. Online. Accessed March 09, 2016.

[Hun14] Bryan Hunt. Mongo EMF. `https://github.com/BryanHunt/mongo-emf/wiki`, 2014. Online. Accessed January 27, 2016.

[Kal16] Sahar Kallel, Chouki Tibermacine, Bastien Tramoni, Christophe Dony, and Ahmed Hadj Kacem. Automatic translation of ocl meta-level constraints into java meta-programs. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015*, pp. 213–226. Springer, 2016.

[Kär09] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*. 2009.

[Kol06] Dimitrios S. Kolovos, Richard F. Paige Paige, and Fiona A.C. Polack. The epsilon object language (EOL). In *Model Driven Architecture - Foundations and Applications, Second European*

Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings, pp. 128–142. 2006.

[Kol08] Dimitrios S. Kolovos, Louis Rose, Antonio Garcia-Dominguez, and Richard F. Paige. The Epsilon Book, 2008.

[Kol13] Dimitrios S. Kolovos, Ran Wei, and Konstantinos Barmpis. An approach for efficient querying of large relational datasets with OCL based languages. In *Proceedings of the Workshop on Extreme Modeling co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2013), Miami, Florida, USA, September 29, 2013.*, pp. 46–54. 2013.

[Map16] MapDB: Database Engine. `http://www.mapdb.org/`, 2016. Online. Accessed March 08, 2016.

[Mar99] Ulrich Marder, Norbert Ritter, and Hans-Peter Steiert. A DBMS-Based Approach for Automatic Checking of OCL Constraints. In *Proceedings of OOPSLA*, vol. 99, pp. 1–5. 1999.

[Mon15] MongoDB CRUD Concepts. `https://docs.mongodb.org/manual/core/crud/`, 2015. Online. Accessed January 27, 2016.

[Mon16] MongoDB. `https://www.mongodb.org`, 2016. Online. Accessed January 27, 2016.

[Mur14a] Aitor Murguzur, Xabier De Carlos, Salvador Trujillo, and Goiuria Sagardui. Context-Aware Staged Configuration of Process Variants@Runtime. In *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*, pp. 241–255. 2014.

[Mur14b] Aitor Murguzur, Xabier De Carlos, Salvador Trujillo, and Goiuria Sagardui. On the Support of Multi-perspective Process Models Variability for Smart Environments. In *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pp. 549–554. 2014.

[Neo16] Neo4j. `http://www.neo4j.org/`, 2016. Online. Accessed January 27, 2016.

[Ori15] Xavier Oriol and Ernest Teniente. Incremental checking of ocl constraints with aggregates through sql. In *Conceptual Modeling*, pp. 199–213. Springer, 2015.

[RHM04] LLC Red Hat Middleware. Hql: The hibernate query language. `https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html`, 2004. [Online; accessed 13-November-2015].

[Sch12] Markus Scheidgen, Anatolij Zubow, Joachim Fischer, and Thomas H. Kolbe. Automated and transparent model fragmentation for persisting large models. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pp. 102–118. Springer-Verlag, Berlin, Heidelberg, 2012.

[Sch13] Markus Scheidgen. Reference representation techniques for large models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pp. 5:1–5:9. ACM, New York, NY, USA, 2013.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Softw*, vol. 20(5):pp. 19–25, Sep 2003.

[Sha14] Seyyed M. Shah, Ran Wei, Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Konstantinos Barmpis. A Framework to Benchmark NoSQL Data Stores for Large-Scale Model Persistence. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, vol. 8767 of *Lecture Notes in Computer Science*, pp. 586–601. Springer International Publishing, 2014.

[Sot09] Jean-Sébastien Sottet, Frédéric Jouault, et al. Program comprehension. In *Proc. 5th Int. Workshop on Graph-Based Tools*. 2009.

[Sql16] SQL. `http://www.w3schools.com/sql/`, 2016. [Online; accessed 13-November-2015].

[Szá15] Gábor Szárnyas, Oszkár Semeráth, István Ráth, and Dániel Varró. The ttc 2015 train benchmark case for incremental model validation. *8th Transformation Tool Contest (TTC 2015)*, 2015.

[Ten12] Teneo Hibernate. `http://wiki.eclipse.org/Teneo/Hibernate`, 2012. Online. Accessed January 27, 2016.

[Ujh15] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: an Integrated Development Environment for Live Model Queries. *Sci Comput Program*, vol. 98:pp. 80–99, 2015.

[Völ09] Markus Völter. Best practices for dsls and model-driven development. *Journal of Object Technology*, vol. 8(6):pp. 79–102, 2009.

[Wei15] Ran Wei and Dimitrios S. Kolovos. An efficient computation strategy for allinstances(). In *Proceedings of the 3rd Workshop on Scalable Model Driven Engineering part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 23, 2015.*, pp. 32–41. 2015.

[Wil12] Edward Willink. OCLinEcore. `https://wiki.eclipse.org/OCL/OCLinEcore`, 2012. Online. Accessed January 27, 2016.

[Win08] Jessica Winkelmann, Gabriele Taentzer, Karsten Ehrig, and Jochen M Küster. Translation of restricted ocl constraints into graph constraints for generating meta model instances by graph grammars. *Electronic Notes in Theoretical Computer Science*, vol. 211:pp. 159–170, 2008.

A
Appendix: QLI Metamodel

A.1 QLI Metamodel Overview

Figure A.1 included in this appendix depicts the overview of the QLI Metamodel. It illustrates all the specified artifacts and relationships within the metamodel.

A.2 Mapping between Model Query Languages and QLI Models

This section includes sample QLI Models that correspond with the queries that are expressed using different model query languages. Queries are simplifications of queries that are used in the Train Benchmark Case (see 8).

A.2.1 QLI Model for an IncQuery query.

Listing A.1 shows a query that has been expressed using IncQuery model query language. The query obtains all the segment instances from the model having length attribute value lower or equal than zero. As the first line of the query indicates, the query returns artifacts collected within the `segment` variable. This variable specifies `Segment` instances within the model, and additionally `length`

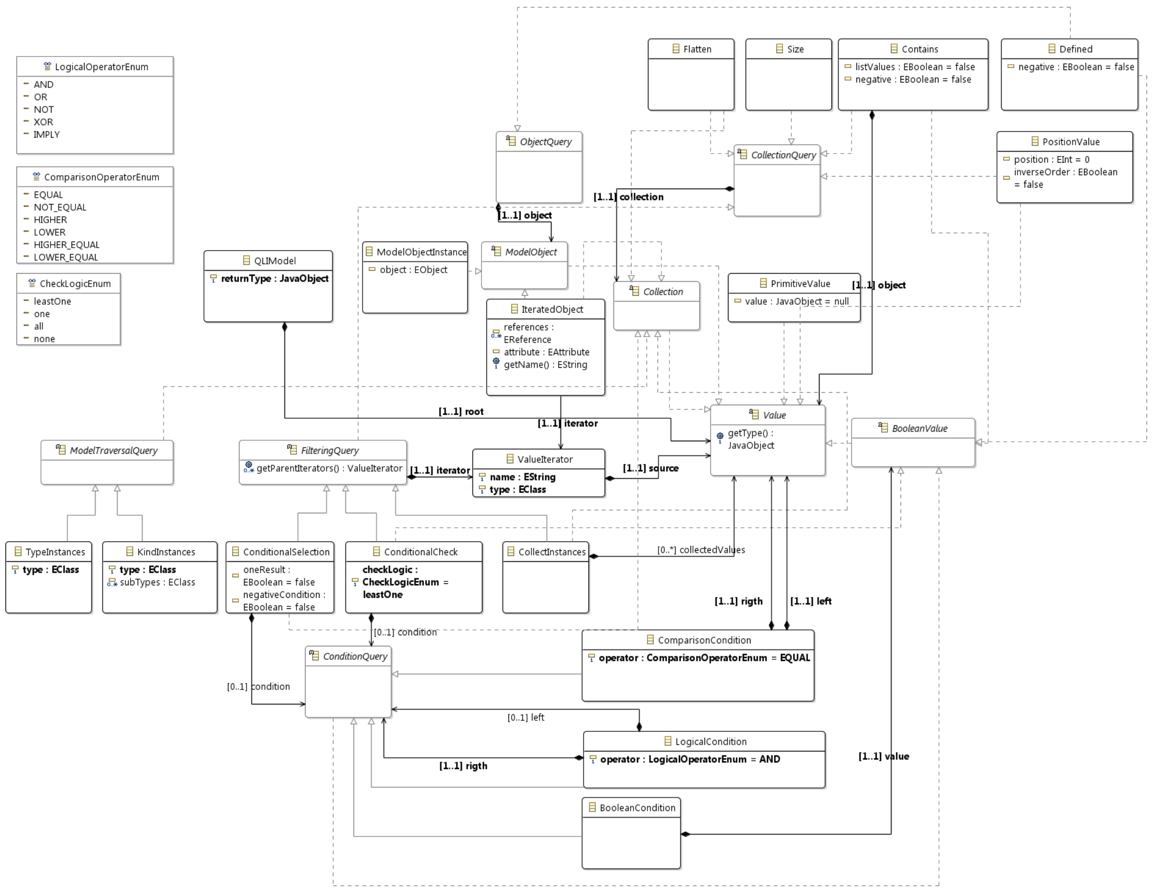


Figure A.1: QLI Metamodel.

variable specifies length attribute value of each segment (line 3). Line 4 of the query adds a condition that checks if the value from the length attribute is lower or equal than zero. This statement causes segment variable to contain only instances satisfying the condition.

```
1 pattern PosLength(segment)
2 {
3   Segment.length(segment, length);
4   check(length <= 0);
5 }
```

Listing A.1: Sample IncQuery query.

Figure A.2 depicts the QLI Model that corresponds with the IncQuery query. As previously described, the query returns `Segment` instances. Consequently, it is specified at the `returnType` attribute of the `QLIModel` instance. Additionally, all the `Segment` instances existing within the models have to be traversed, and this is specified by the `KindInstances` instance. This instance is contained by a `ValueIterator` instance that is part of the `ConditionalSelection` that selects only values from the iterator that satisfy a condition. The condition expression is defined at line 4 of the query (see Listing A.1). This condition is specified in the model by a `ComparisonCondition` instance that compares value of the length attribute (specified by the `IteratedObject` object) and zero value (specified by the `PrimitiveValue` object).

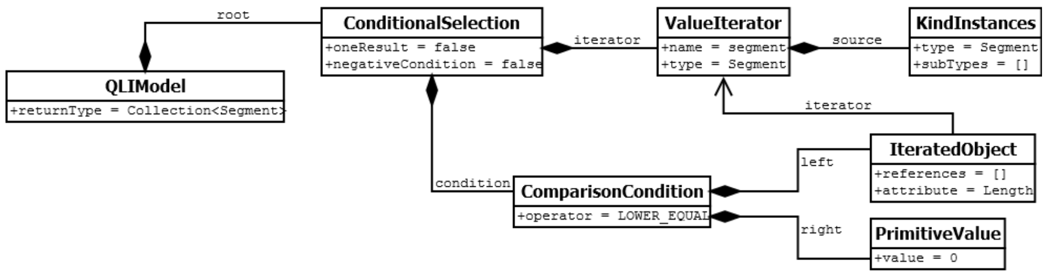


Figure A.2: QLI Model equivalent to the IncQuery query.

A.2.2 QLI Model for an EMF API query.

Listing A.2 shows a query that has been expressed using EMF API. The query iterates all the Switch instances from the model (lines 2-8) and selects only the instances that have not any object referenced at the sensor reference (lines 9-14).

```
 1 public Collection<Switch> switchSensor(Resource model) throws
       Exception {
 2     Collection<Switch> result = new ArrayList<Switch>();
 3     TreeIterator<EObject> contents = model.getAllContents();
 4     while (contents.hasNext()) {
 5       EObject obj = contents.next();
 6       if (!RailwayPackage.eINSTANCE.getSwitch().isInstance(
            obj)) {
 7         continue;
 8       }
 9       final Switch sw = (Switch) obj;
10       if (sw.getSensor() == null) {
11         result.add(sw);
12       }
13     }
14     return matches;
15   }
```

Listing A.2: Sample EMF API query.

Figure A.3 illustrates the QLI Model that corresponds with the previous query. The query returns a collection of Switches, and it is specified at the `returnType` attribute of the root `QLIModel` instance. This instance contains a `ConditionalSelection` instance, abstraction for query expressions that return only values from the input collection that satisfy a condition. Iterated values are specified by the `ValueIterator` instance that contains `KindInstances` instance, where all the `Switch` instances existing within the model are collected (lines 3-6 of Listing A.2). `ConditionalSelection` includes a condition that is specified by the `BooleanCondition` instance. This instance contains a `Defined` instance, an abstraction for the condition specified at lines 9 and 10 of the query shown in Listing A.2.

A.2.3 QLI Model for an OCL query.

Listing A.3 illustrates a query that has been specified using OCL. This is a simplified version of the *SwitchSet* query of the Train Benchmark Case and it searches for `Route` instances (line 1-2) that have a `Semaphore` instance with `GO` signal value (lines 3-4) and additionally the `Route` instance follows a `SwitchPosition` that contains a `Switch` with `position` value different of

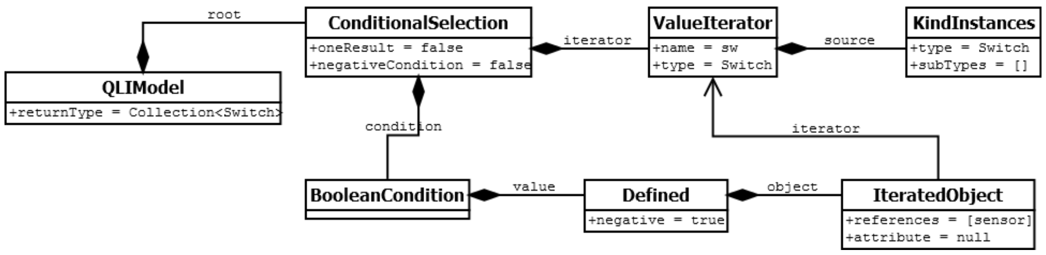


Figure A.3: QLI Model equivalent to the EMF API query.

the `SwitchPosition.currentPosition` value (line 5-6). The query returns a collection containing `Switch` instances where the previous conditions are satisfied (line 5).

```
1 Route.allInstances()
2   ->collect( route |
3     route.entry->select(signal = Signal::GO)
4     ->collect(semaphore |
5       route.follows->collect(swP |
6       swP.switch->select(currentPosition <> swP.position)
7     )
8   )
9 )
```

Listing A.3: Query expressed with OCL.

Figure A.4 illustrates the QLI Model that specifies the OCL query. As figure shows, first line of the query shown in Listing A.3 is specified by a `KindInstances` instance. This instance is contained by a `CollectInstances` instance with a `ValueIterator` for iterating values of the `KindInstances`. These artifacts specify OCL expressions of line 2. Then, entry reference is navigated for each iterated Route, selecting only those values that have the signal attribute with the GO value (line 3). This EOL fragment is specified by a `ConditionalSelection` instance which contains a `ComparisonCondition` that is the abstraction for the condition to be checked. The OCL query collects values returned by the select, and navigates `follows` feature values (lines 4-5). This is specified by two `CollectInstances` instances in the QLI Model. For finishing, the query navigates the switch feature values of the `SwitchPosition` instances and selects only values that satisfy the condition. It is specified by a `ConditionalSelection` instance that contains a `ComparisonCondition` (abstraction used to specify the condition of the select).

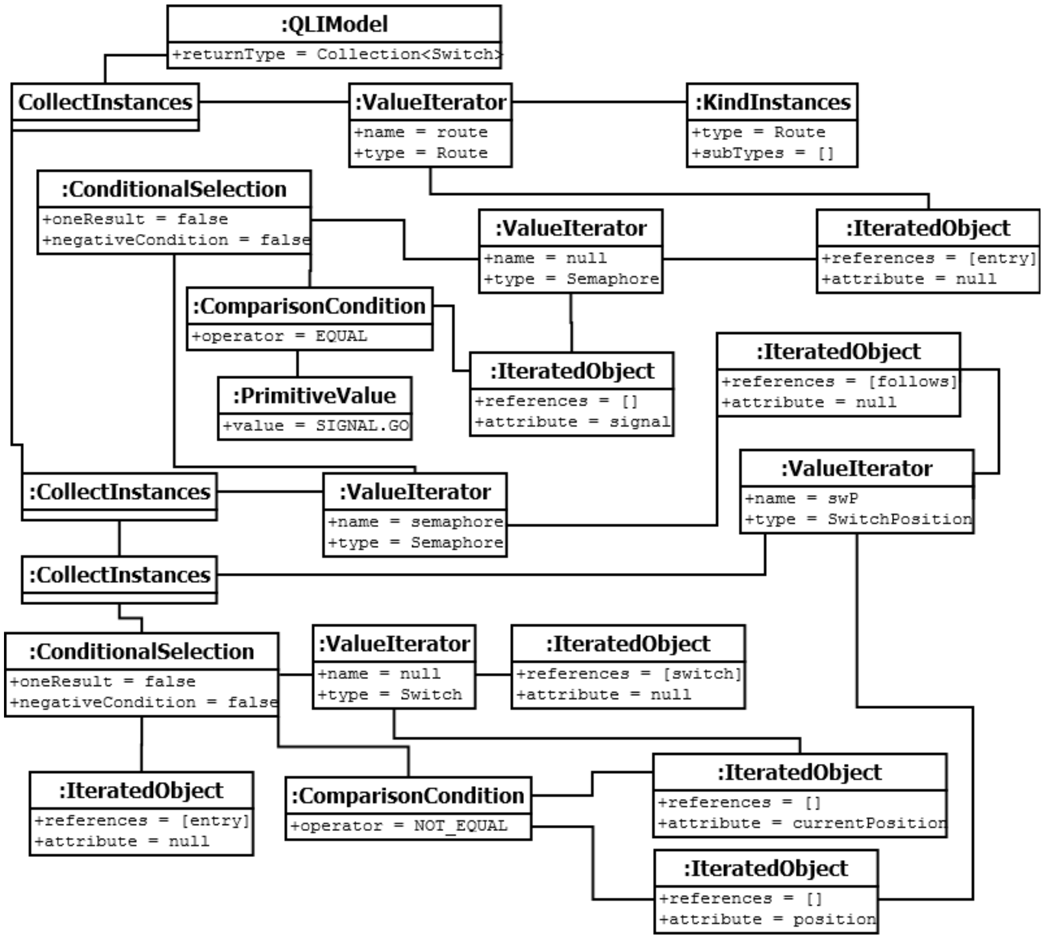


Figure A.4: QLI Model equivalent to the SwitchSet query expressed with OCL.

A.3 Mapping between QLI Model and Persistence-Specific Query Languages

This section illustrates queries that are expressed with different persistence-specific query languages. These queries are the output that would be provided by MQT-Engine from a QLI Model instance. The selected QLI Model instance is illustrated in Figure A.5 and specifies a query that returns all the `Segment` instances existing within the queried model that have `length` attribute value lower or equal than zero.

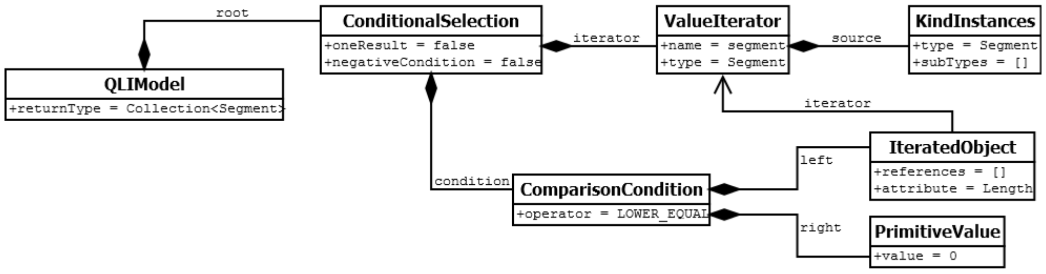


Figure A.5: Input QLI Model instance.

.

As figure shows, the `QLIModel` instance is the root of the model and it contains an attribute that specifies the type returned by the query: a collection of `Segment` instances. `ConditionalSelection` specifies the query that iterates an input collection values (`ValueIterator` instance that inputs all the `Segment` instances, specified by `KindInstances`) and selects only values that satisfy a condition. In the case of this query the condition is specified by a `ComparisonCondition` instance, and it evaluates that the value of the iterated object (`IteratedObject` instance) is lower or equal than zero (specified by a `PrimitiveValue` instance).

A.3.1 Generated Cypher Query for NeoEMF/Graph

NeoEMF/Graph provides alternative model persistence based on the Neo4J noSQL graph database. Neo4J database engine supports querying graph databases with a persistence specific graph query language known as Cypher. Cypher queries are expressed with patterns that search the specified nodes and relationships. Listing A.4 shows the cypher query that would be generated by MQT-Engine for the QLI Model of the Figure A.5, and it is described below:

The first line of the input query indicates that all the nodes within the database are evaluated. The query returns all the Segment instances that satisfy a condition, and it is specified by `ConditionalSelection` in the QLI Model. This instance iterates all the elements of the model that are instance of the Segment type. It is specified by the `KindInstances` abstraction which generates line 2 of the cypher query and adds the first condition of line 3 (`cls.name='Segment'`). The rest of the line 3 corresponds with the condition

```
1 START segment=node(*)
2 MATCH   (segment) -[:kyanosInstanceOf]-> (cls)
3 WHERE cls.name='Segment' AND has(segment.length) AND segment.
    length <= '0'
4 RETURN segment
```

Listing A.4: Cypher query for models persisted with NeoEMF/Graph.

specified by the `ComparisonCondition` instance. As the `returnType` attribute of the `QLIModel` instance indicates, the query returns a collection of segments and it is specified in the last line of the cypher query (line 4).

A.3.2 Generated SQL Query for an Ad-hoc persistence

Figure A.6 illustrates the data-schema of an ad-hoc persistence for models [Car15a]. The persistence uses H2 database back-end for persistence of models, and all the information of models is persisted in the five tables of the figure. The schema is metamodel-agnostic and persistence of models in the database is independent of metamodels they conform to. Thus, any model can be persisted under the same schema, so in case metamodel evolves, no changes are required in the schema. Tables and relations shown on the schema are described below:

- **Object** table: A tuple in this table is created for each element of the model. Model elements are identified by a primary key in the row of the ObjectID column and the meta-class ID (foreign key) of each element is stored in the row of the ClassID column.

- **Class** table: Contains all meta-classes of the model. ClassID (primary key) and Name of the meta-class are stored for each one.

- **Feature** table: It stores an ID (FeatureID column, primary key) and the name (Name column) for each attribute and reference in the metaclasses of the model.

- **AttributeValue** table: This table stores attribute values of model elements. Attribute values are identified by an ObjectID and a FeatureID (both foreign keys), and the Value column stores the primitive value of the attribute. In the case of single-file attributes with empty value, default value is stored.

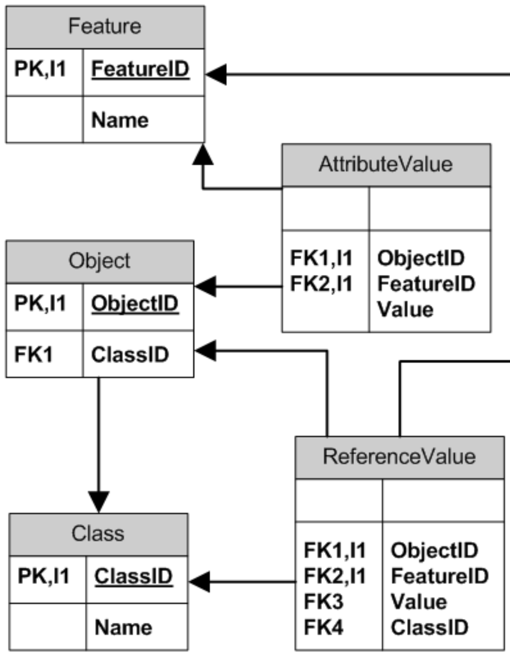


Figure A.6: Data-schema defined in the ad-hoc persistence.

- **ReferenceValue** table: This table stores references of model elements. References are identified by an ObjectID and a FeatureID (both foreign keys). The ID of the referenced element (Value column, foreign key) and meta-class of the referenced element (ClassID column, foreign key) identify the referenced model element.

Listing A.5 shows the SQL query that would be generated by MQT-Engine from the QLI Model instance and for the ad-hoc persistence with the previously described data-schema:

The query returns a collection of `Segment` instance and consequently, the SQL query returns the identifier of each segment that satisfies the condition (line 1). The `KindInstances` abstraction of the model indicates that the ValueIterator iterates all the Segment instances within the model. The output for these abstractions are lines 2 and 3 of the SQL query where identifiers of all the Segment instances within the database are obtained. However, as the `ConditionalSelection` instance of the QLI Model indicates, only values that satisfy the condition are returned by the query (line 4). This line is generated from the `ComparisonCondition` instance, and it checks if the length value obtained

```
SELECT DISTINCT Segment.ObjectID
FROM Object AS Segment
INNER JOIN Class ON Object.ClassID = Class.ClassID AND Class.
    ClassID='Segment'
WHERE (SELECT Value FROM AttributeValue INNER JOIN Feature ON
     AttributeValue.FeatureID = Feature.FeatureID AND Feature.
    Name = 'length' WHERE AttributeValue.ObjectID=Segment.
    ObjectID LIMIT 1) <= (0)
```

Listing A.5: SQL query for models persisted with an ad-hoc persistence.

from the `AttributeValue` table is lower or equal than zero.

B
Experimental Evaluation

This appendix includes extended information about the experimental evaluations presented in Chapters 7 and 8 of this dissertation.

B.1 Queries and Manipulations

This section illustrates and describes the different queries and manipulations that are executed over models in the Train Benchmark Case-based experimental evaluation of Chapter 8.

Q1: PosLength.

Query. Listing B.1 illustrates *Q1* using EOL. As code shows, first all the *Segment* instances are obtained (line 1), selecting then only those that have the *length* value equals or lower than zero (line 2). The query returns a list containing collections, and each collection contains a *Segment* instance (line 3).

Manipulation. The candidates to be modified at the manipulation phase are a randomly selected *Segment* instances. They are returned at the previous execution of the *PosLength* query. The value of the *length* feature for the selected *Segment* instances is increased by one.

```
Segment.all
  .select(segment | segment.length <= 0)
    .collect(segment | Collection{segment});
```

Listing B.1: PosLength query expressed with EOL.

Q2: RouteSensor.

Query. Listing B.2 illustrates *Q2* query using EOL. As code shows, first all the *Route* instances are obtained (line 1). Next, the query collects the *SwitchPosition* instance referenced by routes at the *follows* feature (line 3). Additionally, *Switch* instances referenced at the *switch* feature are also collected (line 4). Then, the query selects the *Sensor* instances referenced at the `sensor` feature of *Switch* instances that are not referenced from the *Route* instances (line 5). Finally, the query returns a collection containing collections with *Route*, *Sensor*, *SwitchPosition* and *Switch* instances that satisfy the query conditions specified at the previous code lines (line 6).

```
1 Route.all
2   .collect( route |
3     route.follows.collect( swP |
4       swP.‘switch‘.collect( sw |
5         sw.sensor.select( sensor:Sensor | route.definedBy.excludes(sensor
            ))
6         .collect( sensor | Collection{route, sensor, swP, sw})
7     )
8   )
9 );
```

Listing B.2: RouteSensor query expressed with EOL.

Manipulation. The candidates to be modified are the *Route* instances that are returned at the previous execution of the *RouteSensor* query. The modification adds a new referenced *Sensor* in the *definedBy* feature of the route. The *Sensor* instance is also obtained from the previous query execution.

Q3: SemaphoreNeighbor.

Query. Listing B.3 illustrates *SemaphoreNeighbor* in EOL. First, it select all the route instances (line 1). Then, for each route, the query collects the `Semaphore` instance referenced at the *entry* feature (line 3), the *Sensors* that define the route (line 4), *TrackElement* instances connected to Sensors (line 5), and *TrackElement* instances connected to the previously collected *TrackElement*s (line 6). Next, the query obtains the sensor of the *TrackElement*s collected at line 6. Having all these artifacts, the query traverses again all the Route instances and selects only the routes that satisfy the conditions of the query that have been described previously (line 8). The query returns a collection composed by collections that contain *Semaphore*, *Route*, *Sensor* and *TrackElement* instances that satisfy the conditions.

```
1 Route.all
2   .collect( route1 |
3     route1.exit.collect(semaphore |
4       route1.definedBy.collect( sensor1 |
5         sensor1.elements.collect( te1 |
6           te1.connectsTo.collect( te2 |
7             te2.sensor.collect( sensor2 |
8               Route.all.select(route2 | route2.definedBy.includes(sensor2
                    ) and route2.entry<>semaphore and route1<>route2)
9             .collect(route2 | Collection{semaphore, route1, route2,
                  sensor1, sensor2, te1, te2})))))));
```

Listing B.3: SemaphoreNeighbor query expressed with EOL.

Manipulation. The candidates to be modified are a randomly selected set of *Route* instances that are returned at the previous execution of the *SemaphoreNeighbor* query. The modification changes the entry reference value (*Semaphore* instance).

Q4: SwitchSensor.

Query. Listing B.4 illustrates *Q4* in EOL. As code shows, first all the *Switch* instances are obtained (line 1), selecting then only those that do not reference any *Sensor* (line 2). The query returns a list containing collections, and each collection contains a *Switch* instance (line 3).

```
1 Switch.all
2   .select(sw | sw.sensor.isUndefined())
3   .collect(sw | Collection{sw});
```

Listing B.4: SemaphoreNeighbor query expressed with EOL.

Manipulation. The modified artifacts are a subset of the *Switch* instances returned by the previous execution of the *SwitchSensor* query. For each candidate, the modification creates a new *Sensor* instance and references it from the *sensor* feature of the *Switch*.

Q5: SwitchSet.

Query. Listing B.5 illustrates the query in EOL. As code shows, it first collects all the *Route* instances, and then searches *Semaphore* instances that are referenced by routes and have *GO* signal value (lines 3 and 4). Next, the query checks if the *Route* instances contain Switch instances with a position value that is different of the *SwitchPosition* that refers to the *Switch* (line 5). The query returns a collection containing collections that contain *Route*, *Semaphore*, *SwitchPosition* and *Switch* instance that satisfy the conditions.

```
1 Route.all
2   .collect(route | route.entry.select(semaphore |
3     semaphore.signal = Signal#GO)
4       .collect(semaphore | route.follows
5     .collect(swP | swP.‘switch‘.select(sw | sw.currentPosition<>swP.
          position)
6     .collect(sw | Collection{route, semaphore, swP, sw}))));
```

Listing B.5: SemaphoreNeighbor query expressed with EOL.

Manipulation. The modified artifacts are a subset of the *Switch* instances returned by the previous execution of the *SwitchSet* query. The modification sets the *currentPosition* value of the *Switch*, with the value that has the *SwitchPosition* instance also returned by the previous query execution. This way, positions are equal for the *Switch* and the *SwitchPosition* and the constraint is not violated.

B.2 Results

This section presents the results that have been obtained in the experimental evaluations of Chapters 7 and 8. Tables B.1 and B.2 correspond with the results obtained in the experiments that are part of the *Reverse Engineering Case* evaluation. Tables B.3, B.4, B.5, B.6, B.7 correspond with the results obtained in the experiments that are part of the *Train Benchmark Case* evaluation.

Table B.1: Performance and memory usage results for F1 in the Reverse Engineering Case experiments.

F1: Size of the model											
		Plain EMF		MDT OCL		CDO OCL		SQL		MQT-Engine	
		time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)
Q1	Set0	17	396	17	335	1	123	0	115	1	130
	Set1	48	869	45	708	1	67	1	119	1	77
	Set2	464	3599	443	3198	5	136	4	146	4	139
	Set3	1059	5749	1018	5711	8	246	7	286	8	254
	Set4	1166	6016	1090	6076	8	235	8	285	8	263
Q2	Set0	18	513	18	342	1	123	0	118	1	78
	Set1	48	1026	45	686	2	67	1	65	1	79
	Set2	463	3456	454	3065	10	238	4	152	5	223
	Set3	1050	5701	1016	5672	20	558	10	429	12	454
	Set4	1155	6095	1090	6081	22	636	11	375	12	315
Q3	Set0	18	400	18	322	1	67	0	65	1	130
	Set1	48	1028	46	934	2	67	1	126	1	130
	Set2	473	3407	453	3112	11	337	2	154	3	244
	Set3	1069	5672	1023	5731	26	525	5	289	7	295
	Set4	1140	6133	1101	6110	28	590	6	289	7	264

Table B.2: Performance and memory usage results for F2 in the Reverse Engineering Case experiments.

F2: Size of the repository											
		Plain EMF		MDT OCL		CDO OCL		SQL		MQT-Engine	
		time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)
Q1	Set4	1166	6016	1090	6076	8	235	8	285	8	263
	Set5	1167	6016	1092	6013	28	381	12	418	14	434
	Set6	1166	6052	1104	6072	49	442	17	575	19	463
	Set7	1174	5988	1100	6027	77	533	23	608	24	674
	Set8	1170	6071	1102	6082	101	680	28	833	28	718
	Set9	1169	6073	1081	6044	131	695	32	849	34	761
Q2	Set4	1155	6095	1090	6081	22	636	11	375	12	315
	Set5	1142	6079	1098	6076	51	991	18	545	20	500
	Set6	1146	6097	1109	6017	90	1192	26	811	28	685
	Set7	1158	6006	1094	6055	128	1395	34	1020	34	1037
	Set8	1153	6046	1095	6033	169	1675	41	1273	42	948
	Set9	1154	6161	1104	6049	200	1860	48	1249	48	1136
Q3	Set4	1140	6133	1101	6110	28	590	6	289	7	264
	Set5	1144	6005	1107	6074	67	1099	10	524	12	396
	Set6	1155	5986	1113	6038	102	1224	15	642	17	526
	Set7	1170	6043	1093	6092	144	1516	19	978	19	625
	Set8	1155	6025	1095	6043	179	1771	23	969	23	620
	Set9	1143	6046	1089	6064	217	2171	27	968	27	579

Table B.3: Execution time (seconds) for read phase (M1) in the Batch scenario.

Batch M1: read phase execution time average in seconds (s)												
		XMI				MySQL	Neo4J		CDO			
Query	Model	EMF	OCL	IncQuery Local	IncQuery Incremental	SQL	Core API	Cypher	EMF	OCL Client-Side	OCL Server-Side	MQT-Engine EOL
PosLength	batch-1	0.2	0.1	0.2	0.2	0.2	0.8	1.3	1.2	0.8	0.7	0.7
	batch-2	0.1	0.2	0.2	0.1	0.1	0.9	0.8	0.9	0.7	0.6	0.6
	batch-4	0.2	0.2	0.3	0.3	0.1	1.1	1.1	0.9	0.7	0.8	0.6
	batch-8	0.3	0.7	0.5	0.5	0.1	1.5	1.4	0.7	0.7	0.7	0.7
	batch-16	0.4	0.7	0.8	0.8	0.2	2.2	2.2	0.7	0.7	0.7	0.7
	batch-32	0.5	0.8	1.2	1.2	0.3	3.5	3.4	0.8	0.7	0.7	0.7
	batch-64	0.8	1	1.6	1.7	0.5	5.7	5.2	1	1	1	1
	batch-128	1.3	1.3	2.5	2.6	0.8	11	10.6	1.2	1.1	1.1	1.1
	batch-256	2.3	2.4	4.7	4.8	1.7	22.4	23.6	1	0.9	1	0.8
	batch-512	4.6	5.1	8.9	8.4	3.6	50.1	51.4	1.8	1.7	1.7	1.6
	batch-1024	10	9.5	17.6	17.8	7.9			2.9	2.8	3.6	2.5
	batch-2048	23.3	22.6	36.8	37.6	17.4			3	3.7	4.1	3.2
	batch-4096	50.2	44.7	77.6	82.7	38.4			3.8	3.9	3.5	3.8
RouteSensor	batch-1	0.1	0.1	0.1	0.1	0.1	0.8	0.9	0.9	0.8	0.7	0.8
	batch-2	0.1	0.1	0.2	0.2	0.1	0.9	0.9	0.8	0.7	0.6	0.8
	batch-4	0.2	0.2	0.3	0.4	0.1	1.1	1.1	0.7	0.8	0.7	0.9
	batch-8	0.3	0.3	0.6	0.6	0.1	1.5	1.5	0.8	0.7	0.7	0.9
	batch-16	0.4	0.3	0.9	0.8	0.2	2.4	2.1	0.7	0.7	0.7	1
	batch-32	0.6	0.6	1.2	1.2	0.3	3.5	3.3	0.8	0.8	0.7	1
	batch-64	0.9	0.8	1.6	1.8	0.5	6	5.3	1.1	1.1	1	1.1
	batch-128	1.5	1.2	2.9	3.2	0.8	10.6	10.3	1.2	1.3	1.1	1.3
	batch-256	2.4	2.2	5.9	6	1.8	22.9	22.6	1	1.1	0.8	1
	batch-512	4.2	3.9	10.6	11.5	3.6	52.6	51.3	1.8	2.4	1.7	1.6
	batch-1024	9.8	9.4	23.8	25.6	7.9			3	2.8	2.6	2.4
	batch-2048	23.4	20	54.1	58.6	17.5			3.2	2.8	3	3.1
	batch-4096	45.2	42.7	119	122.5	37.8			8.5	3.8	4	4
SemaphoreNeighbor	batch-1	0.1	0.1	0.2	0.1	0.1	0.8	0.9	1	0.9	0.7	0.6
	batch-2	0.1	0.1	0.2	0.2	0.1	0.9	1	0.7	0.8	0.6	0.6
	batch-4	0.2	0.2	0.3	0.4	0.1	1.1	1.1	0.7	0.7	0.7	0.6
	batch-8	0.3	0.2	0.5	0.5	0.1	1.5	1.5	0.7	0.7	0.7	0.6
	batch-16	0.5	0.3	0.9	0.9	0.2	2.4	2.2	0.7	0.7	0.7	0.6
	batch-32	0.6	0.5	1.3	1.3	0.3	3.7	3.6	0.8	0.8	0.7	0.6
	batch-64	0.9	0.8	2.5	2.3	0.5	5.1	5.5	1	1	1	0.8
	batch-128	1.3	1.3	4.7	5.1		11.6	9.8	1.2	1.2	1.2	1
	batch-256	2.6	2.3	10	10.1		24	22.6	0.9			0.8
	batch-512	4.5		21.7	23.5		47.7	49.4	1.7			1.5
	batch-1024	10.1		47.1	45				2.7			2.5
	batch-2048	21.6		122.4	113.1				2.6			3.2
	batch-4096	45.3							3.7			4.1
SwitchSensor	batch-1	0.1	0.1	0.1	0.2	0.1	0.8	0.8	0.8	0.8	0.7	0.6
	batch-2	0.1	0.1	0.2	0.3	0.2	0.9	0.9	0.7	0.8	0.6	0.6
	batch-4	0.2	0.2	0.3	0.4	0.1	1.1	1	0.7	0.8	0.6	0.6
	batch-8	0.2	0.3	0.5	0.6	0.2	1.6	1.6	0.7	0.8	0.7	0.6
	batch-16	0.4	0.4	0.9	0.9	0.2	2.2	2.2	0.7	0.9	0.7	0.6
	batch-32	0.6	0.6	1.1	1.4	0.3	3.4	3.3	0.7	0.8	0.7	0.9
	batch-64	0.9	0.9	1.7	2.3	0.5	6.2	5.5	1	1.2	1	1.2
	batch-128	1.4	1.3	2.8	3.2	1	10.6	9.7	1.2	1.3	1.2	1.2
	batch-256	2.3	2.6	5	5.9	2.1	23.3	22.8	0.9	1.4	1.4	0.8
	batch-512	4.3	4.3	8.5	10.1	4.1	52.8	48.7	1.8	1.8	1.9	1.7
	batch-1024	10	9.4	20.2	22.5	9			2.6	2.7	2.8	2.6
	batch-2048	23.2	22	45.1	52.1	20.2			2.6	2.9	3.6	3.4
	batch-4096	47.6	45.8	91.5	111.9	41.9			5.5	4.7	4.1	4
SwitchSet	batch-1	0.1	0.1	0.1	0.1	0.1	0.9	0.8	0.8	0.9	0.7	0.6
	batch-2	0.1	0.1	0.2	0.1	0.1	0.9	0.9	0.7	0.7	0.7	0.6
	batch-4	0.2	0.2	0.3	0.3	0.1	1.1	1.1	0.7	0.7	0.7	0.6
	batch-8	0.3	0.3	0.4	0.4	0.2	1.5	1.5	0.7	0.7	0.7	0.6
	batch-16	0.4	0.4	0.6	0.7	0.2	2.3	2.3	0.7	0.7	0.8	0.6
	batch-32	0.6	0.6	0.9	0.9	0.3	3.7	3.6	0.7	0.8	0.7	0.6
	batch-64	1.2	0.9	1.5	1.3	0.5	6	5.1	1	1	1.1	0.9
	batch-128	1.3	1.4	2.7	2.4	1	11.2	10.3	1.2	1.2	1.3	1
	batch-256	2.5	2.6	4.4	4.6	2	23.6	24.3	0.9	0.9	1	0.8
	batch-512	4.4	4.8	7.6	7.9	4.3	54.1	47.5	1.7	1.7	2	1.6
	batch-1024	10	9.6	16.7	15.3	9			2.5	2.7	2.9	2.6
	batch-2048	23.5	22.1	34.8	34.9	19.8			3	2.8	3.6	3.3
	batch-4096	49.2	45.8	78.3	71.9	43.3			3.7	4.6	3.6	4

Table B.4: Execution time (seconds) for check phase (M2) in the Batch scenario.

Batch M2: check phase execution time average in seconds (s)												
		XMI				MySQL	Neo4J		CDO			
Query	Model	EMF	OCL	IncQuery Local	IncQuery Incremental	SQL	Core API	Cypher	EMF	OCL Client-Side	OCL Server-Side	MQT-Engine EOL
PosLength	batch-1	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.3	0.7	0.3	0.1
	batch-2	0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.3	1.0	0.4	0.1
	batch-4	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.8	1.9	0.6	0.1
	batch-8	0.0	0.1	0.0	0.0	0.0	0.0	0.3	2.7	3.0	1.0	0.1
	batch-16	0.0	0.2	0.0	0.0	0.0	0.1	0.4	4.9	5.0	1.7	0.1
	batch-32	0.0	0.2	0.0	0.0	0.0	0.1	0.6	8.8	9.3	2.6	0.2
	batch-64	0.1	0.2	0.0	0.0	0.0	0.1	0.8	16.2	16.7	3.6	0.2
	batch-128	0.1	0.3	0.0	0.0	0.0	0.2	0.5	27.3	32.5	7.3	0.3
	batch-256	0.3	0.7	0.0	0.0	0.0	1.7	0.7	56.9	62.2	15.3	0.5
	batch-512	0.4	1.3	0.0	0.0	0.0	0.6	3.0	108.2	125.3	27.6	0.7
	batch-1024	0.8	2.2	0.0	0.0	0.0			226.9	248.1	45.3	1.0
	batch-2048	1.2	5.2	0.0	0.0	0.1			488.6	501.2	97.9	1.9
	batch-4096	3.0	11.0	0.0	0.0	0.1			1870.3	2193.3	221.6	3.7
RouteSensor	batch-1	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.7	0.8	0.2	0.2
	batch-2	0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.0	1.1	0.3	0.2
	batch-4	0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.8	2.2	0.4	0.3
	batch-8	0.0	0.0	0.0	0.0	0.0	0.0	0.4	3.0	3.4	0.5	0.3
	batch-16	0.0	0.1	0.0	0.0	0.0	0.1	0.5	5.2	5.9	0.7	0.4
	batch-32	0.0	0.1	0.0	0.0	0.0	0.1	0.7	9.7	10.5	1.1	0.4
	batch-64	0.1	0.2	0.0	0.0	0.0	0.1	1.0	17.7	18.7	1.5	0.4
	batch-128	0.1	0.2	0.0	0.0	0.0	0.2	1.5	31.9	33.8	2.8	0.5
	batch-256	0.3	0.5	0.0	0.0	0.0	0.3	0.7	64.8	75.2	5.1	0.8
	batch-512	0.5	0.8	0.0	0.0	0.0	0.6	1.0	115.1	140.0	8.3	0.8
	batch-1024	0.6	1.8	0.0	0.0	0.1			225.3	249.2	17.4	1.5
	batch-2048	1.1	3.2	0.0	0.0	0.2			562.7	497.2	37.1	2.9
	batch-4096	4.6	5.5	0.0	0.0	0.4			1430.3	2044.2	69.6	5.5
SemaphoreNeighbor	batch-1	0.0	0.1	0.0	0.0	0.0	0.0	0.4	0.8	1.2	0.9	0.3
	batch-2	0.0	0.2	0.0	0.0	0.0	0.1	0.5	1.2	1.9	1.4	0.3
	batch-4	0.0	0.6	0.0	0.0	0.3	0.1	0.6	2.5	4.9	3.3	0.4
	batch-8	0.0	1.7	0.0	0.0	2.6	0.2	0.6	3.7	12.1	8.4	0.5
	batch-16	0.0	6.4	0.0	0.0	22.9	0.3	0.9	6.6	45.1	28.6	0.8
	batch-32	0.1	26.3	0.0	0.0	223.3	0.4	1.1	13.4	175.7	117.3	1.4
	batch-64	0.1	98.9	0.0	0.0	1897.6	1.1	1.9	24.1	673.5	426.2	2.5
	batch-128	0.1	429.4	0.0	0.0		1.2	3.7	46.1	2523.0	1689.2	4.2
	batch-256	0.3	2505.0	0.0	0.0		2.2	6.2	89.1			8.8
	batch-512	0.6		0.0	0.0		6.5	10.6	168.7			18.0
	batch-1024	0.8		0.0	0.0				358.4			33.5
	batch-2048	2.6		0.0	0.0				668.7			70.3
	batch-4096	4.2							1855.4			162.4
SwitchSensor	batch-1	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.6	0.8	0.2	0.1
	batch-2	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.9	1.1	0.2	0.1
	batch-4	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.7	1.8	0.2	0.1
	batch-8	0.0	0.0	0.0	0.0	0.0	0.0	0.4	2.8	2.8	0.3	0.1
	batch-16	0.0	0.0	0.0	0.0	0.0	0.0	0.4	5.1	5.9	0.4	0.1
	batch-32	0.0	0.1	0.0	0.0	0.0	0.1	0.5	9.3	9.6	0.6	0.2
	batch-64	0.1	0.1	0.0	0.0	0.0	0.1	0.8	17.2	18.9	0.9	0.2
	batch-128	0.1	0.1	0.0	0.0	0.0	0.1	0.4	30.1	32.0	1.2	0.2
	batch-256	0.3	0.2	0.0	0.0	0.0	0.2	0.6	65.1	64.3	2.2	0.4
	batch-512	0.4	0.5	0.0	0.0	0.0	0.5	0.9	123.9	117.5	3.8	0.6
	batch-1024	0.5	0.9	0.0	0.0	0.0			245.0	240.9	6.5	0.9
	batch-2048	1.1	1.6	0.0	0.0	0.1			482.0	475.0	12.2	1.5
	batch-4096	2.8	3.6	0.0	0.0	0.1			1422.8	1486.2	24.6	2.8
SwitchSet	batch-1	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.7	0.8	0.2	0.2
	batch-2	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.9	1.0	0.2	0.1
	batch-4	0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.8	1.9	0.3	0.2
	batch-8	0.0	0.0	0.0	0.0	0.0	0.0	0.4	2.7	3.2	0.4	0.2
	batch-16	0.0	0.0	0.0	0.0	0.0	0.0	0.5	5.1	4.8	0.5	0.2
	batch-32	0.0	0.1	0.0	0.0	0.0	0.1	0.6	9.0	9.5	0.7	0.2
	batch-64	0.1	0.1	0.0	0.0	0.0	0.1	0.8	17.5	17.8	1.0	0.2
	batch-128	0.1	0.2	0.0	0.0	0.0	0.2	1.2	31.1	30.2	1.3	0.3
	batch-256	0.3	0.4	0.0	0.0	0.0	0.3	0.6	60.8	64.0	2.5	0.4
	batch-512	0.4	0.6	0.0	0.0	0.0	0.4	1.0	118.9	120.1	3.7	0.5
	batch-1024	0.6	1.1	0.0	0.0	0.0			234.1	244.2	6.6	0.8
	batch-2048	1.1	1.9	0.0	0.0	0.0			478.0	475.9	13.6	1.5
	batch-4096	2.1	6.5	0.0	0.0	0.1			1444.6	1727.6	25.6	2.4

Table B.5: Execution time (seconds) for read phase (M1) in the Repair scenario.

Repair M1: read phase execution time average in seconds (s)

		XMI				MySQL	Neo4J		CDO			
Query	Model	EMF	OCL	IncQuery Local	IncQuery Incremental	SQL	Core API	Cypher	EMF	OCL Client-Side	OCL Server-Side	MQT-Engine EOL
PosLength	repair-1	0.1	0.1	0.1	0.1	0.1	0.7	1.2	0.9	0.9	0.8	0.5
	repair-2	0.1	0.1	0.2	0.2	0.1	0.8	0.8	0.8	0.9	0.8	0.6
	repair-4	0.2	0.2	0.3	0.2	0.2	1.1	1	0.8	0.8	0.7	0.5
	repair-8	0.2	0.2	0.5	0.4	0.2	1.5	1.6	0.8	0.9	0.7	0.6
	repair-16	0.3	0.3	0.8	0.7	0.2	2	1.8	0.8	0.8	0.7	0.6
	repair-32	0.5	0.5	1	1	0.4	3.4	3.2	0.8	0.8	0.8	0.6
	repair-64	0.7	0.7	1.5	1.7	0.6	5.3	4.7	1.3	1.5	1.3	1
	repair-128	1.3	1.2	2.4	2.5	1	9.8	8.9	1.4	1.5	1.3	0.9
	repair-256	2.4	2.2	4.3	4.5	2.1	21.9	22.6	1.2	1.2	1.1	0.7
	repair-512	4.1	3.9	7.4	7.3	4.3	51.4	46.3	2.3	1.9	1.9	1.4
	repair-1024	7.8	8	15.3	17.1	9.4			3.2	2.8	3.1	2.4
	repair-2048	17	18.8	33.4	33.5	20.1			3	2.8	3.5	3.2
	repair-4096	42.1	38.7	68.6	72.3	44.7					4.2	3.8
RouteSensor	repair-1	0.1	0.1	0.1	0.1	0.1	0.8	0.7	0.9	0.9	0.8	0.5
	repair-2	0.1	0.1	0.2	0.2	0.1	0.9	0.9	0.8	0.8	0.7	0.6
	repair-4	0.2	0.2	0.2	0.2	0.1	1	1	0.8	0.7	0.6	0.5
	repair-8	0.2	0.2	0.4	0.5	0.2	1.4	1.4	0.8	0.7	0.7	0.4
	repair-16	0.3	0.3	0.8	0.7	0.2	1.9	1.9	0.9	0.8	0.7	0.6
	repair-32	0.5	0.5	1.3	1.2	0.3	3.1	3	0.9	0.8	0.5	0.5
	repair-64	0.7	0.8	1.7	1.8	0.5	5.1	4.7	1.4	1.4	1.3	0.7
	repair-128	1.2	1.2	2.7	2.7	1	10.7	9.3	1.4	1.5	1.2	1
	repair-256	2.2	2.3	5	4.9	2.1	23.1	22.3	1.2	1.1	1	0.7
	repair-512	4.2	4.4	10.1	11.9	4.3	50.2	45.3	2.2	1.8	2.1	1.3
	repair-1024	7.9	7.6	23.2	22.9	9.5			3	3.9	2.9	2.4
	repair-2048	17.6	16.5	50.4	51	20.3			3.1	3	3.1	3
	repair-4096	40.3	37.2	449.7	297.5	43.3					3.2	3.8
SemaphoreNeighbor	repair-1	0.1	0.1	0.1	0.1	0.1	0.7	0.7	0.9	0.9	0.8	0.5
	repair-2	0.1	0.1	0.2	0.2	0.1	0.8	0.8	0.9	0.7	0.8	0.5
	repair-4	0.2	0.1	0.3	0.3	0.1	1	1	0.8	0.7	0.7	0.4
	repair-8	0.2	0.2	0.5	0.5	0.2	1.4	1.3	0.9	0.8	0.8	0.5
	repair-16	0.3	0.3	0.7	0.8	0.2	2	1.8	0.8	0.8	0.8	0.6
	repair-32	0.5	0.5	1.2	1.3	0.3	3.3	3	0.9	0.8	0.9	0.5
	repair-64	0.7	0.8	2.1	2.3		4.7	4.8	1.5		1.4	0.8
	repair-128	1.3	1.3	4.1	3.9		9.4	9.4	1.6			0.7
	repair-256	2.4		8.5	8.4		22.1	23.5	1.2			0.5
	repair-512	4.2		21	21.6		54	47.2	2.1			0.8
	repair-1024	8.6		47.6	48.2				3			2.8
	repair-2048	18.2							3.3			3.1
	repair-4096	40.6										3.7
SwitchSensor	repair-1	0.1	0.1	0.1	0.1	0.1	0.8	0.7	1.1	0.8	0.8	0.5
	repair-2	0.1	0.1	0.2	0.2	0.1	0.9	0.8	0.8	0.8	0.8	0.6
	repair-4	0.2	0.2	0.2	0.3	0.1	1.1	1	0.9	0.9	0.8	0.6
	repair-8	0.3	0.2	0.5	0.4	0.2	1.5	1.3	0.8	0.9	0.8	0.5
	repair-16	0.3	0.3	0.7	0.7	0.2	2.1	1.9	0.8	0.9	0.7	0.6
	repair-32	0.5	0.5	1.1	1.1	0.3	3.4	3.1	0.8	1	0.8	0.6
	repair-64	0.8	0.9	1.6	1.7	0.5	5.3	4.7	1.4	1.9	1.2	1.1
	repair-128	1.2	1.3	2.5	2.5	1	10.4	10	1.4	1.3	1.6	0.9
	repair-256	2	2.3	4.5	4.5	2	23.5	20.8	1.1	1.1	1	0.7
	repair-512	3.9	4.2	8.7	7.9	4.1	46.7	43.9	2.1	1.9	2	1.4
	repair-1024	7.5	7.8	18.6	18.3	8.9			2.8	3	3	2.5
	repair-2048	18.3	20.7	39.1	40.7	18.9			3.1	3	3.5	3.2
	repair-4096	41.1	41	78.9	88.4	44.1					3.7	4.4
SwitchSet	repair-1	0.1	0.1	0.1	0.1	0.1	0.9	0.7	0.9	1	0.8	0.5
	repair-2	0.1	0.1	0.1	0.1	0.1	0.9	0.9	1	0.8	0.7	0.5
	repair-4	0.2	0.1	0.2	0.2	0.1	1.2	1	0.8	0.8	0.6	0.5
	repair-8	0.2	0.2	0.4	0.4	0.2	1.6	1.4	0.8	0.8	0.7	0.4
	repair-16	0.3	0.3	0.6	0.6	0.2	2.4	1.9	0.9	0.8	0.7	0.6
	repair-32	0.5	0.5	0.8	0.8	0.3	3.7	3.2	0.8	1	0.6	0.5
	repair-64	0.7	0.7	1.3	1.2	0.5	6	4.8	1.4	1.4	1.4	1
	repair-128	1.3	1.2	2.1	2.2	1	13.4	9.6	1.3	1.4	1.3	0.6
	repair-256	2.2	2.4	4	4.1	2	25.5	20.8	1.1	1.1	1	0.7
	repair-512	4.1	4.2	6.8	6.9	4.2	54.9	44.7	2.1	1.8	1.8	1.4
	repair-1024	7.8	7.1	15.6	16	9.1			3	2.9	2.9	2.4
	repair-2048	19	16.4	32.1	32.4	19.9			3.1	3.2	3.2	3.4
	repair-4096	38	39.4	67.9	69.3	43.8					3.7	4.1

Table B.6: Execution time for check (M2, in seconds) in the Repair scenario.

Repair M2: check phase execution time average in seconds (s)												
		XMI				MySQL	Neo4J		CDO			
Query	Model	EMF	OCL	IncQuery Local	IncQuery Incremental	SQL	Core API	Cypher	EMF	OCL Client-Side	OCL Server-Side	MQT-Engine EOL
PosLength	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.7	1.1	0.4	0.2
	repair-2	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.1	1.5	0.7	0.3
	repair-4	0.0	0.0	0.0	0.0	0.0	0.0	0.3	2.0	2.0	0.9	0.3
	repair-8	0.0	0.0	0.0	0.0	0.0	0.0	0.3	3.3	4.4	1.5	0.5
	repair-16	0.0	0.1	0.0	0.0	0.0	0.0	0.4	5.7	7.5	2.2	0.7
	repair-32	0.0	0.1	0.0	0.0	0.0	0.3	0.5	11.3	13.7	4.1	1.0
	repair-64	0.1	0.2	0.0	0.0	0.0	0.1	0.7	21.6	24.2	6.6	1.6
	repair-128	0.1	0.3	0.0	0.0	0.0	0.2	1.1	39.8	48.8	12.1	2.8
	repair-256	0.2	0.6	0.0	0.0	0.0	0.3	0.8	76.7	92.9	24.8	5.0
	repair-512	0.3	1.2	0.0	0.0	0.1	0.4	3.0	159.9	167.3	52.2	8.0
	repair-1024	0.6	2.2	0.0	0.0	0.1			314.8	350.0	96.9	14.8
	repair-2048	1.0	3.6	0.0	0.0	0.2			576.4	744.6	179.1	28.1
	repair-4096	2.7	10.3	0.1	0.0	0.3					390.3	55.1
RouteSensor	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.8	1.0	0.3	0.2
	repair-2	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.4	1.3	0.3	0.3
	repair-4	0.0	0.0	0.0	0.0	0.0	0.0	0.3	2.2	2.1	0.4	0.3
	repair-8	0.0	0.0	0.0	0.0	0.0	0.0	0.4	3.9	4.0	0.7	0.2
	repair-16	0.0	0.0	0.0	0.0	0.0	0.1	0.4	6.1	6.8	0.8	0.4
	repair-32	0.0	0.1	0.0	0.0	0.0	0.1	0.5	11.9	12.4	1.3	0.4
	repair-64	0.0	0.1	0.0	0.0	0.0	0.1	0.7	21.2	21.5	2.3	0.6
	repair-128	0.1	0.2	0.0	0.0	0.0	0.2	0.6	42.4	42.8	4.1	0.9
	repair-256	0.2	0.4	0.0	0.0	0.0	0.5	0.6	81.6	87.8	7.1	1.5
	repair-512	0.3	0.6	0.0	0.0	0.1	1.2	3.0	159.0	159.5	14.6	2.1
	repair-1024	0.6	1.2	0.0	0.0	0.1			319.5	323.1	25.9	3.7
	repair-2048	1.3	2.0	0.0	0.0	0.2			648.2	645.7	50.8	6.9
	repair-4096	3.0	5.4	0.0	0.0	0.4					106.4	12.7
SemaphoreNeighbor	repair-1	0.0	0.1	0.0	0.0	0.0	0.1	0.4	1.0	1.3	0.9	0.3
	repair-2	0.0	0.1	0.0	0.0	0.1	0.1	0.5	1.7	2.2	1.6	0.3
	repair-4	0.0	0.4	0.0	0.0	0.4	0.1	0.5	2.8	4.5	3.4	0.4
	repair-8	0.0	1.3	0.0	0.0	3.1	0.1	0.6	5.3	15.9	10.5	0.6
	repair-16	0.0	4.7	0.0	0.0	24.4	0.2	0.7	9.0	34.9	31.8	0.9
	repair-32	0.0	21.1	0.0	0.0	224.6	0.2	1.0	18.1	155.2	127.2	1.3
	repair-64	0.1	77.8	0.0	0.0		0.6	1.5	34.1		438.5	2.4
	repair-128	0.1	321.9	0.0	0.0		1.2	2.7	65.2			4.4
	repair-256	0.2		0.0	0.0		4.4	5.6	128.8			8.1
	repair-512	0.4		0.0	0.0		9.7	12.9	255.8			15.5
	repair-1024	0.9		0.0	0.0				498.8			31.3
	repair-2048	1.6							1021.4			64.9
	repair-4096	3.6										132.5
SwitchSensor	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.8	0.9	0.2	0.1
	repair-2	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.2	1.3	0.2	0.1
	repair-4	0.0	0.0	0.0	0.0	0.0	0.0	0.3	2.3	2.4	0.3	0.2
	repair-8	0.0	0.0	0.0	0.0	0.0	0.0	0.3	3.9	4.1	0.3	0.1
	repair-16	0.0	0.0	0.0	0.0	0.0	0.0	0.4	7.0	6.5	0.4	0.2
	repair-32	0.0	0.0	0.0	0.0	0.0	0.1	0.5	13.0	12.5	0.6	0.2
	repair-64	0.0	0.1	0.0	0.0	0.0	0.1	0.7	22.5	22.4	1.0	0.3
	repair-128	0.1	0.2	0.0	0.0	0.0	0.1	0.4	43.1	46.7	1.4	0.3
	repair-256	0.2	0.2	0.0	0.0	0.0	0.2	0.4	88.9	89.5	2.5	0.5
	repair-512	0.4	0.5	0.0	0.0	0.0	0.2	0.5	161.3	182.2	4.3	0.7
	repair-1024	0.8	0.9	0.0	0.0	0.0			318.9	347.4	8.2	1.2
	repair-2048	1.0	1.4	0.0	0.0	0.1			649.6	668.8	15.7	2.2
	repair-4096	2.2	2.5	0.0	0.0	0.2					36.4	4.0
SwitchSet	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.9	1.0	0.2	0.2
	repair-2	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.4	1.3	0.2	0.2
	repair-4	0.0	0.0	0.0	0.0	0.0	0.0	0.4	2.3	2.4	0.3	0.3
	repair-8	0.0	0.0	0.0	0.0	0.0	0.0	0.4	3.9	3.9	0.4	0.2
	repair-16	0.0	0.0	0.0	0.0	0.0	0.1	0.5	6.1	6.8	0.5	0.3
	repair-32	0.0	0.0	0.0	0.0	0.0	0.1	0.5	12.8	13.4	0.8	0.4
	repair-64	0.0	0.1	0.0	0.0	0.0	0.1	0.7	22.5	22.8	1.2	0.5
	repair-128	0.1	0.2	0.0	0.0	0.0	0.2	0.5	45.3	45.6	2.1	0.7
	repair-256	0.2	0.3	0.0	0.0	0.0	0.3	0.6	90.2	91.2	3.2	1.2
	repair-512	0.3	0.4	0.0	0.0	0.0	0.3	0.7	165.6	187.3	5.8	1.9
	repair-1024	0.8	1.3	0.0	0.0	0.0			337.3	346.7	11.1	2.8
	repair-2048	1.2	1.7	0.0	0.0	0.1			664.9	684.6	20.1	4.9
	repair-4096	3.9	2.6	0.0	0.0	0.1					39.5	9.5

Table B.7: Execution time for re-check (M4, in seconds) in the Repair scenario.

Repair M4: re-check phase execution time average in seconds (s)

		XMI				MySQL	Neo4J		CDO			
Query	Model	EMF	OCL	IncQuery Local	IncQuery Incremental	SQL	Core API	Cypher	EMF	OCL Client-Side	OCL Server-Side	MQT-Engine EOL
PosLength	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0
	repair-8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0
	repair-16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.1
	repair-32	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.2	0.1
	repair-64	0.0	0.1	0.0	0.0	0.0	0.0	0.1	0.2	0.3	0.5	0.1
	repair-128	0.1	0.1	0.0	0.0	0.0	0.1	0.1	0.5	0.7	0.8	0.2
	repair-256	0.2	0.2	0.0	0.0	0.0	0.2	0.4	0.9	1.4	1.6	0.5
	repair-512	0.3	0.5	0.0	0.0	0.0	0.2	0.5	1.9	2.5	3.0	0.8
	repair-1024	0.6	0.9	0.0	0.0	0.1			3.4	5.7	6.5	1.5
	repair-2048	1.3	1.9	0.0	0.0	0.2			6.2	11.5	16.4	3.0
	repair-4096	2.1	4.0	0.1	0.0	0.3					43.8	6.3
RouteSensor	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0
	repair-16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.0
	repair-32	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.1	0.1
	repair-64	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.2	0.1
	repair-128	0.1	0.1	0.0	0.0	0.0	0.1	0.1	0.4	0.8	0.3	0.2
	repair-256	0.2	0.1	0.0	0.0	0.0	0.2	0.1	0.8	1.6	0.6	0.4
	repair-512	0.3	0.2	0.0	0.0	0.0	0.8	0.2	1.5	2.7	1.2	0.6
	repair-1024	0.6	0.4	0.0	0.0	0.1			2.9	5.3	2.4	1.2
	repair-2048	1.0	0.8	0.0	0.0	0.2			5.6	10.2	5.2	2.3
	repair-4096	2.0	1.5	0.3	0.0	0.4					9.9	4.7
SemaphoreNeighbor	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.0
	repair-2	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.7	0.0
	repair-4	0.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0	1.4	2.0	0.0
	repair-8	0.0	1.1	0.0	0.0	0.3	0.0	0.1	0.1	7.7	7.5	0.0
	repair-16	0.0	4.5	0.0	0.0	2.4	0.0	0.1	0.1	23.7	27.7	0.1
	repair-32	0.0	20.5	0.0	0.0	44.9	0.1	0.2	0.3	124.4	115.3	0.1
	repair-64	0.1	77.3	0.0	0.0		0.2	0.4	0.5		383.0	0.3
	repair-128	0.1	324.7	0.0	0.0		0.5	0.7	1.0			0.9
	repair-256	0.2		0.0	0.0		0.5	1.1	1.9			2.4
	repair-512	0.4		0.0	0.0		1.0	2.2	4.0			7.4
	repair-1024	0.7		0.0	0.0				7.9			30.4
	repair-2048	1.3							15.5			64.1
	repair-4096	2.6										130.6
SwitchSensor	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.0
	repair-32	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.0
	repair-64	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.1	0.1
	repair-128	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.4	0.4	0.1	0.1
	repair-256	0.2	0.0	0.0	0.0	0.0	0.1	0.1	0.7	0.8	0.2	0.2
	repair-512	0.3	0.0	0.0	0.0	0.0	0.1	0.2	1.3	1.4	0.3	0.3
	repair-1024	0.5	0.0	0.0	0.0	0.0			2.6	2.7	0.4	0.7
	repair-2048	1.2	0.1	0.0	0.0	0.1			5.2	6.0	0.9	1.2
	repair-4096	2.1	0.1	0.0	0.0	0.1					1.7	2.5
SwitchSet	repair-1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	repair-16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.0
	repair-32	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.1
	repair-64	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.1	0.1
	repair-128	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.4	0.5	0.1	0.1
	repair-256	0.2	0.1	0.0	0.0	0.0	0.0	0.1	0.8	0.9	0.2	0.2
	repair-512	0.3	0.1	0.0	0.0	0.0	0.1	0.1	1.5	1.6	0.4	0.3
	repair-1024	0.5	0.2	0.0	0.0	0.0			2.9	3.1	0.8	0.5
	repair-2048	1.1	0.4	0.0	0.0	0.0			5.8	6.8	1.4	0.9
	repair-4096	1.9	0.6	0.0	0.0	0.1					2.9	1.9