Efficient persistence, query, and transformation of large models

Gwendal Daniel

To cite this version:

Gwendal Daniel. Efficient persistence, query, and transformation of large models. Programming Languages [cs.PL]. Ecole nationale supérieure Mines-Télécom Atlantique, 2017. English. NNT : 2017IMTA0049 . tel-01668561

HAL Id: tel-01668561

https://theses.hal.science/tel-01668561

Submitted on 20 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE
BRETAGNE
LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

Gwendal DANIEL

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des Mines-Télécom Atlantique Bretagne Pays de la Loire
sous le sceau de l'Université Bretagne Loire

École doctorale : Mathématiques et STIC

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire des sciences du numérique de Nantes (LS2N)

Soutenue le 14 Novembre 2017
Thèse n° : 2017IMTA0049

Efficient Persistence, Query, and Transformation of Large Models

JURY

Rapporteurs :	**M^{me} Marie-Pierre GERVAIS**, Professeur des Universités, Université Paris Nanterre
	M. Jean-Michel BRUEL, Professeur des Universités, Université Toulouse 2 Jean Jaurès
Examinateurs :	**M. Sébastien GÉRARD**, Directeur de Recherche, CEA-LIST
	M. Jean-Claude ROYER, Professeur des Grandes Écoles, IMT Atlantique
Invité :	**M. Massimo TISI**, Maître Assistant, IMT Atlantique
Directeur de thèse :	**M. Jordi CABOT**, Professeur des Universités, Open University of Catalonia
Co-directeur de thèse :	**M. Gerson SUNYÉ**, Maitre de Conférence HDR, Université de Nantes

Acknowledgement

Foremost, I would like to express my sincere gratitude to my advisors Prof. Jordi Cabot, Dr. Gerson Sunyé, and Dr. Massimo Tisi. Their support, patience, and guidance helped me in all the steps of this thesis, and made me feel that our work was always going in the right direction.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Jean-Claude Royer, Prof. Jean-Michel Bruel, Prof. Marie-Pierre Gervais, Dr. Sébastien Gérard for their insightful comments on the present manuscript, as well as for the interesting questions and discussions that followed my thesis defense.

I thank my fellow team-mates for the stimulating discussions, the coffee-breaks, and all the fun we have had in the last three years.

Finally, I would like to thank Dr. Amine Benelallam together with Robin Boncorps for the joyful working moments as well as the friendly nightly discussions. You inspired me way more than you think!

To the Meat Boy Team
To my Friends
To my Family
To Malo
To *You*

Résumé Français

Introduction

Dans tous les domaines scientifiques, la modélisation est une activité commune qui vise à construire une vue abstraite simplifiant un système complexe de la réalité. Les modèles sont utilisés dans différents domaines d'études, tels que la biologie [1], le génie civil [2], ou la description de lignes de produits [91], et sont reconus comme une solution efficace pour appréhender des problèmes complexes et résoudre des questions spécifiques. Dans le domaine de l'ingénierie et du développement logiciel, les modèles sont utilisés pour décrire un système à développer, en représentant sa structure, ses composants, et sa logique. Ces modèles sont typiquement définis à l'aide de langages de modélisation, qui fournissent un ensemble règles permettant le partage de l'information entre les différents intervenants. Le langage UML (Unified Modeling Language) est un example de langage de modélisation largement adapté par l'académie et l'industrie, et qui a été standardisé par l'OMG (Object Management Group).

L'Ingénierie Dirigée par les Modèles (IDM) est une méthode de développement logicielle qui place les techniques de modélisations au centre du processus de développement. Les modèles deviennent des artefacts de premier ordre utilisés dans toutes les activités d'ingénierie, telles que le développement logiciel, mais également son évolution, ou la modélisation des exigences fonctionnelles et non fonctionnelles. Les modèles sont automatiquement traités par des transformations de modèles qui permettent de les rafiner afin de fournir différentes vues du système, générer des modèles d'implémentation spécifiques à des plateformes de déploiement, de la documentation, etc. L'IDM définit en général une dernière étape basée sur un transformation modèle vers texte, qui génère le code applicatif, les schemas des bases de données, ainsi que l'implémentation des invariants et règles métier.

La génération et l'extraction automatique de modèles sont des domaines particuliers de l'IDM permettant de construire des modèles à partir d'artefacts existants (code source [19], API web [56], etc). Les modèles obtenus sont ensuite utilisés pour assister le modeleur dans sa compréhension du système étudié, construire des vues précises, générer de la documentation, ou évaluer la qualité du système considéré. Ces techniques ont été popularisée par les techniques de rétro-ingénierie dirigée par les modèles, qui permettent de construire automatiquement un ensemble de modèle à partir d'une base de code. Ces modèles sont ensuite utilisés dans des processus complexes tels que l'évolution logicielle ou la restructuration de code source, qui sont typiquement exprimés par des langage de requêtage et de transformation de modèles.

Ces dernières années, l'IDM a été appliquée avec succès dans plusieurs scenario industriels. En effet, les études existantes [76, 54] reportent qu'utiliser les techniques d'IDM améliore la productivité et la maintenabilité des logiciels créés, tout en diminuant leurs coûts ainsi que les efforts nécessaires à leur construction. Cette intégration industrielle a notamment débouché sur la création de plusieurs plateformes de modélisations telles qu'EMF (Eclipse Modeling Framework) [102] et Papyrus [70], fournissant de solides bases pour construire, stocker, et requêter des modèles. Dans la communauté scientifique, l'IDM est reconnue comme un des sujets importants dans les conférences d'ingénierie logicielle majeures telles qu'ICSE [1] et ASE [2], et est le sujet principal de conférences et journaux reconnus tels que MoDELS [3] et SoSym [4].

Description de la problématique

Bien que l'IDM ait montré ses atouts pour améliorer les processus de développement logiciels, l'usage de plus en plus important de grands modèles complexes (en particulier dans des contextes industriels) a montré de claires limitations entravant son adoption [55, 68]. Les évaluations empiriques en situations industriels [117] ont en effet montré que l'une des principales raisons d'échec de l'intégration des techniques d'IDM est liée au manque de support pour le passage à l'échelle des outils existants.

En effet, les outils de modélisation développés ces 15 dernières années ont été conçus pour traiter des activités de modélisation basiques et mono-utilisateur, et n'ont pas été pensés pour supporter les modèles de grandes tailles utilisés de nos jours. Par exemple, le métamodèle BIM [2] définit un ensemble riche de concepts (environ 800) permettant de décrire précisément différents aspects d'un bâtiment ou d'une infrastructure. Les instances de ce métamodèle contiennent typiquement plusieurs millions d'éléments interconnectés, et habituellement stockés dans de larges fichiers monolithiques de plusieurs gigabytes.

Un exemple typique de problèmes de passage à l'échelle concerne la modernisation automatique de logiciels patrimoniaux basée sur des techniques de rétro-ingénierie dirigée par les modèles. Comme le montre la Figure 1.1, un processus de modernisation d'application dirigée par les modèles est définie comme une séquence d'opérations ayant pour but d'extraire un modèle représentant le logiciel existant (tels que son code source, ses fichiers de configurations, ou ses schémas de bases de données), puis effectuant une série de requêtes et de transformations dans le but de rafiner l'application existante. Enfin, une étape de génération (en général définie par une transformation de modèles) est utilisée pour créer —une partie de— la plateforme modernisée. Dans cet exemple, la taille de l'application à migrer peut être de taille arbitraire, et le passage à l'échelle des solutions techniques peut être une limitation majeure lorsque le processus est appliqué à de grandes bases de codes (contenant plusieurs millions de lignes de codes), et avoir des impacts à plusieurs étapes du processus: (i) l'environnement de modélisation doit permettre de stocker efficacement le modèle représentant l'application existante, (ii) les requêtes doivent être calculées sur les modèles créés efficacement, et (iii) les transformations doivent être effectuées de manière performante pour rafiner (potentiellement de manière répétée) les modèles existants vers l'application modernisée. Ainsi, un ensemble

---

1. `http://www.icse-conferences.org/`
2. `http://ase-conferences.org/`
3. `https://www.cs.utexas.edu/models2017/home`
4. `http://www.sosym.org/`

de solutions de modélisation prenant en charge les modèles de grande taille est nécessaire pour permettre d'appliquer les techniques de rétro-ingénierie dirigée par les modèles sur des applications patrimoniales de grande taille.

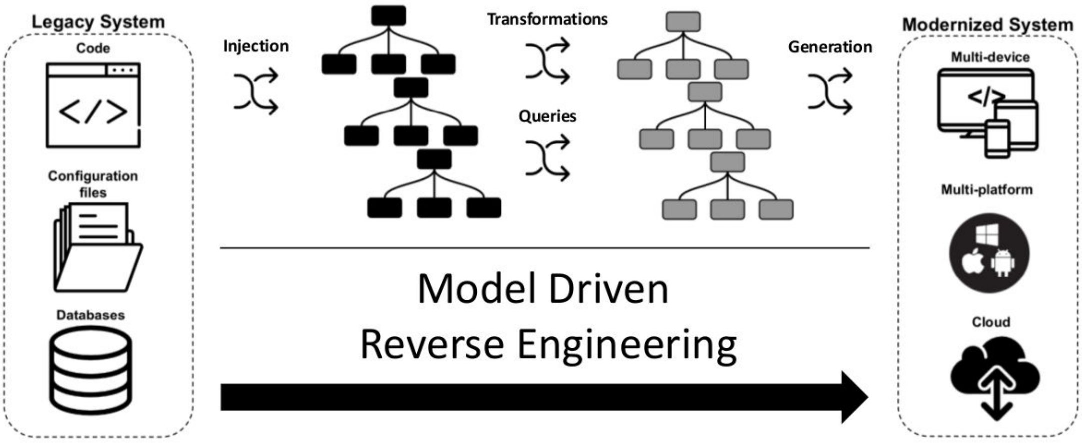


Figure 1 – Legacy System Modernization using MDRE Techniques

Dans cette thèse, nous nous concentrons sur deux problèmes majeurs afin d'améliorer la mise à l'échelle des solutions techniques existantes et permettre l'utilisation des techniques d'IDM dans des contextes industriels impliquant de larges modèles.

Mise à l'échelle des techniques de persistance de modèles Historiquement, la sérialisation sous forme de fichiers XML (eXtensible Markup Language) a été la solution privilégiée pour stocker et partager des modèles. Cependant, ce format a été conçu pour supporter des activités de modélisation simples telles que la création manuelle de modèles, et a montré ses limites dans le cadre de scénarios industrial actuels [48, 87] manipulant de larges modèles, potentiellement générés automatiquement [19]. En particulier, la représentation XML présente deux inconvénients majeurs limitant son efficacité dans le cadre de l'utilisation de grands modèles: (i) elle repose généralement sur de lourds fichiers nécessitant d'être intégralement chargés en mémoire pour être navigables, et (ii) elle offre un support limité au (dé)charger de fragments d'un modèle. Plusieurs solutions basées sur des bases de données relationnelles ou NoSQL [43, 87] ont été proposées pour résoudre ces limitations, mais elles se limitent généralement à fournir des améliorations génériques (comme des stratégies de chargements paresseux), et le choix de la base de données est totalement découplé de l'utilisation attendue du modèle. De fait, une solution donnée peut être appropriée à une tâche de modélisation spécifique, et inadaptée à une autre. De plus, les solutions de persistance actuelles manque en général de solutions avancées de mise en cache et de préchargement, qui pourraient être intégrées pour améliorer leurs performances.

Mise à l'échelle des techniques de requêtage et de transformation de modèles Le requêtage et la transformation de modèles sont les deux pierres angulaires des outils d'IDM, et plusieurs approches ont été conçues pour permettre leur définition et exploitation sur les platformes de modélisation existantes. Les infrastructures de requêtage et de transformation fournissent en général un langage de haut niveau (tel que le standard OCL

(Object Constraint Language)) qui est interprété en traduit en une séquence d'opérations déléguée à la plateforme de modélisation et finalement calculé par la base de données stockant le modèle. Bien que cette technique soit efficace lorsqu'elle est appliqué à des modèles sérialisés en XML, elle présente deux inconvénients majeurs lorsqu'elle est appliquée aux plateformes de modélisation actuelles: (i) les APIs de modélisations ne sont pas alignées avec les capacités de manipulation de données des solutions de stockage actuelles, limitant leur utilité, et (ii) un temps et une consommation mémoire importants sont nécessaire pour construire les objets intermédiaires qui peuvent être manipulés par ces APIs. De plus, les solutions de requêtage et de transformations actuelles sont en général implémentées en mémoire, et stockent des informations additionnelles (telles que les *traces* de transformation) qui posent des problèmes de consommation mémoire sur de grands modèles.

Pour résumer, dans cette thèse, nous soutenons que la taille et la complexité croissante des modèles est un problème majeur qui empêche l'adoption des techniques d'IDM dans l'industrie, et que de nouvelles approches permettant de stocker, requêter, et transformer ces grands modèles efficacement sont nécessaires. En particulier, l'alignement entre les solutions de stockage et les outils de modélisation doit être amélioré afin de permettre d'utiliser à leur plein potentiel les nouvelles générations de bases de données et leurs capacités de requêtage avancées.

Contributions

Pour pallier ces problématiques, nous proposont une nouvelle infrastructure de modélisation basée sur l'utilisation de base de données NoSQL et de leurs languages de requêtes avancés. La Figure 1.2 présente l'ensemble de nos contributions et montre comment ils interagissent entre eux pour créer un écosystème visant à stocker, requêter, et transformer efficacement de grands modèles.

Les prototypes développés à partir des approches présentées dans cette thèse sont construits sur l'infrastructure EMF, l'ecosystem standard *de-facto* pour la construction de langage dédiés et d'outils de modélisation dans l'environnement Eclipse. Des informations complémentaires sur l'intégration de nos solutions dans des solutions alternatives de modélisations sont fournis dans les chapitres correspondants.

- NEOEMF est notre solution pour améliorer le stockage et la manipulation de grands modèles. Notre approche définit une nouvelle plateforme de modélisation, intégrée de manière transparent aux outils EMF, et fournit un ensemble de base de données NoSQL qui peuvent être sélectionnées en fonction du scénario de modélisation attendu. NEOEMF est basé sur une architecture modulaire qui permet de facilement intégrer de nouvelles solutions de stockage, et fournit des mécanismes d'extensions réutilisés dans nos différentes approches afin d'améliorer l'efficacité du requêtage et des transformations de grands modèles.
- PREFETCHML est un langage dédié à la définition de règles de mise en cache et de pré-chargement sur un modèle. Ces règles sont combinées dans des plans qui peuvent être appliqués à des tâches de modélisation spécifiques. Les plans sont ensuite traités par un moteur responsable du chargement et déchargement des éléments du modèle, améliorant les performances lors des accès et le calcul de requêtes sur le modèle.

— MOGWAÏ est une nouvelle approche d'évaluation de requêtes basée sur un générateur de requêtes NoSQL à partir d'expressions définies en OCL. Notre solution se base sur les capacités de requêtage avancées des bases de données NOSQL (en particulier les bases de données en graphes) pour contourner les limitations des APIs des plateformes de modélisation actuelles. MOGWAÏ est intégré à NEOEMF , et nos expérimentation montrent des gains significatifs en terme de temps d'exécution et de consommation mémoire comparé aux solutions existantes.

— GREMLIN-ATL est une extension de notre approche de requêtage ayant pour objectif de supporter le calcul de transformations de modèles. Notre approche fournit un nouvel environement d'exécution de transformation qui peut être paramétré afin de supporter de grands modèles en stockant les informations de transformation dans une base de données dédiée, et fournit un ensemble de connecteurs permettant d'interfacer notre moteur sur différentes sources de données.

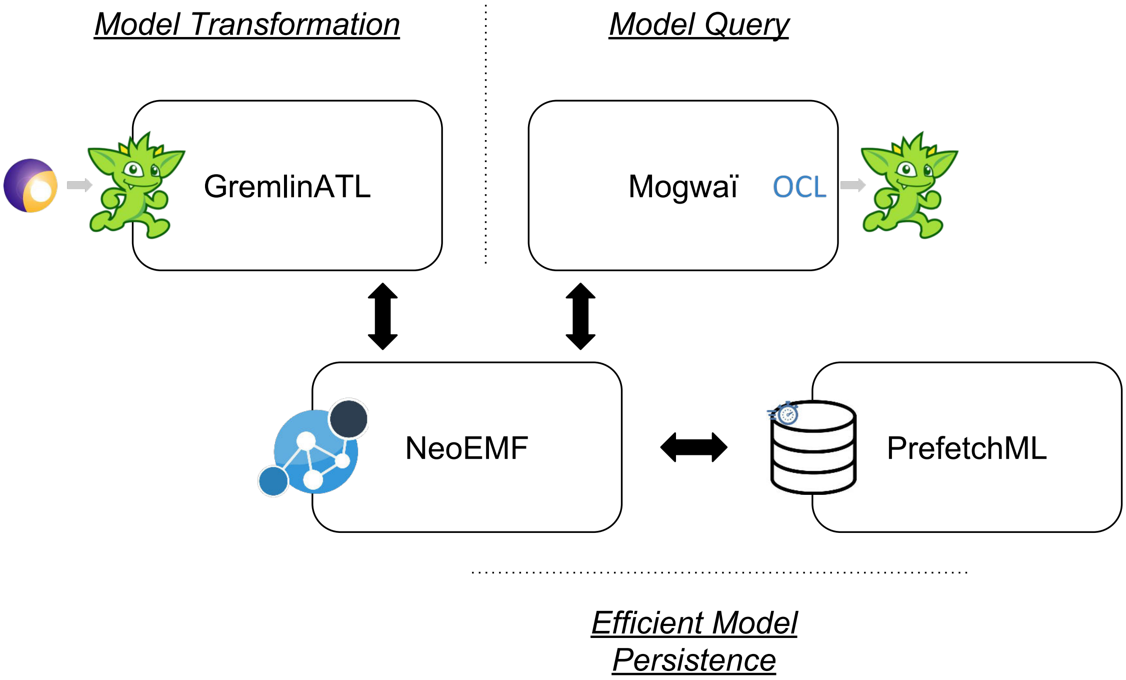


Figure 2 – NeoEMF Modeling Ecosystem

Outils et Résulats

Les approches présentées dans ce manuscrit sont implémentées sous forme de plugins Eclipse sous licence libre, et disponibles en ligne [5]. La documentation des différents outils ainsi que des tutoriels, guides d'utilisations, et ressources pour les développeurs sont disponibles sur les dépôts Github correspondants [6].

Dans ce manuscrit, nous évaluons la mise à l'échelle de nos solutions sur un ensemble de cas d'études reconnus dans les domaines de la rétro-ingénierie [19] et de l'industrie féroviaire [103]. Nous montrons qu'utiliser une base de données optimisée pour une activité de modélisation donnée permet d'augmenter significativement les performances en terme de temps d'exécution de consommation mémoire. De plus, nous démontrons l'intérêt des techniques de mise en cache et de préchargement et montrons qu'utiliser un plan adapté de préchargement permet d'améliorer les performances d'un scénario de modélisation jusqu'à 95%.

---

[5] `www.neoemf.com`
[6] `https://github.com/atlanmod`

Nous évaluons également nos techniques de requêtage et de transformation en nous basant sur des exemples industriels de rétro-ingénierie, et mettons en évidence qu'utiliser une traduction des languages de modélisation hauts niveaux (tels qu'OCL ou ATL) vers les languages de requêtage spécifiques aux bases de données permet d'améliorer significativement le temps d'exécution et la consommation mémoire.

Enfin, nous illustrons dans ce manuscrit comment nos approches peuvents être combinées afin de créer une solution permettant de réduire le fossé entre les techniques de modélisation conceptuelles et les bases de données NoSQL. Concrêtement, notre approche permet de générer une brique logicielle permettant d'accéder à une base de données en graphe et de vérifier automatiquement un certain nombre de contraintes d'intégrité à partir d'un schema conceptuel défini en UML et OCL. Pour cela, nous réutilisons le schema implicite défini dans NEOEMF permettant de sérialiser un modèle dans une base de données particulière, ainsi que notre approche de requêtage afin de générer des requêtes bas niveau permettant de vérifier les contraintes d'intégrité et d'exprimer les règles métier au niveau base de données.

Abstract

The Model Driven Engineering (MDE) paradigm is a software development method that aims to improve productivity and software quality by using models as primary artifacts in all the aspects of software engineering processes. In this approach, models are typically used to represent abstract views of a system, manipulate data, validate properties, and are finally transformed to application artifacts (code, documentation, tests, etc).

Among other MDE-based approaches, automatic model generation processes such as Model Driven Reverse Engineering are a family of approaches that rely on existing modeling techniques and languages to automatically create and validate models representing existing artifact. Model extraction tasks are typically performed by a modeler, and produce a set of views that ease the understanding of the system under study.

While MDE techniques have shown positive results when integrated in industrial processes, the existing studies also report that scalability of current solutions is one of the key issues that prevent a wider adoption of MDE techniques in the industry. This is particularly true in the context of generative approaches, that require efficient techniques to store, query, and transform very large models typically built in a single-user context.

Several persistence, query, and transformation solutions based on relational and NoSQL databases have been proposed to achieve scalability, but they often rely on a single model-to-database mapping, which suits a specific modeling activity, but may not be optimized for other use cases. For example a graph-based representation is optimized to compute complex navigation paths, but may not be the best solution for repeated atomic accesses. In addition, low-level modeling framework were originally developed to handle simple modeling activities (such as manual model edition), and their APIs have not evolved to handle large models, limiting the benefits of advance storage mechanisms.

In this thesis we present a novel modeling infrastructure that aims to tackle scalability issues by providing (i) a new persistence framework that allows to choose the appropriate model-to-database mapping according to a given modeling scenario, (ii) an efficient query approach that delegates complex computation to the underlying database, benefiting of its native optimization and reducing drastically memory consumption and execution time, and (iii) a model transformation solution that directly computes transformations in the database. Our solutions are built on top of OMG standards such as UML and OCL, and are integrated with the *de-facto* standard modeling solutions such as EMF and ATL.

1

Context

1.1 Introduction

Modeling is a common activity in all scientific disciplines that aims to build simplified and abstract views of a real-world situation for its systematic study. Models are used in various domains such as biology [1], civil engineering [2], product lines [91], and are recognized as a sound solution to understand complex problems and address specific questions. In the field of software engineering, they are widely used to describe a system under development, and can represent its structure, components, and logic. Software engineering models are typically expressed using modeling languages, defining a common set of rules that enables to share models between stakeholders. The Unified Modeling Language (UML) [95] is an example of such a modeling language that has been standardized by the Object Management Group (OMG).

The Model-Driven Engineering (MDE) is a software development method that puts modeling techniques in the center of the development process. Models become primary artifacts that drive all software engineering activities, including software development itself, but also evolution tasks and requirement modeling. Models are automatically processed using model transformations that refines them in order to provide views of the system, generate platform-specific models, or documentation. MDE processes typically define a final generation step relying on a model-to-text transformation that creates the final software artifacts, such as the application code, database schema, and constraints implementations.

Automatic model generation and extraction are particular fields in MDE that are used to construct models from existing artifacts (source code [19], web API [56], etc). The obtained models are used to help the modeler understand the system under study, build fine-grained views, generate documentation, or compute quality metrics. These techniques have been popularized by Model Driven Reverse Engineering (MDRE) approaches, that automatically build models representing an existing code base. The generated models constitute the input of complex processes such as software evolution tasks and source code refactoring which are typically expressed using model query and transformation languages.

In the last decade, MDE techniques has been successfully applied to several industrial scenario. As reported in existing studies [76, 54], using MDE techniques increases the productivity of software development compared to traditional methods, and improves the maintainability of the created software while decreasing the cost and effort to build it. This industrial adoption has leveraged the creation of several modeling platforms such as the Eclipse Modeling Framework (EMF) [102] and Papyrus [70] aiming at providing a strong foundation for building, storing, and querying models. In the scientific community, MDE is recognized as an important research topic in the major software engineering conferences such as ICSE [1] and ASE [2], and is the main topic of recognized conferences and journals such as MoDELS [3] and SoSym [4].

1.2 Problem Statement

While MDE pretended to be the silver bullet for software engineering, the growing use of large and complex models in industrial contexts has clearly emphasized serious limitations hampering its adoption [55, 68]. Existing empirical assessments from industrial companies adopting MDE [117] have shown that the limited support for large model management in existing technical solutions is one of the main factor in the failure of industrial MDE processes.

Indeed, modeling tools were primarily designed to handle simple modeling activities, and existing technical solutions are not designed to scale to large models commonly used and automatically constructed nowadays. As an example the BIM [2] metamodel defines a rich set of concepts (around 800) that describes different aspect of physical facilities and infrastructures. Instances of this metamodel are typically composed of millions of elements densely interconnected. The resulting models are stored in large monolitical files of several gigabytes, and cannot be processed efficiently by the current modeling infrastructures.

A typical example where scalability issues arise is the automatic modernization of legacy systems using MDRE techniques. As shown in Figure 1.1, a model-driven software modernization process is defined as a sequence of operations that first extract a model representing an existing software (such as its code base, configuration files, and database schemas), then defines a set of model queries and transformations aiming at refining the current application. Finally, a generation step (usually defined as a model transformation) is designed to create —part of— the modernized platform. In this example, the input software artifact to migrate can be of an arbitrary size, and the scalability of existing technical solutions can be a major limitation when the process is applied to systems involving very large code bases (such as several million lines of code), and impact multiple steps of the process: (i) the modeling framework needs to efficiently store the model representing the existing application, (ii) model queries should be computed effi-

---

1. http://www.icse-conferences.org/
2. http://ase-conferences.org/
3. https://www.cs.utexas.edu/models2017/home
4. http://www.sosym.org/

ciently to allow interactive querying, and (iii) model transformations should be computed efficiently to refine (potentially multiple times) existing models towards their modernized representation. Thus, a set of scalable modeling techniques is required to enable such refactoring operation when applied to large software artifacts.

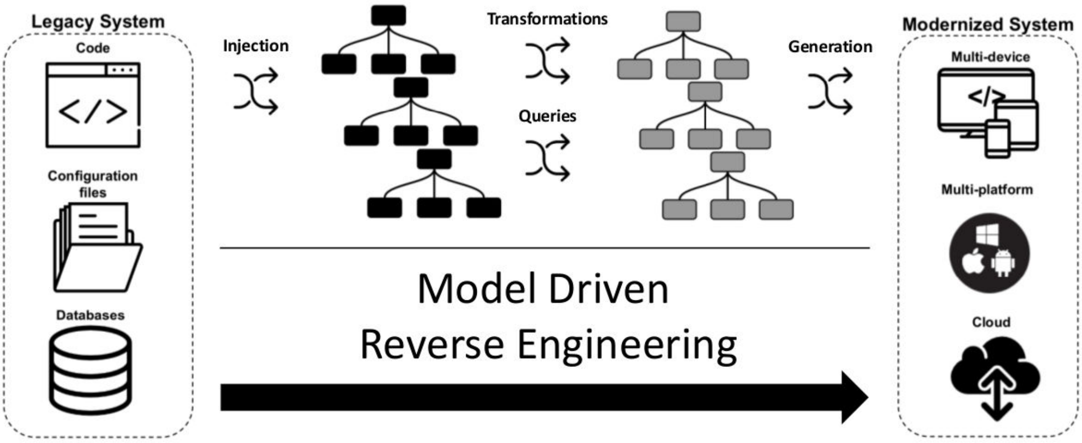


Figure 1.1 – Legacy System Modernization using MDRE Techniques

In this thesis, we focus on two major issues that have to be addressed in order to improve the scalability of existing technical solutions and enable industrial usage of MDE techniques applied to large models.

Model Storage Scalability In the last decade, filed-based EXtensible Markup Language (XML) serialization has been the preferred format for storing and sharing models. While this format was a good fit to support simple modeling activities such as human model sketching, it has shown clear limitations when applied to nowadays industrial use cases [48, 87], that typically manipulate large models, potentially automatically generated [19]. Indeed, XML-like representation usually rely on large monolithic files that require to be entirely parsed to be navigable, and provides limited support to partial loading and unloading of model fragments. Several solutions based on relational and NoSQL databases [43, 87] have been proposed to address this issue, but they often focus on providing generic scalability improvements (e. g. *lazy-loading* strategies), and the choice of the data-store is totally decoupled of the expected model usage. As a result, a given solution can fit a specific modeling scenario, and be unadapted for another one. Furthermore, existing model persistence frameworks typically lack advanced caching and prefetching mechanisms that could be integrated to improve their performance.

Model Query and Transformation Scalability Model queries and transformations are the cornerstones of MDE processes, and multiple approaches have been designed to compute them on top of existing modeling platforms. Model query and transformation frameworks typically provide a high-level language (such as the Object Constraint Language (OCL) OMG standard [83]) that is translated into sequences of modeling framework's API calls and computed by the underlying data-store. While this query computation technique is efficient on top of XML-based serialization platforms (because the entire model has to be loaded in memory), it presents two major drawbacks when applied to current scalable persistence solutions: (i) the modeling framework APIs are not aligned with the query capabilities of the data-store, limiting its benefits, and (ii) an important time and memory overhead is necessary to reify intermediate objects that can be manipulated using these APIs. In addition, current query and transformation solution typically store additional informations in-memory (such as transformation *traces*), that grow accordingly to the model size and limit their performances when applied to large models.

To summarize, in this thesis, we argue that the increasing size and complexity of models that is being experienced by the industry is an important issue that prevents the adoption of MDE techniques, and thus a novel generation of scalable approaches to persist, query, and transform large models is required. Specifically, the alignment between current data-storage solutions and existing modeling frameworks should be improved in order to fully benefit from the novel generation of databases and their advanced query capabilities.

1.3 Approach

One way to improve the support of existing modeling frameworks for large models is to rely on advanced storage mechanisms designed to handle large amount of highly interconnected data. The NoSQL movement is a family of storage techniques that aims to overcomes classical Relational Database Management System (RDBMS) issues (such as horizontal scaling and support for semi and unstructured data) by providing task-specific databases highly optimized for particular data processings. Our model persistence approach aims to benefit from NoSQL implementation specificities by integrating multiple data storage solutions designed to fit specific modeling tasks.

In order to improve the scalability of existing model persistence solution, we propose an approach based on existing prefetching and caching techniques that have been integrated for decades in relational database and file systems. We argue that bringing these low-level concepts at the modeling level can significantly improve the performances of I/O intensive applications, such as model validation and model transformation, and complements existing NoSQL storage solutions that typically lack such components.

Finally, to cope with scalability issues of model transformations and queries, we propose a novel approach based on a translation from high-level modeling languages into NoSQL-specific languages. Our model query and transformation environment relies on the advanced capabilities of NoSQL data-stores by generating efficient queries that are directly computed by the database, bypassing the modeling framework limitations and improving performances both in terms of execution time and memory consumption.

1.4 Contributions

Figure 1.2 summarizes the contributions of this thesis and shows how they are combined into an ecosystem designed to efficiently handle the storage, query, and transformation of large models. Note that all the presented contributions are open source and available online through the NeoEMF website[5], and additional links to code repositories and tutorials are provided in the corresponding chapters. The prototypes developed from the approaches presented in this thesis are built on top of the EMF infrastructure, the *de-facto* standard framework to build Domain Specific Language (DSL)s and modeling tools in the Eclipse ecosystem. Additional information on the integration of our techniques into alternative modeling solutions are provided in the related chapters.

---

[5] `www.neoemf.com`

— NEOEMF is our solution to improve the storage and access of large models. It is defined as a generic modeling framework that can be transparently plugged into the EMF platform, and provides a set of NoSQL database implementations that can be selected to suit a given modeling activity. NEOEMF is based on a modular architecture that can be complemented with additional model storage techniques, and provides extension mechanisms that are reused along this thesis to further improve performances of model query and transformation computations.

— PREFETCHML is a DSL that allows modelers to define prefetching and caching instructions over a model. The resulting PREFETCHML plan is processed by an execution engine that takes care of loading and unloading elements, speeding-up model accesses and query computation. Our approach aims to be generic, and can be applied on any persistence solution that provides an EMF compatible interface, and an advanced integration in NEOEMF has been proposed to further improve performances.

— MOGWAÏ is a novel model query approach that generates NoSQL database instructions from high-level model queries expressed in OCL. Our solution relies on the rich database query languages that are provided by NoSQL databases (in particular graph implementations) to bypass the modeling stack limitations. MOGWAÏ is natively integrated in NEOEMF , and our experiments show a significant improvement in terms of execution time and memory consumption when compared to state of the art solutions.

— GREMLIN-ATL is an extension of the MOGWAÏ approach that supports model transformation expressed in the AtlanMod Transformation Language (ATL). Our approach embeds a novel transformation execution engine that can be parameterized to scale to large models by storing transformation information in a dedicated data-store, and provides a set of low-level connectors that allow to compute transformations on heterogeneous data-sources.

1.5 Outline of thesis

The rest of this thesis is structured as follow:

Chapter 2 introduce the basic concepts that are required to grasp the remaining of the thesis. We start by presenting the MDE paradigm and its main components and standard languages. Then, we introduce NoSQL databases, their data representation strategies, and the query languages they embed to access stored information.

Chapter 3 presents our conceptual solution to store large models in NoSQL databases. We introduce the different model-to-database mapping we have defined, and discuss their benefits and drawbacks. The chapter also provides guidelines on which data-store to choose according to a specific modeling task to execute. Finally, we present the implementation of these concepts in the NEOEMF model persistence framework, and evaluate it on a set of industrial case studies.

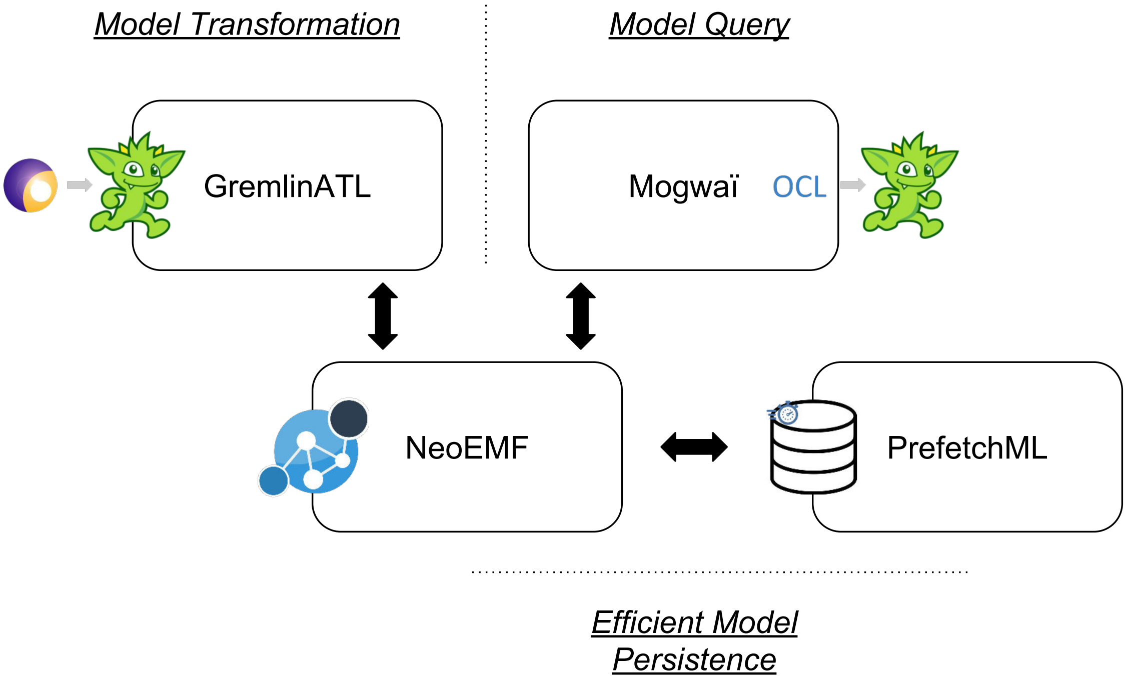


Figure 1.2 – NeoEMF Modeling Ecosystem

Chapter 4 presents our approach to enhance the efficiency of model persistence solutions by applying prefetching and caching techniques at the metamodel-level. We introduce a novel DSL that allows to express fine-grained prefetching and caching rules based on modeling events, and an execution environment that computes the rules according to the model usage. We introduce the PREFETCHML framework that implements our approach and show how it can be integrated in existing modeling applications. Finally, we evaluate our solution on top of well-known model queries and discuss its benefits and trade-offs.

Chapter 5 details MOGWAÏ , our scalable query approach relying on an OCL-to-Gremlin transformation that takes as its input an OCL expression and generates a low-level graph database query. We evaluate the efficiency of our approach on a set of well-known MDRE queries and show that our solution can drastically improve query execution performances on top of large models.

Chapter 6 extends the work presented in Chapter 5 with support for model-to-model transformations. We show how model transformation language constructs are integrated in our query generation process. The chapter also introduce our novel scalable transformation engine, and a set of data-store connectors that allow to perform model transformations on top of various persistence solution.

Chapter 7 presents an example that integrates our solutions to build a framework aiming at generating NoSQL application code from conceptual schemas. We show how the implicit model-to-database mappings defined in NEOEMF can be reused to translate schema structure into database access code, and how the OCL-to-Gremlin transformation embedded in MOGWAÏ can be used to generate code that dynamically checks constraints and invariants in the generated application.

Finally, Chapter 8 concludes this thesis by summarizing the key points and contributions, and describes our perspectives and future work.

1.6 Scientific Production

During this thesis, we have produced 9 articles (8 are currently published, 1 under review): 4 international conferences, 3 international workshops, and 2 journals

- Journals
  1. **Daniel, G.**, Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., & Cabot, J. NeoEMF: a Multi-database Model Persistence Framework for Very Large Models. In Science of Computer Programming (SCP), 2017. Elsevier Publishing.
  2. **Daniel, G.**, Sunyé, G., Cabot, J. Advanced Prefetching and Caching of Models with PrefetchML. *Submitted* to Software and Systems Modeling (SoSym), 2017. Springer Publishing.
- International Conferences
  1. **Daniel, G.**, Sunyé, G., & Cabot, J. (2016, June). Mogwaï: a Framework to Handle Complex Queries on Large Models. In Proceedings of the IEEE Tenth International Conference on Research Challenges in Information Science (RCIS) (pp. 1-12). IEEE Publishing.
  2. **Daniel, G.**, Sunyé, G., & Cabot, J. (2016, October). PrefetchML: a Framework for Prefetching and Caching models. (*distinguished paper award*) In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS) (pp. 318-328). ACM Publishing.
  3. **Daniel, G.**, Sunyé, G., & Cabot, J. (2016, November). UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases. In Proceeding of the 35th International Conference on Conceptual Modeling (ER) (pp. 430-444). Springer International Publishing.
  4. **Daniel, G.**, Sunyé, G., Jouault, F., Cabot, J. (2017). Gremlin-ATL: a Scalable Model Transformation Framework. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017. ACM Publishing.
- International Workshops
  1. **Daniel, G.**, Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., & Cabot, J. NeoEMF: a Multi-database Model Persistence Framework for Very Large Models. In Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS) (pp. 1-7). CEUR-WS.
  2. Brucker, A. D., Cabot, J., **Daniel, G.**, Gogolla, M., Herrera, A. S., Hilken, F., Tuong, F., Willink, E., & Wolff, B. Recent Developments in OCL and Textual Modelling. In Proceedings of International Workshop on OCL and Textual Modeling (OCL 2016) (pp. 157-165). CEUR-WS.
  3. **Daniel, G.** Efficient Persistence and Query Techniques for Very Large Models. In Proceedings of the ACM Student Research Competition (*3rd place*) co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS) (pp. 17-23). CEUR-WS.

1.7 Awards

1. **Distinguished paper award** at MoDELS 2016 for PrefetchML: a Framework for Prefetching and Caching models.
2. **3rd place** at the ACM Student Research Competition (co-located with MoDELS 2016) for Efficient Persistence and Query Techniques for Very Large Models.

2

Background

In this chapter we introduce the background and the main concepts that constitute the basis of the contributions presented in this manuscript. We first present the basis of MDE approach and its core concepts, then we review the standard languages and technologies we use to build our MDE solutions. Finally, we introduce the NoSQL movement and the main database families it contains.

2.1 Model-Driven Engineering

Traditionally, models were used as initial design sketches mainly aimed for communicating ideas among engineers. On the contrary, the MDE approach has emerged as a generalization of the Model Driven Architecture (MDA) [61] standard defined by the OMG [80] that promotes the use of models as a primary artifacts that drive all software engineering activities (i. e. not only software development but also also evolution, non-functional requirements modeling, traceability). These models are then refined to create specific views of the system to construct, generate platform-dependent software artifacts, and can be used in a final refinement step that produces (part of) the application code.

MDE has proven to be a powerful systems engineering approach in many different architecture domains, and existing studies have reported its benefits in terms of productivity and maintainability compared to traditional development processes [55]. As a result, the MDE methodology is progressively adopted as a valuable software development methodology in several companies, such as Sony Ericsson and Thales [117, 14].

A typical example of the successful application of MDE principles is the model-based modernization of legacy systems (exemplified in Figure 2.1). A software modernization process follows a systematic approach that starts by automatically building high-level abstraction models from source code (*injection*). This initial task is usually performed in a single-user context, where a modeler define the system to study and setup a set of modeling tools to store and access the created models. The injection frameworks analyzes

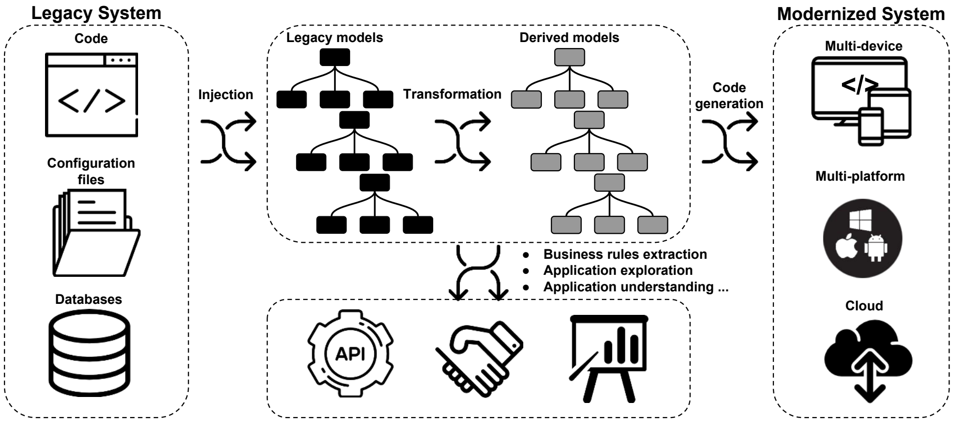

Legacy System
Code
Configuration files
Databases
Injection
Legacy models
Transformation
Derived models
Code generation
Modernized System
Multi-device
Multi-platform
Cloud
Business rules extraction
Application exploration
Application understanding ...
API

Figure 2.1 – Modernization process of legacy applications

the configured system, creates a set of –potentially large– models, and stores them in the preset modeling environment. Thanks to these models, engineers are able to *understand*, *evaluate*, and *refactor* legacy enterprise architectures. Using these intermediate models, engineers are able to partially generate modernized code for a specific platform (*code generation*).

The MDE methodology relies on three fundamental concepts: *metamodels* that represent how a model is structured and what are the possible interactions between its elements, *models* that conforms to a given metamodel, and represent a particular instance of it, and *model transformations*, which constitute the operational part of MDE processes that are used to refine models and generate code. In the following we detail these core MDE concepts.

2.1.1 Models, Metamodels, Model Transformations

Models A model is a (partial) representation of a system/domain that captures some of its characteristics into an abstraction that can be easily understood and manipulated by designers. Models are defined using a formal or semi-formal modeling language (such as the UML [86]) that provides a common vocabulary to ease communication between modelers. Models are used in various engineering fields, such as civil engineering [2], automotive industry [11], or biology [1], and are usually designed for a specific purpose (analysis, refactoring, maintenance, etc). In the context of MDE, models are used to drive all software engineering activities and are considered as the unifying concept between technologies and languages [13].

Metamodels A metamodel defines the set of concepts, relationships, and semantic rules regulating how models can be denoted in a particular language definition. A model which *conforms to* a given metamodel is an *instance* of it that satisfies all these rules. Metamodels are themselves described as models, easing the exchange of user models between different modeling tools by providing a detailed specification of its content. Note that a metamodel can also be seen as an abstract syntax, that can be complemented with one or more concrete syntaxes (the graphical or textual representations that designers use to express models in that language) to provide a complete modeling language.

Models and metamodels are organized in multiple levels (also referred as the meta-modeling stack) that are related by the *conformance* relationship. In the MDE approach, this stack contains three levels, with a self-reflective metamodel level on top, called *meta-metamodel* level, that describes the concepts and rules of every metamodel. The MetaObject Facility (MOF) [82] is the standard meta-modeling architecture proposed by the OMG, that is built around a set of modeling standard, namely OCL [83] for specifying constraints on MOF-based models, and XML Metadata Interchange (XMI) [84] for storing and interchanging MOF-based models in XML.

Model Transformations A model transformation is a modeling operation that consists of the automatic production of one or more output models from one or more input models according to a transformation specification. Model transformations are specified at the metamodel level, and are executed by transformation engines on models conforming to these metamodels. They have been standardized through the Query/View/Transformation (QVT) OMG specification [85], that formally describes model transformations, and has been implemented in a plethora of model transformation languages and frameworks [67, 58, 118].

Model transformations can be expressed through various technologies, from general purpose programming languages such as Java to dedicated transformation DSLs like the ATL [58], and can either define a *declarative*, *imperative*, or *hybrid* language. In the declarative style, only relationships between source and target model elements are specified, and no explicit execution order is given. The declarative style relies on rules to specify these relationships. In contrast, the imperative style explicitly specifies the execution steps. Finally, the hybrid style extends the declarative rules with optional imperative bodies. These three variants are defined in the QVT standard.

2.2 MDE Standards and Technologies

In the following we introduce an overview of the main standards and technologies that are used along this manuscript. Note that additional details on technologies and implementations related to a specific contribution are provided in the corresponding chapter.

2.2.1 UML/OCL

UML [86] is a standard language for object oriented modeling mainly used for software engineering. The first version of UML (0.9) has been defined in 1996 by the OMG as a standard aiming at the unification of existing modeling methodologies [95], such as Booch, OMT [94], and OOSE [57]. The latest version of the specification (UML 2.5) defines 14 diagrams which can be categorized in two families: (i) *static views* representing the structure of a system, including *Class*, *Package*, and *Object* Diagrams, and (ii) *dynamical views* emphasizing on the interaction between the system parts, inluding *Sequence* and *Activity Diagrams*, as well as *State Machines*.

UML class diagram has been widely adopted as the standard solution to define models, metamodels, and conceptual schemas. As an example, Figure 6.2 shows a class diagram representing an excerpt of a simple Java metamodel aiming to represent Java programs at a low-level of abstraction[1]. A *Package* is a named container that can recursively contain other *Packages* through its *subPackages* composition. A *Package* also contains several *Classes*, which define a *name* attribute and an *imports* association representing its imported *Classes*. Each *Class* contains a set of *Methods*. A *Method* is linked to a *Modifier* describing its *Visibility* (*public*, *private*, or *protected*), and a *returnType* that represents the *Type* that is returned by the method. Finally, a *Constructor* is a specialization of *Method*.

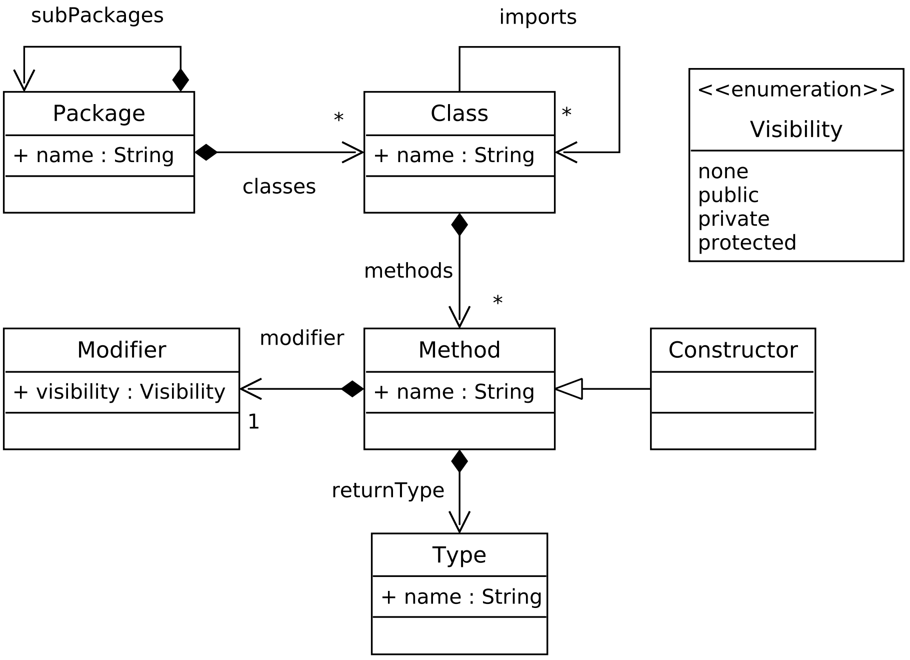


Figure 2.2 – A Simple Java Metamodel

Figure 2.3 shows an UML object diagram representing an instance of this metamodel. It contains a single *Package* instance `p1` named *package1* composed of two *Classes* `c1` and `c2`, respectively named *class1* and *class2*. The *Class* `c2` *imports* `c1`, and contains a single *Method* `m1` named *method1*. This *Method* is linked to a *private Modifier* (`mod1`) and the *void ReturnType* (`t1`). Note that these two examples will be used in the different chapters of this manuscript as running examples to illustrate our solutions to efficiently store, query, and transform large models.

UML diagrams can be complemented with OCL [83] expressions, an OMG standard allowing to define textual descriptions of invariants, operation contracts, derivation rules, and query expressions over models and metamodels. OCL is a declarative, side-effect free language that is used to extend UML diagrams — and any MOF model since the release of OCL 2.0 — with constraints and precise semantic that cannot otherwise be expressed by diagrammatic notation.

Each OCL expression is written in the *context* of an instance of a specific type, and defines the reserved keyword *self* to refer to the contextual instance currently manipulated. In addition, the language offers advanced support for collection operations, model navigations, and attribute value analysis. The complete reference of the language constructs

---

[1] The complete metamodel is available on MoDisco git repository at `http://git.eclipse.org/c/modisco/org.eclipse.modisco.git/tree/org.eclipse.gmt.modisco.java`

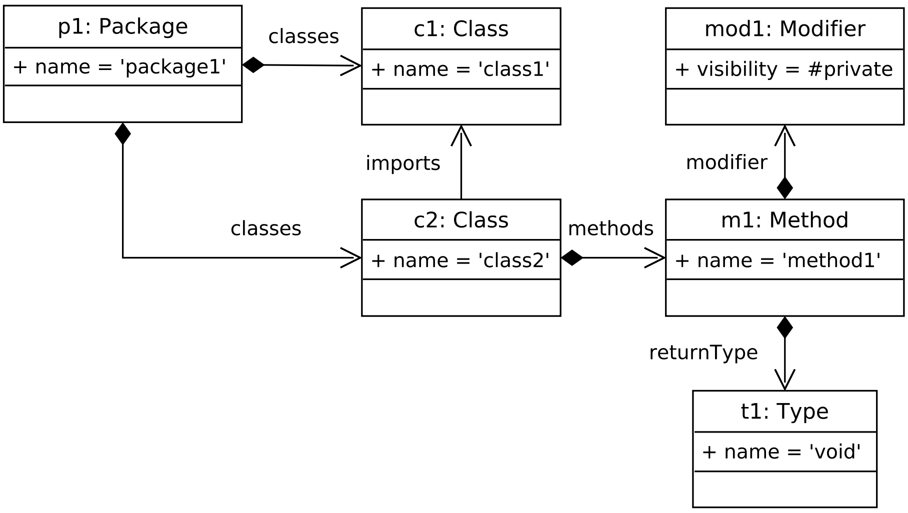


Figure 2.3 – A Simple Instance

is available online [2].

As an example, Listing 1 presents three OCL invariants that can be defined on our example metamodel presented in Figure 6.2. The first one, *validClassName* is a typical OCL constraint that ensures that a *Class name* is not empty. The second invariant (*packageHierarchyCycle*) is applied on *Package* instances, and checks that a *Package* does not contain itself in its *subPackages* association. Note that UML associations can be navigated in OCL using their label, and multi-valued expressions can be filtered according to the *select* operation that returns the elements of a collection satisfying a given condition. Finally, the third invariant (*invalidConstructorReturnType*) checks that a *Constructor* always returns a *Type* that has the same name as its containing *Class*. These expressions can be provided to an OCL evaluation environment such as Model Development Tools (MDT)-OCL [42] that takes care of the validation of a given model and returns the results to the modeler.

Listing 1 – Sample OCL Invariants

```
context Class
inv validClassName : self.name <> ''

context Package
inv packageHierarchyCycle : self.subPackages->select(p | p = self)->isEmpty()

context Constructor
inv invalidConstructorReturnType : self.returnType.name = self.class.name
```

2.2.2 Modeling Frameworks

Modeling frameworks are development platforms that offer technical solutions to create, manipulate, and persist (meta) models. They usually provide a low-level programming interface that allows to interact with models, and a set of high-level graphical tools that ease (meta) model definitions, constraint specifications, or model element creations. In addition, current modeling frameworks typically embeds a code generator that creates a set of software artifacts representing a given model that can be easily integrated in client applications.

---

[2] http://www.omg.org/spec/OCL/

In the last decade, EMF [102] has become the *de-facto* standard baseline framework to build DSLs and modeling tools within the Eclipse ecosystem. The growing popularity of EMF is attested by the large number of available EMF-based tools on the Eclipse marketplace [104], coming from both industry and academia.

EMF embeds Ecore as its own metamodeling language which consist of a subset of the UML class diagram, and thus, can be considered as the reference implementation of the Essential MOF language —a subset of MOF that closely corresponds to the facilities found in object oriented programming languages— proposed by the OMG. In the EMF environment, an Ecore model is a model of the classes of a software application (i.e. the structural description), and is used to generate Java code that allows to manipulate conceptual elements at the application level. This straightforward mapping between Ecore and Java allows to bring several benefits of modeling in standard Java development environment: part of the application code is generated by the framework, reducing development costs, and Ecore model updates can be safely propagated to the Java side thanks to the efficient code generation algorithm, improving maintainability of existing applications.

Alternative implementations of the MOF modeling stack have been proposed, such as Epsilon [65] or the Kevoree Modeling Framework (KMF) [46], that provide similar tooling to define, manipulate, and query models. While these approaches differ on specific features (multi-language integration for Epsilon and native model distribution for KMF), they are all based on a similar architecture, that provides a low-level, element-centered API to manipulate models, and compute queries and transformation by expressing them as sequences of atomic calls.

2.3 NoSQL Databases

The increasing popularity of web-based services, open API initiatives, and distributed cloud-based applications has emphasized the need to provide solutions to efficiently store and query the renowned *Big Data* [114], that describes large volume of heterogeneous data that are frequently updated and queried [120]. Practically, applications have to handle huge amounts of structured, semi-structured, and unstructured data that is fastly produced and updated, and does not have a fixed schema.

In the last decades, RDBMS have been the preferred solutions to store and query information. However, the strict relational schema has not been designed to handle efficiently this amount of data. Indeed, processing huge volume of unstructured information (such as JavaScript Object Notation (JSON) documents, Resource Description Framework (RDF) triples, etc) coupled with the frequent update and concurrent access operations has emphasized the limitations of RDBMS, that are not flexible enough to handle both reactivity and scalability required by Big Data applications. In addition, the distributed nature of current (cloud-based) applications requires high availability and concurrent read/write operations that are typically limiting RDBMS scalability.

These limitations has led to the popularization of the NoSQL movement that provides task-specific databases to overcome RDBMS issues in specific data processing contexts (such as querying highly interconnected data or semi-structured information). Compared to traditional relational databases that rely on the ACID consistency model that ensures database consistency using a sophisticated locking mechanism, NoSQL databases often rely on the BASE consistency model, that relaxes ACID properties (in particular full consistency at any time) to support other properties such as horizontal scaling, fault tolerance, or massive concurrency. This has lead to the development of several NoSQL database implementations, dedicated to specific tasks. They are usually classified into four categories based on their data model [77, 60]: *document* databases, *key-value* stores, *graph* databases, and *wide-column* stores. In the following, we introduce these categories by highlighting their specificities and the workflow they are designed to handle.

2.3.1 Key-Value Stores

Key-value stores such as Redis[3] and Amazon DynamoDB[4] rely on a simple data model that represent information using associative arrays. In a key-value store, a record is accessed using a unique identifier (the *key*). Store values can hold any kind of information, usually serialized as a byte array. This low-level representation makes the information opaque to the database system, meaning that no metadata or internal structures are maintained. The serialization/deserialization of the information is delegated to the application level.

Since values can contain any arbitrary information, key-value stores are by definition able to handle unstructured data, and are highly optimized to retrieve elements given their unique identifier. This simple data model also allows key-value stores to support high distribution features such as database partitioning, replication, and fault tolerance.

Compared to most of the RDBMS, key-value stores does not represent optional values with placeholders, limiting the memory required to store the same information, and improving input/output operations in data intensive workflows.

2.3.2 Document Databases

Document databases are a subclass of key-value stores that adds the concept of *document* to structure data on top of the raw key-value pairs representation. A document is stored as a first-class citizen in the database, and contains a set of key-value pairs defining its attribute. An attribute can be a primitive type supported by the database (e. g.integers, strings, or arrays), or a nested document, allowing to represent composition association between documents. Documents are organized in *collections*, that groups related information in order to reduce I/O operations and optimize query execution.

Most document databases implementations (such as Apache CouchDB[5] and MongoDB[6]) rely on a JSON-like data model, that easily supports data evolution such as the addition of new concepts, property updates, and multiple versions of the same concept. This data model is particularly popular in web-based applications, that are subject to data model changes among time, and where the speed of deployment is an important issue.

---

[3] `https://redis.io/`

[4] `https://aws.amazon.com/dynamodb/`

[5] `http://couchdb.apache.org/`

[6] `https://www.mongodb.com`

2.3.3 Column Databases

Wide column databases are another subclass of key-value stores that organize records in tables handling dynamic column definition. Originally designed by Google's Bigtable project [27], this database family is designed to store huge amount of structured data in a highly distributed environment. Wide column databases can be seen as bi-dimensional key-value stores: records are organized in tables, where each line is defined by a unique identifier, and specific attributes can be accessed using its containing column name. Columns are grouped into column families that are used to structure the database and improve access performance on columns frequently queries together, and support dynamic updates such as the addition or deletion of a specific column.

The distributed nature of wide column database makes them promising candidates for parallel and cloud-based data computation. They are used as the low-level persistence solutions of MapReduce [38] processings, a distributed programming model designed to compute queries on semi-structured data stored in large clusters.

Wide column databases (such as Facebook's Cassandra[7] and Apache HBase[8], the open source implementation of Bigtable) are used in social networks and large scale cloud-based applications, that typically require to store and process large amount of data efficiently. Some implementations, such as Cassandra, provide a high-level, Structured Query Language (SQL)-like query language that allows to query and update the store efficiently.

2.3.4 Graph Databases

Graph databases are a different kind of NoSQL database that provide graph primitives to store and structure information. In a graph database, records are stored as *nodes* and connected together using *edges*. Most of the current implementations (such as Neo4j[9] and Titan[10]) rely on a property graph data model which allows to store additional attributes as key-value pairs in nodes and edges. Graph databases are designed to handle complex hierarchy structures and highly interconnected data: the database engine is optimized to navigate efficiently from one node to another, and to compute complex navigation patterns involving large volume of nodes.

These complex navigation queries are expressed using dedicated query languages, that provide constructs to navigate a graph, match specific node and edge patterns, filter properties, etc. Cypher and Gremlin are two popular graph query languages: the former expresses graph queries using a sophisticated pattern matching language, the later describes finely graph traversals using navigation steps that can be combined into complex processings.

Note that graph databases and their query languages constitute the cornerstone of the model query and transformation approaches we present in this manuscript (Chapters 5 and 6), and additional details of their structure and language constructs are provided in the corresponding chapters.

---

[7] `http://cassandra.apache.org/`

[8] `https://hbase.apache.org/`

[9] `https://neo4j.com/`

[10] `http://titan.thinkaurelius.com/`

2.4 Conclusion

In this chapter we have introduced the main concepts and standards that constitute the basis of our work. We first introduced the MDE method, and we have shown that it has been succesfully applied in several industrial contexts with positive results in terms of productivity and maintainability.

Then, we have explored the core components of MDE techniques (models, metamodels, and model transformations), and we have shown how these concepts are articulated into a global process. Note that these components are defined at the same level of abstraction as the solutions proposed along this manuscript.

We have presented a set of standards and tooling approaches that are widely used in the community to represent model, constrain them, and query them. Among other solutions, we presented an overview of modeling frameworks, that constitute the basic component of nowadays MDE tools, and are also the core component of the solutions presented in this thesis. Note that the solutions presented in the remaining of this manuscript are largely based on EMF, the *de-facto* modeling framework in the Eclipse environment.

Finally, we have introduced the NoSQL database environment, and detailed the different data representation approaches they use. Note that this chapter introduce the four main families of NoSQL database, whereas the solutions detailed in the remaining of this manuscript only focus on three of them: graph, key-value, and wide-column databases. Further details on the selected database families can be found in Chapter 3.

Note that additional details related to the proposed contributions can also be found in the corresponding chapters, as well as the *state of the art* sections that explore, for each contribution, the existing work and extract a set of problematic and research challenges to tackle.

3

Scalable Model Persistence

Model persistence is one of the cornerstone in MDE processes: input models are loaded in memory from an existing source, navigated and updated using model queries and transformations, and stored into a dedicated format to be accessed later on by client applications or shared between modelers.

In the last decades, the progressive adoption of MDE techniques in the industry [117, 55] has emphasized the need to provide persistence solutions that are able to address scalability issues to store, query, and transform large and complex models. Indeed, existing modeling frameworks were first designed to handle simple modeling activities, and often relied on an XML serialization to store models. While this format is a good fit for small models, it has shown clear limitations when scaling to large ones [88].

To overcome these limitations, several persistence frameworks based on relational and NoSQL databases have been proposed [43, 88]. These solutions typically provide a intermediate mechanism to serialize in-memory models into an on-disk representation by describing a model mapping that allows to save and access model elements from a dedicated persistence solution (file, RDBMS, NoSQL databases, etc).

Most of the existing approaches rely on a *lazy-loading* mechanism, which reduces memory consumption by loading only accessed objects. While persistence framework have globally improved the support for large models in existing MDE toolchains, they are often tailored to a specific data-store implementation, and their integration usually implies to update the code base to integrate their advanced modeling API.

In this chapter we introduce NEOEMF , the first brick of our scalable modeling ecosystem (introduced in Section 1.4) that is able to store very large models in multiple databases, allowing designers to choose the one that fits a given modeling scenario. Our approach is based on a modular architecture allowing model storage into multiple data stores. NEOEMF provides three new model-to-database mappings that complement existing persistence solutions and enable model persistence in graph, key-value, and column databases, each one optimized for a specific modeling task. The framework provides two APIs, one strictly compatible with the EMF API, easing its integration into existing modeling applications, and an *advanced* API that provides specific features complementing the standard EMF API to further improve scalability of particular modeling scenarios.

The rest of this chapter is structured as follows: Section 3.1 provides an overview of existing model persistence solutions and emphasizes the existing issues we aim to address. Section 3.2 presents an overview of our solution's architecture and its core features. Section 3.3 introduces the datastores that are currently supported by our approach and the model-to-database mappings we have defined. Section 3.4 and 3.5 presents our implementation and evaluates it against state of the art solutions. Finally, Section 3.6 wraps up this chapter, draws conclusions, and discusses the benefit and drawbacks of our solution.

3.1 State of the Art

Since the publication of the XMI standard [84], file-based XML serialization has been the preferred format for storing and sharing models and metamodels. This choice was driven by the fact that modeling frameworks were originally designed to handle human-produced models, whose size does not cause significant performance concerns. However, the adoption of MDE practices in the industry [117] as well as the development of generative framework such as MDRE approaches [19] has popularized the need to handle large and complex (potentially generated) models, emphasizing XMI's limitations.

Indeed, XML-based serialization presents two drawbacks: (i) it sacrifices compactness in favor of human-readability and (ii) XML files need to be completely parsed and loaded in memory to obtain a navigational model of their contents. The former reduces efficiency of I/O accesses, while the later increases the memory required to load and query models, and limits the use of proxies and partial loading to inter-document relationships. In addition, XMI persistence layers do not provide advanced features such as transactions or collaborative edition, and large monolithic model files are challenging to integrate in existing versioning systems [3].

As a result, scalability of model persistence framework has been an active field of research in the last decade [68], and several approaches have been proposed to reduce their memory consumption and enable support for very large models. They can be classified into two categories based on their low-level model representations: (i) RDBMS-based solutions that store models in relational tables, and (ii) NoSQL solutions that uses semi-structured databases. Existing approaches usually expose an interface that is semi-compliant with the *de-facto* standard modeling APIs, and provide a *lazy-loading* mechanism which reduces the memory consumption by loading model elements from the datastore only when they are accessed.

3.1.1 Relational Persistence Layers

Historically, RDBMS have been the preferred solution to store large models. Existing approaches derive a relational schema from an existing metamodel, for example by creating tables to store the instances of each metamodel's class and columns for every class attribute. This schema is then used to store model elements, access attributes, or navigate associations using low-level query languages such as SQL. Existing frameworks implements the *de-facto* standard EMF API, and can be transparently integrated (once configured) into existing modeling applications to enhance their scalability.

The Connected Data Objects model repository (CDO) [43] was the first attempt designed to handle large models by relying on a client-server repository structure. A CDO application can connect to a CDO server using a specialized interface, and a dedicated implementation of the EMF API is provided to manipulate the model. CDO is based on a *lazy-loading* mechanism and supports transactions, access control policies, and provides a collaborative modeling environment allowing concurrent editing of a model. CDO's default implementation uses a relational database connector to serialize models into SQL compatible databases, but the modular architecture of the frameworks can be extended to support different data storage solutions. However, in practice only relational connectors are used and regularly maintained.

Teneo [106] is another approach based on relational databases to store EMF models. It relies on a dedicated mapping that allows to store EMF models using the Hibernate Object Relational Mapping (ORM) [5]. Teneo uses metamodel information to derive a relational schema and an EMF-compatible API that allows to access the model at a high level of abstraction. The Hibernate implementation provides an additional API to express model queries using the Hibernate Query Language (HQL) [1] query language, improving performance by lowering the level of abstraction. Teneo is embedded in the default Hibernate connector provided by CDO.

While these solutions have proven their efficiency w.r.t XMI-based implementations, the highly interconnected nature of models often requires multiple table join operations to compute complex model queries, limiting the performance both in terms of execution time and memory consumption [4]. In addition, the strict schema used in RDBMS makes them hard to align with metamodel updates which can define new types, associations, etc. Finally, the extraction of the relational schema from an existing metamodel requires to integrate platform-specific initialization code, that can be a limiting factor for the adoption of the solution into existing applications.

3.1.2 NoSQL Persistence Layers

NoSQL-based solutions have been proposed to tackle the limitations of relational databases to handle large models. The proposed approaches are based on the schema-less nature of NoSQL data-stores to handle metamodel modifications efficiently, and rely on the database's query performance to compute complex model navigations efficiently.

Morsa [88] is the first approach designed to take benefit of the scalability features provided by NoSQL document databases to store and access large models in an efficient way. As CDO, Morsa relies on a *lazy-loading* mechanism to limit memory consumption, and supports incremental updates. The framework is based on MongoDB, and uses the document hierarchy capabilities of the data-store to represent model elements and their associations. Morsa models can be created and accessed transparently using the standard EMF mechanisms. However, model queries have to be expressed using a dedicated query language —MorsaQL [89]— to fully benefit from the underlying data-store performances.

---

[1] http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch16.html

Mongo EMF [21] is another alternative to store EMF models in a MongoDB database. Mongo EMF provides the same standard API than previous approaches, however, according to the documentation, the storage mechanism behaves slightly different than the standard persistence backend (for example, for persisting collections of objects or saving bi-directional cross-document containment references). For this reason, Mongo EMF cannot be plugged without performing any modification to replace another data-store in an existing system.

Hawk [3] is a model indexer framework that stores models in graph data-stores and provides an efficient model query API. The framework allows modelers to define specific indexes that will be reused during the query computation to speed-up element and attribute access. While Hawk can be considered as a NoSQL persistence layer for large models, it is not designed to handle EMF-based query computation efficiently, and relies on the Epsilon Object Language (EOL) [66] to efficiently navigate and manipulate models.

EMF fragments [96] is another NoSQL-based persistence layer for EMF that aims to achieve fast storage of new data and fast navigation of persisted models. EMF fragments principles are simpler than in other similar approaches and reuse the existing proxy mechanism of EMF. In EMF fragments, models are automatically partitioned in several chunks (*fragments*). Unlike CDO and Morsa, the granularity of the lazy-loading mechanism is defined at the fragment level, that are entirely parsed and loaded when they are accessed. Another difference with other approaches is that additional information have to be specified in the metamodel to benefit from the partitioning capabilities of the framework. This approach makes EMF fragments both dependent on the quality of the provided partitions and the size of individual fragments.

EMFStore [63] is a model repository that relies on a client-server infrastructure to store models. The framework provides a default XMI implementation to manipulate standard EMF models, and a MongoDB connector that aims to manage larger models. EMFStore focuses on providing collaborative modeling support, using a git-like approach supporting model versioning, branching, and history tracking. Thus, scalability is not the primary objective of the framework, and the underlying data-stores are not optimized to support very large models.

These solutions typically improve the performance for storing and accessing large models when compared to relational database persistence layers. However, they also require a specific initial step to start the database server and open a connection before allowing model manipulations. In addition, the use of an additional query language is often required to fully benefit from the database capabilities. Finally, to the best of our knowledge only document and graph data-stores have been explored to store large models, while key-value and wide-column stores could also be interesting candidates because of their capabilities to store huge amount of semi-structured data.

3.1.3 Problematic & Requirements

In most of these approaches, scalability is achieved by using a client-server architecture that provides an additional API that has to be integrated in client code to access the model (e.g. to create the server, open a new connection, commit changes, etc). Furthermore, the choice of the data-store is totally independent of the expected model usage (for example complex querying, interactive editing, or complex model-to-model transformation): the persistence layer offers generic scalability improvements, but it is not optimized for a specific scenario. For example, a graph-based representation of a model can improve scalability by exploiting databases' facilities to handle complex relationships between elements, but will have poor execution time performance in scenarios involving repeated atomic value accesses, or a given model partitioning policy can be a good fit to a modeling task and be inefficient for another one.

Our previous work on model persistence [47, 48, 35] has shown that providing a well-suited data store for a specific modeling scenario can dramatically improve performance of modeling applications [98]. Based on these observations and the current state of the art in large model persistence, we define a set of requirements that have to be addressed in order to provide an efficient model persistence framework for large models. We divide these requirements into two categories: (i) *iteroperability requirements* that define the level of integration to make the solution usable in existing applications, and (ii) *performance requirements* that define a new solution's scalability performances w.r.t existing approaches.

Interoperability Requirements:

IR1 The persistence layer must be fully compliant with the modeling framework's API. For example, client code should be able to manage models persisted with an alternative persistence manager as if they were persisted using the standard serialization.

IR2 The underlying persistence data-store engine should be easily replaceable to avoid vendor lock-ins.

IR3 The persistence layer must provide extension points for additional (e.g., domain-specific) caching mechanisms independent from the underlying engine.

Performance Requirements:

PR1 The persistence layer must be memory-friendly, by using on-demand element loading and unloading unused objects from the memory.

PR2 The persistence layer must provide at least one model-to-database mapping that outperforms the execution time of current persistence layers when executing queries on large models using the standard modeling API.

In the following we introduce NEOEMF , our solution that addresses these issues by providing a modular, easy to integrate solution allowing designers to customize the model storage according to the expected modeling scenario. The framework embeds three new model-to-database mappings enabling model persistence in popular NoSQL databases. NEOEMF is released as a set of open-source Eclipse projects available online [2].

3.2 NeoEMF: a Multi-Database Persistence Framework

This section presents NEOEMF , our persistence framework that aims to manage large models efficiently by using task-specific data-stores. We first introduce an overview of the framework architecture, then we detail its main functionalities and the integration in the modeling ecosystem, and finally we introduce the advanced capabilities that allow to fully benefit from the underlying data-stores.

---

[2] www.neoemf.com

3.2.1 Architectural Overview

Figure 3.1 describes the architecture of NEOEMF in a typical modeling environment. Modelers typically manipulate models using *Model-based Tools*, which provide high-level modeling features such as a graphical interface, interactive console, or query editors. These features internally rely on a *Model Access API* to navigate the models, perform CRUD operations, check constraints, etc. The modeling framework delegates the operations to a persistence manager using its *Persistence API*, which is in charge of the (de)serialization of the model. This *Persistence API* can be complemented by a low-level connector that interacts directly with the data-store API. This generic architecture is used in popular modeling frameworks such as EMF and KMF [46] which typically provide a default XML connector to store models.

The NEOEMF *core* component implements the *Persistence API*, and provides a set of methods allowing the modeling framework to interact with it as a regular persistence layer. This design makes NEOEMF both transparent to the client application, and the modeling framework itself, that simply delegates the calls without taking care of the actual storage.

Once the NEOEMF *core* component has received the request of the modeling operation to perform, it forwards the operation to the appropriate *Backend Connector* (*/Map*, */Graph*, or */Column*), which is in charge of handling the low-level model-to-database mapping of the model. These connectors translate modeling operations into *Backend API* calls, store the results, and reify database records into high-level modeling framework elements when needed. NEOEMF also embeds a set of default caching strategies that are used to improve performance of client applications, and can be configured transparently at the EMF API level.

In addition to this integration in the classical modeling toolchains, NEOEMF also exposes an *advanced API* that provides additional operations to customize the data-stores, compute model queries efficiently, and define custom *caching* policies. This API provides methods that are typically missing or not efficient at current modeling frameworks (such as the `allInstances` operation [116]), a set of efficient model *importers* that are able to convert XMI files into NEOEMF databases with a low memory footprint, and additional APIs developed in the remaining of this manuscript to allow efficient model queries, transformations, and model prefetching.

3.2.2 Integration in the Modeling Ecosystem

As shown in Figure 3.1, NEOEMF is designed to be easily pluggable into existing modeling frameworks. Specifically, it provides an API that is compatible with EMF, the *de-facto* standard framework for building modeling tools. NEOEMF reimplements the EMF modeling API and ensures that calling a NEOEMF method produces the same behavior (including potential side effects) as standard EMF API calls. Note that EMF-based implementation is self-contained, and does not require specific code to setup a database server of start a transation. As a result, existing applications can easily integrate NEOEMF and benefit immediately from scalability improvements.

Precisely, NEOEMF supports all typical EMF features including: (i) a dedicated *code generator* that allows client applications to manipulate models using generated java

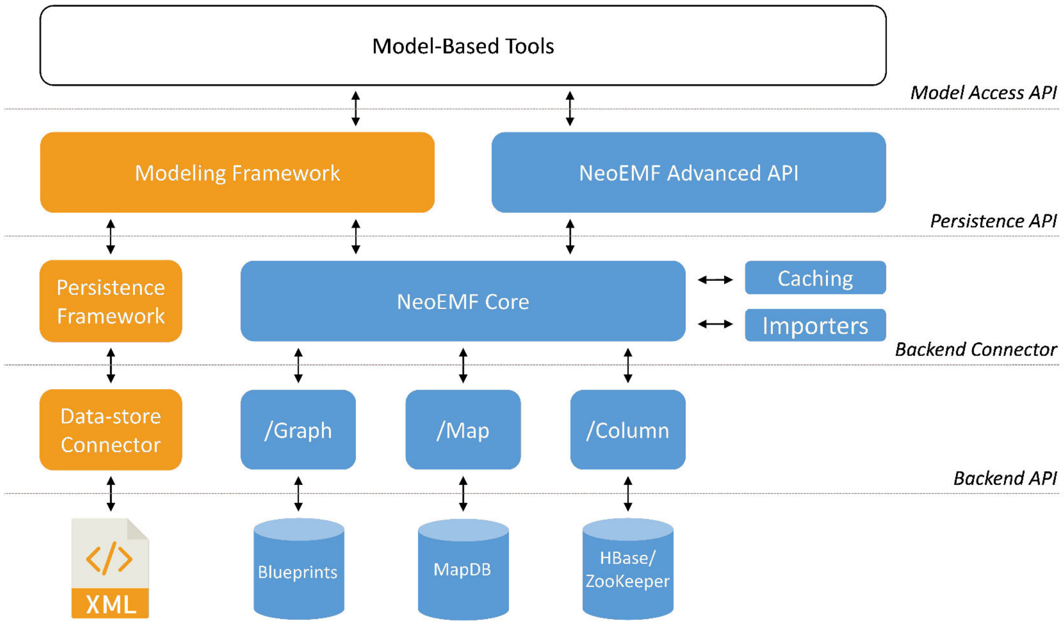

Model-Based Tools
Model Access API
Modeling Framework
NeoEMF Advanced API
Persistence API
Persistence Framework
NeoEMF Core
Caching
Importers
Backend Connector
Data-store Connector
/Graph
/Map
/Column
Backend API
XML
Blueprints
MapDB
HBase/ ZooKeeper

Figure 3.1 – NEOEMF Integration in the Modeling Ecosystem

classes, (ii) support of *Reflective/Dynamic EMF API*, and (iii) a *Resource API* implementation.

Figure 3.2 shows how NEOEMF (blue) is integrated in the EMF infrastructure (orange): the *PersistentResource* class implements the EMF *Resource* interface, which represent the global API to manipulate a model. A *PersistentResource* contains a set of *PersistentEObjects* through the *top-level elements* containment reference that corresponds to the root elements of the model. *PersistentEObject* implements the standard *EObject* interface, that provides a set of generic method to access model element's attributes, references, and containment hierarchy. A *PersistentResource* also embeds a *PersistentStore* that implements the *EStore* interface that defines the low-level persistence API that is internally used by EMF to persist models. The *PersistentStore* contains a *PersistenceBackend*, that maintain a connection to the data-store and translate EMF calls into database ones. Note that for the sake of clarity we did not put all the methods defined in the interfaces and their implementations, but the can be retrieved in EMF and NEOEMF online documentations.

As other model solutions, NEOEMF achieves scalability using a *lazy-loading* mechanism, which loads into memory objects only when they are accessed. *Lazy-loading* is defined at the *core* component: NEOEMF implementation of *EObject* consists of a lightweight wrapper delegating all its method calls to an *EStore*, that directly manipulates elements at the database level. Using this technique, NEOEMF benefits from datastore optimizations (such as caches or indices), and only maintains a small amount of elements in memory (the ones that have not been saved), reducing drastically the memory consumption of modeling applications.

Finally, NEOEMF provides a set of caching strategies that can be plugged on top of the data store according to specific memory and execution time requirements. For example, the framework provides caching capabilities for model element's attributes, reference

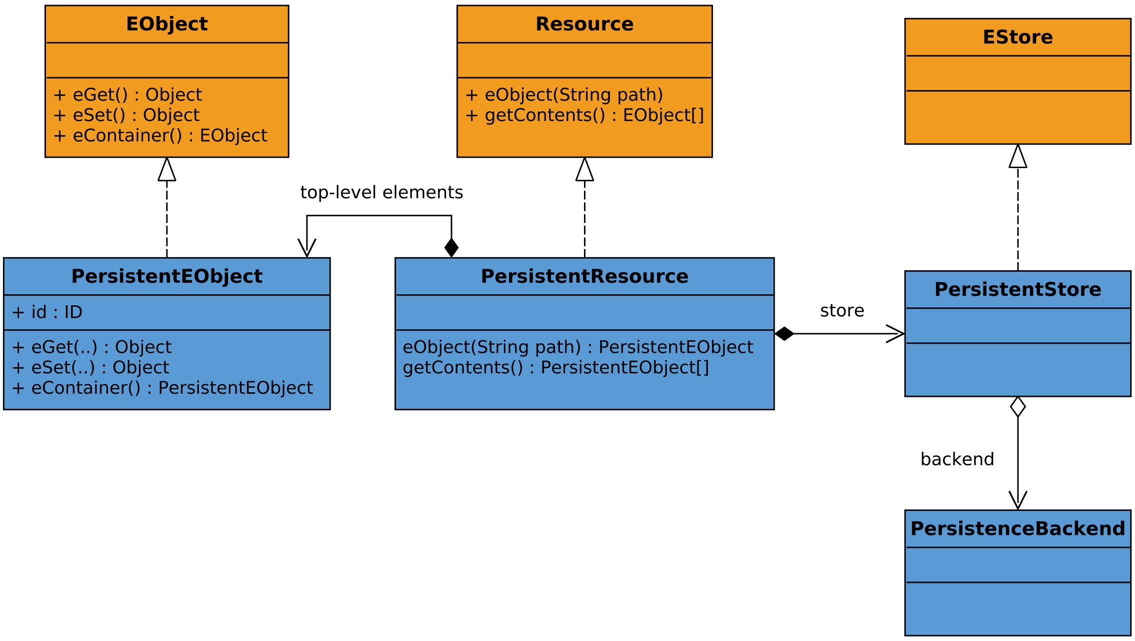

EObject
+ eGet() : Object
+ eSet() : Object
+ eContainer() : EObject
Resource
+ eObject(String path)
+ getContents() : EObject[]
EStore
top-level elements
PersistentEObject
+ id : ID
+ eGet(..) : Object
+ eSet(..) : Object
+ eContainer() : PersistentEObject
PersistentResource
eObject(String path) : PersistentEObject
getContents() : PersistentEObject[]
store
PersistentStore
backend
PersistenceBackend

Figure 3.2 – NEOEMF Integration in EMF Infrastructure

collections, or result of standard EMF operations such as *size* and *isSet*. These caching strategies are provided as *resource options*, and can be configured using a set of dedicated *option builders*. Note that additional *option builders* are available to tune the underlying data store.

3.2.3 Advanced Capabilities

In addition to its native integration in the modeling ecosystem, NEOEMF provides a set of utility features that bypasses modeling framework's limitations to further improve performances. These features are accessible using a dedicated *Advanced API*, and can be used to improve memory consumption and execution time of critical part of client application. While these features usually require to be integrated, they also provide connection points to the modeling framework ecosystem (e. g.by returning regular EMF objects) to limit the integration cost.

The *io* module is composed of a set of dedicated *importers* that are able to store XMI-based models into NeoEMF databases. They are based on an efficient XML parser, that is designed to limit the memory consumption. Compared to a standard model import using the EMF API, the *io* module operates directly at the database level, bypassing the EMF workflow that forces to fully load in memory the input XMI file into a navigable resource and transfer its content to a NEOEMF resource. The resulting database conforms to the internal NEOEMF mapping (detailed in Section 3.3), and can be loaded as a regular resource. Our experiments have shown that using the *io* module to import an existing model significantly speeds-up the execution time and reduces the memory consumption.

NEOEMF also provides utility methods to manipulate a model, such as the `allInstances` method, which is added on top of the standard *Resource* interface. This feature tackles the well-known performance issues of *allInstances* computation in EMF (which has to traverse the entire resource [116]) by delegating it to the data store, allowing to retrieve requested elements fast, using data store indices, or specific data representations.

Finally, NEOEMF embeds a *lazy model editor*, that provide a graphical interface to explore models stored in NEOEMF with a small memory footprint. Compared to classical model editors that typically load the entire model fefore allowing navigation and update operations, NEOEMF 's implementation heavily relies on the *lazy-loading* nature of the framework to only load in memory the elements that are visible in the editor. Elements are dynamically unloaded when they are not presented anymore (e. g.when the designer close a view or navigate in another part of the model). This editor can be used to navigate very large models transparently, and supports all the model to data-store mappings provided with NEOEMF .

3.3 Model-to-Database Mappings

The features presented in the previous section are defined at the *core* component level, and are available for a variety of data stores supported by NEOEMF . In this section we introduce the supported backends used to persist models: we first present their model to data-store mapping, then we detail how this mapping can be used to better address a specific modeling scenario. Note that both standard and advanced features presented in the previous section are implemented in all of them.

3.3.1 NeoEMF/Graph

NEOEMF/GRAPH is a graph-based connector designed to efficiently compute complex model navigations [6]. It relies on the graph database structures (detailed in Section 2.3) to represent models, where each element is represented as a database node, and connections between elements (associations, compositions, etc) are represented as edges. Using this graph-based representation, the NEOEMF/GRAPH connector benefits from the database engine that is designed to efficiently compute complex edge-based navigations. Our model query and transformation approaches (detailed in Chapter 5 and 6) are based on this particular connector, and leverage the capabilities of the graph representation to compute complex model queries and transformation efficiently.

Figure 3.3 describes how the instance model presented in the running example (Section 2.2.1) is persisted in a graph database using the NEOEMF/GRAPH connector. Figures 2.3, 6.2, and 3.3 show that:

— Model elements are represented as nodes. Nodes `p1`, `c1`, `c2` are examples of this, and correspond to the elements `p1`, `c1`, and `c2` shown in Figure 2.3.

— Element attributes are represented as properties stored in the node corresponding to the containing element. Node properties are represented using key-value pairs $\langle property\_name, property\_value \rangle$. For example, nodes `p1`, `c1`, and `c2` contain a `name` property that contains the value of the name attribute in Figure 2.3.

— Metamodel elements are also represented in the database as (grey) nodes, that are indexed to ease their access. Metamodel nodes contain two properties: the first one hold the name of the metamodel element, and the second one the metamodel unique identifier (*nsURI*). `Package` and `Class` are examples of metamodel nodes.

- Type conformance relationships are represented as `instanceof` edges between the node representing the metamodel element and the one representing the instance of thie particular type.
- References are represented as edges between the nodes corresponding to the connected elements. These edges are labeled with the name of the association defined in the metamodel, and can contain a position property that defines the index of the relationship if the base association is multi-valued. This is emphasized in Figure 3.3 by the two `classes` edges between `p1` and `c1`/`c2`. For example the edge `imports` that links `c1` and `c2` corresponds to the `imports` association in Figure 2.3. Note that EMF compositions are implicitly bidirectional, and are translated into one edge representing the composition itself, and an opposite one labeled `eContainer` (green edges in Figure 3.3) that represent this implicit containment link.

The resulting database contains all the model's information as well as part of the metamodel structure to optimize type based operations. For example, the *allInstances* method can be easily computed by searching the indexed node representing the metaclass to compute the instances of, and by navigating all its outgoing `instanceof` edges.

NEOEMF/GRAPH intensively relies on the interconnected nature of models to efficiently compute navigation queries. However, this representation also intensively uses node properties to store attribute information, that are typically costly to access in existing graph database implementations. In addition, the efficient navigation capabilities of the database engine are restricted by the high-level modeling framework APIs, that typically generate low-level fragmented queries that are hard to optimize and have a significant impact on the engine's performances.

3.3.2 NeoEMF/Map

NEOEMF/MAP is a key-value store connector designed to provide fast access to atomic operations, such as accessing a single element/attribute and navigating a single reference. Compared to NEOEMF/GRAPH , this implementation is optimized for modeling framework API-based accesses, which typically generate this kind of atomic and fragmented calls on the model. NEOEMF/MAP embeds a key-value store, which maintains a set of in-memory/on disk maps to speed up model element accesses. Our previous experiments [48] show that using this particular model to data-store mapping is the most suitable solution to improve performance and scalability of EMF API based tools that need to access very large models on a single machine.

We have designed the underlying data model of NEOEMF/MAP to reduce the computational cost of each method of the EMF model access API. The design takes advantage of the key-value nature of the underlying store to map each model element to a unique identifier that allows to retrieve a specific element efficiently using map-based lookups. Using this approach, the model structure is flattened into a set of key-value mappings that provides constant model element access time regardless its localization in the model.

NEOEMF/MAP uses three different maps to store models' information: (i) a *property map*, that keeps all objects' data (such as attributes and associations) in a centralized place; (ii) a *type map*, that tracks how objects interact with the metamodel-level (such as the instance of relationships); and (iii) a *containment map*, that defines the models'

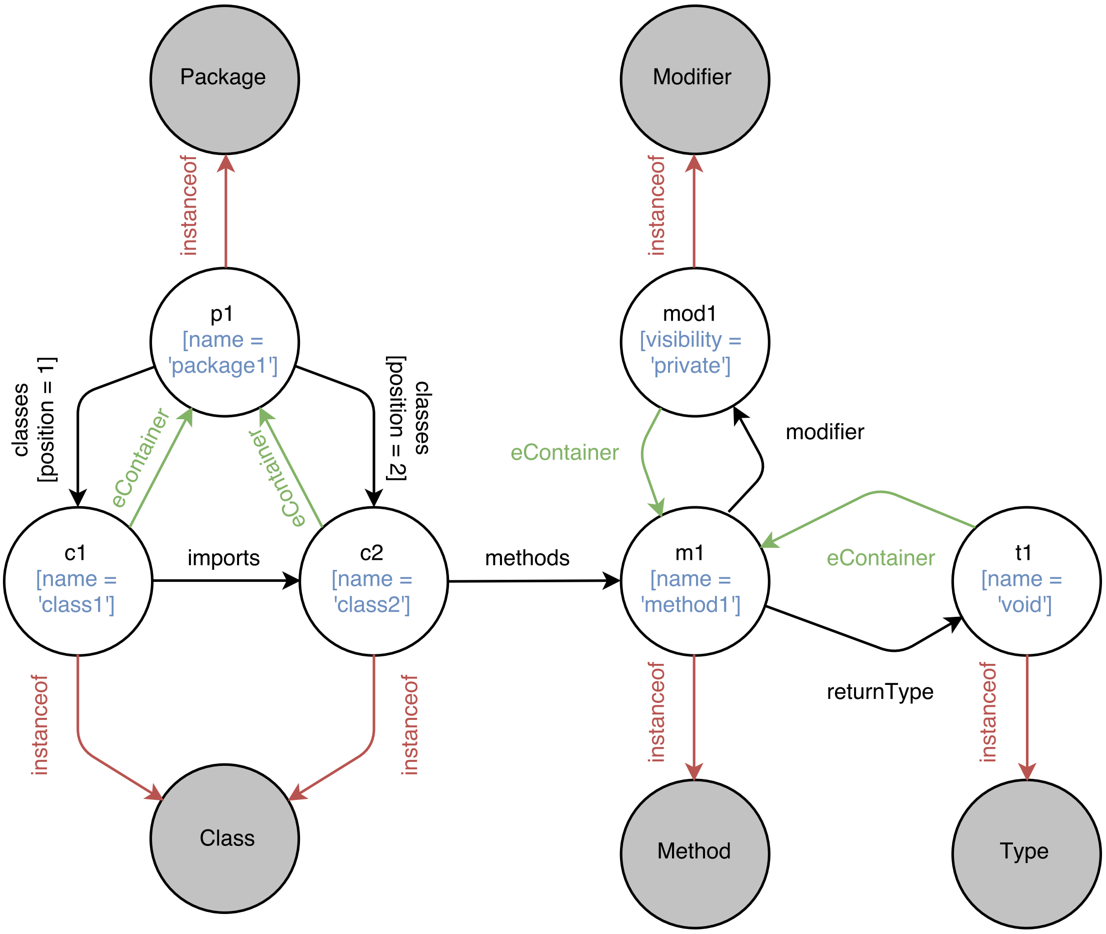

Package
Modifier
instanceof
instanceof
p1
[name =
'package1']
mod1
[visibility =
'private']
classes
[position = 1]
classes
[position = 2]
eContainer
eContainer
eContainer
modifier
c1
[name =
'class1']
imports
c2
[name =
'class2']
methods
m1
[name =
'method1']
eContainer
t1
[name =
'void']
returnType
instanceof
instanceof
instanceof
instanceof
Class
Method
Type

Figure 3.3 – Running example persisted in NEOEMF/GRAPH

structure in terms of containment references. Tables 3.1, 3.2, and 3.3 show how the sample model in Figure 2.3 is represented using a key-value structure.

As Table 3.1 shows, keys in the property map are a pair, the object unique identifier, and the property name. The values depend on the property type and cardinality (i.e., upper bound). For example, values for single-valued attributes (like the name of a Package) are directly saved as a single literal value as the entry $\langle\langle 'p1', 'name' \rangle, 'package' \rangle$ shows; while values for many-valued attributes are saved as an array of single literal values. Values for single-valued references, such as the `modifier` reference from `m1` to `mod1`, are stored as a single value (corresponding to the id of the referenced object). Finally, multi-valued references are stored as an array containing the literal identifiers of the referenced objects. An example of this is the `classes` reference, from Package to Class, that for the case of the `p1` object is stored as $\langle\langle 'c1', methods' \rangle, \{'c1', 'c2'\}\rangle$.

Table 3.2 shows the structure of the type map. The keys are again the identifier of the persisted objects and the values are named tuples containing the basic information used to identify the corresponding meta-element. For example, the second row of the table specifies that the element `p1` is an instance of the Package class of the Java metamodel (that is identified by the http://java nsUri). This map is used to efficiently compute the type of a given object and access its metamodel-related information.

Structurally, EMF models are trees (a characteristic inherited from its XML-based representation). That implies that every object (except the root object a.k.a top level container) must be contained within another object (i.e., referenced from another object via a containment reference). The containment map is the data structure in charge of maintaining a record of which is the container for every persisted object. Keys in the structure map are the identifier of every persisted object, and the values are named tuples that record both the identifier of the container object and the name of the property that relates the container object with the child object (i.e., the object to which the entry corresponds). Table 3.3 shows in the first row that, for example, the container of the Class `c1` is `p1` through the `classes` association.

Key	Value
$\langle 'p1','name'\rangle$	'package1'
$\langle 'p1','classes'\rangle$	'c1', 'c2'
$\langle 'c1','name'\rangle$	'class1'
$\langle 'c2','name'\rangle$	'class2'
$\langle 'c2','imports'\rangle$	'c1'
$\langle 'c2','methods'\rangle$	'm1'
$\langle 'm1','name'\rangle$	'method1'
$\langle 'm1','modifier'\rangle$	'mod1'
$\langle 'm1','returnType'\rangle$	't1'
$\langle 'mod1','visibility'\rangle$	'private'
$\langle 't1','name'\rangle$	'void'

Table 3.1 – Property Map

Key	Value
'p1'	$\langle nsURI =' http://java', class =' Package'\rangle$
'c1'	$\langle nsURI =' http://java', class =' Class'\rangle$
'c2'	$\langle nsURI =' http://java', class =' Class'\rangle$
'm1'	$\langle nsURI =' http://java', class =' Method'\rangle$
'mod1'	$\langle nsURI =' http://java', class =' Modifier'\rangle$
't1'	$\langle nsURI =' http://java', class =' Type'\rangle$

Table 3.2 – Type Map

Key	Value
'c1'	$\langle container =' p1', featureName =' classes'\rangle$
'c2'	$\langle container =' p1', featureName =' classes'\rangle$
'm1'	$\langle container =' c2', featureName =' methods'\rangle$
'mod1'	$\langle container =' m1', featureName =' modifier'\rangle$
't1'	$\langle container =' m1', featureName =' returnType'\rangle$

Table 3.3 – Containment Map

3.3.3 NeoEMF/Column

NEOEMF/COLUMN has been designed to enable the development of distributed MDE-based applications by relying on a distributed column-based datastore [47]. In contrast with NEOEMF/MAP and NEOEMF/GRAPH implementations, NEOEMF/COLUMN offers concurrent read/write capabilities and guarantees ACID properties at model element level. It exploits the wide availability and distributed nature of column stores to efficiently distribute intensive read/write workloads across datanodes.

NEOEMF/COLUMN uses a single table with three *column families* to store models' information: (i) a *property* column family, that maintains all objects' attributes and associations stored together; (ii) a *type* column family, that tracks how objects interact with the metamodel-level; and (iii) a *containment* column family, that defines the models' structure in terms of containment references. This column families have been designed to group information that is usually queried together in order to improve data distribution and reduce access time.

Table 3.4 shows how the sample instance in Figure 2.3 is represented using this structure: as in the NEOEMF/MAP mapping, row keys are a unique identifier that allows to retrieve a specific model element. The *property* column family stores the objects' actual data, where each column corresponds to a specific attribute or association. As it can be seen, not all rows have a value for a given column. Data representation depends on the property type and cardinality (i.e., upper bound). For example, values for single-valued attributes (like the name, which stored in the `name` column) are directly saved as a single literal value; while values for many-valued attributes are saved as an array of single literal values. Values for single-valued references, such as the modifier reference from `m1` to `mod1`, are stored as a single value corresponding to the identifier of the referenced object. Finally, multi-valued references are stored as an array containing the literal identifiers of the referenced objects. An example of this is the `classes` reference from `p1` to `c1` and `c2`, that is stored as { 'b1', 'b2' } in the $\langle p1, classes\rangle$ cell.

The *containment* column family maintains a record of which is the container for every persisted object. The container column records the identifier of the container object, while the feature column records the name of the containing association. Table 3.4 shows that, for example, the container of the Class `c1` is `p1` through the `classes` association.

The *type* column family groups the type information by means of the metamodel unique identifier and *EClass* columns. The former represents the metamodel the element is an instance from, and the later describe the specific class it conforms to. For example, the table specifies the element `p1` is an instance of the Package class of the Java metamodel (that is identified by its nsURI).

NEOEMF/COLUMN has been designed to benefit from the distributed nature of the underlying column store, and is the basis of the ATL-MR framework [7] that reuses the database capabilities to efficiently compute ATL transformations on top of the MapReduce programming model. ATL-MR has shown positive results in terms of execution time when computing complex model-to-model transformations on top of very large models distributed among a HBase cluster.

	Property						
Key	name	classes	methods	imports	modifier	visibility	returnType
'p1'	'package1'	{'c1', 'c2'}					
'c1'	'class1'						
'c2'	'class2'		{'m1'}	{'c1'}			
'm1'	'method1'				'mod1'		't1'
'mod1'						'private'	
't1'	'void'						

Containment			Type	
Key	container	feature	nsURI	EClass
'p1'			'http://java'	'Package'
'c1'	'p1'	'classes'	'http://java'	'Class'
'c2'	'p1'	'classes'	'http://java'	'Class'
'm1'	'c2'	'methods'	'http://java'	'Method'
'mod1'	'm1'	'modifier'	'http://java'	'Modifier'
't1'	'm1'	'returnType'	'http://java'	'Type'

Table 3.4 – Examples instance stored as a sparse table in NeoEMF/Column

3.4 Tooling

NeoEMF is developed as a set of open source Eclipse plugins distributed under the EPL license[3]. The NeoEMF website[4] presents an overview of the key features and current ongoing work. The source code repository and wiki are available on GitHub[5]. NeoEMF has been used as the persistence solution of the MONDO European project [69] and is used to store large models automatically extracted from reverse engineering processes. Details on dependencies and library versions are provided in Table 3.5.

Up-to-date benchmark results are available on the project wiki, and can be computed from the latest version of the tool by running the benchmarks available on the project repository. In addition, a convenience docker image is provided[6] to quickly install NeoEMF in a dedicated environment and run the benchmarks locally.

NeoEMF wiki provides a set of examples and resources for beginners and advanced users: a tutorial showing how to install and get started with NeoEMF (also available in Appendix A of this manuscript), a ready to use demonstration, code examples, database configuration snippets, and specific backend configurations. An additional demonstration video is available online[7].

---

[3] `https://www.eclipse.org/legal/epl-v10.html`

[4] `http://www.neoemf.com`

[5] `http://www.github.com/atlanmod/NeoEMF`

[6] `https://hub.docker.com/r/atlanmod/neoemf/`

[7] `http://hdl.handle.net/20.500.12004/1/U/293557`

Software metadata	**Description**
Current Software Version	1.0.2
Permanent link to executables of this version	Eclipse Update Site: https://atlanmod.github.io/NeoEMF/releases/1.0.2/plugin/ Maven Repository https://mvnrepository.com/search?q=neoemf
Legal Software License	EPL (NeoEMF) GPL (Neo4j convenience bundle)
Computing Platform / Operating System	Java 8-compatible platform Eclipse users: Eclipse Luna or later
Installation requirements & dependencies	Java 8
Software code languages, tools, and services used	Java, Eclipse, Neo4j 1.9.6, MapDB 3.0.2, HBase 1.2.4
If available, link to user manual - if formally published include a reference to the publication in the reference list	Website: www.neoemf.com Tutorial: https://github.com/atlanmod/NeoEMF/wiki/Get-Started
Support email for questions	neoemf@googlegroups.com

Table 3.5 – Software metadata

3.5 Empirical Evaluation

In this section we evaluate the performance of our proposal by comparing it against the *de-facto* standard model persistence solutions. Based on our experience with industrial partners, we have reverse-engineered three models (*set1* to *set3*) of increasing sizes from open source Java projects whose sizes resemble those one can find in real world scenarios (see Table 3.6). We compare the different solutions by using a set of model queries extracted from real reverse-engineering use cases, and compare the results of the different solutions in terms of execution time and memory consumption.

3.5.1 Benchmark Presentations

We consider four persistence solutions in our benchmarks: NEOEMF/GRAPH , NEOEMF/MAP , CDO, and the default XMI serialization mechanism of EMF. The executed queries access the model using the standard EMF API, making them agnostic of which backend they are running on. Other persistence solutions have been discarded of this comparison because they do not strictly comply with the standard EMF behavior (e. g.MongoEMF), they require manual modifications in the source models and metamodels (e. g.EMF Fragments), or because we were only able to execute a small subset of the experiments on them (e. g.Morsa). Note that persistence solutions that does not aim to handle very large models are also discarded (e. g.NEOEMF/COLUMN and EMF Store).

The executed model queries retrieve, and are ordered by their computation complexity (i. e.number of model element to traverse and properties to access):

— **ClassAttributes** computes the attributes of all the *Class* instances in the model
— **SingletonMethods** find static methods returning their containing *Class* (singleton pattern)
— **InvisibleMethods** find all the methods that have a private or protected modifier
— **UnusedMethods** computes the set of methods that are private and not internally called

Queries are executed using two memory configurations: the first one uses a large Java Virtual Machine (JVM) of 8 GB that is used to evaluate the performance of our approach when there is enough memory to compute the query and garbage collection is negligible. The second configuration uses a small JVM of 512 MB that is used to evaluate how the benchmarked solutions behave in a highly constrained memory environment. Note that this benchmark focuses on evaluating query execution time, and does not take into account model loading and unloading.

Experiments are executed on a computer running Fedora 20 64 bits. Relevant hardware elements are: an Intel Core I7 processor (2.7 GHz), 16 GB of DDR3 SDRAM (1600 MHz) and a SSD hard-disk. Experiments are executed on Eclipse 4.5.2 (Mars) running Java SE Runtime Environment 1.8. Note that if not specified, this setup is used by default to evaluate our different contributions in the remaining of this manuscript.

Model	**# Elements**	**XMI Size** (MB)
set1	6756	1.7
set2	80 665	20.2
set3	1 557 007	420.6

Table 3.6 – Benchmarked Models

3.5.2 Results

Table 3.7 to 3.10 present the results of executing the presented queries over the benchmarked persistence frameworks. Each table presents the result of a specific query, and each cell contain both the execution time in the large and the small JVM. Note that execution time is measured in milliseconds, and the correctness of the results have been checked by comparing the results obtained by running the queries on the different implementations with the ones obtained from the standard XMI-based implementation using EMF Compare [18].

Model	**XMI**		**CDO**		**NeoEMF/Graph**		**NeoEMF/Map**	
set1	2	2	2945	2998	2024	2267	1124	1226
set2	13	14	11 125	12 256	9630	10 727	4660	4880
set3	304	*OOM*[1]	148 218	619 008	95 314	638 041	57 011	58 827

[1] OutOfMemory Error

Table 3.7 – ClassAttributes Results in milliseconds (Large VM / Small VM)

Model	**XMI**		**CDO**		**NeoEMF/Graph**		**NeoEMF/Map**	
set1	4	4	3481	3514	1955	1784	1130	1074
set2	27	28	12 759	12 548	7421	10 657	4576	4393
set3	312	*OOM*	159 170	553 141	87 680	635 254	57 288	57 010

Table 3.8 – SingletonMethods Results in milliseconds (Large VM / Small VM)

3.5.3 Discussion

The analysis of the results show that both NeoEMF/Graph and NeoEMF/Map are interesting candidates to store and access large models in constrained memory environments. Both NeoEMF implementations perform better than CDO in the evaluated scenarios, and are able to handle *set3* in a constrained memory environment while XMI-based implementation crashes with an *OutOfMemory* error. However when the model to query fits in memory, the XMI serialization outperforms all the existing solutions in terms of execution time. This result is expected because XMI initially loads the full model, allowing to compute the entire query in memory while *lazy-loading* approaches bring into memory elements when they are needed, and usually have to perform more input/output operations to enable element unloading and improve memory consumption.

Model	**XMI**		**CDO**		**NEOEMF/GRAPH**		**NEOEMF/MAP**	
set1	4	4	3143	3248	1878	1789	1192	1176
set2	18	17	12 496	11 290	8505	9940	4856	5056
set3	551	*OOM*	176 928	549 892	97 104	693 537	78 574	78 584

Table 3.9 – InvisibleMethods Results in milliseconds (Large VM / Small VM)

Model	**XMI**		**CDO**		**NEOEMF/GRAPH**		**NEOEMF/MAP**	
set1	7	7	3212	2924	1942	2346	1425	1437
set2	46	42	12 255	12 169	10 274	11 652	7283	7177
set3	654	*OOM*	171 558	1 160 980	97 782	1 368 399	114 539	118 498

Table 3.10 – UnusedMethods Results in milliseconds (Large VM / Small VM)

In the presented results NEOEMF/MAP outperforms other scalable persistence frameworks in terms of exectution time. In addition, the constrained memory environment does not have a significant impact on the connector's performance, enabling very large model querying. This can be explained by the model to data-store mapping used in NEOEMF/MAP that is optimized to access a single feature from a modeling element. Technically, the framework does not require any complex in-memory structure to represent the model, and only keeps in memory one key-value pair representing the element currently processed. This architecture allows to remove from memory elements as soon as they have been processed, thus reducing the memory consumption.

NEOEMF/GRAPH also outperforms CDO when a large virtual machine is allocated to the computation, but is less interesting in constrained memory environment. This can be explained by the underlying model to graph mapping, which allows efficient model navigations, while CDO's relational schema requires multiple table join operations to compute a complex navigation. However, the nature of the EMF API that performs low-level and fragmented queries implies a lot of database lookups to find a node corresponding to a given element, which is typically costly in terms of memory in graph databases, limiting NEOEMF/GRAPH benefits in highly constrained memory environment.

As a summary, NEOEMF/MAP is a good solution to query very large models in a constrained memory environment. The underlying model to data-store mapping has been designed to handle the typical query patterns generated by the high-level EMF API, improving both execution time and memory consumption.

Note that our experimentations does not evaluate the cost in terms of execution time of additional framework's features that are not directly related to model persistence. For example, some of the CDO core features such as transaction support, collaborative editing, and versioning cannot be disabled and potentially have an impact on the results. To provide a fair comparison between the presented solution we should either evaluate this feature-specific overhead or provide the same capabilities for other implementations.

3.6 Conclusion

In this chapter we introduced NEOEMF , a multi-datastore model persistence framework. It relies on a *lazy-loading* capability that loads model element individually, allowing very large model navigation in a reduced amount of memory, by loading elements when they are accessed. NEOEMF provides three implementations that can be plugged transparently to provide an optimized solution to different modeling use cases: atomic accesses through interactive editing, complex query computation, and cloud-based model transformation.

Our solution fulfill all the requirements presented in Section 3.1. *Interoperability* requirements are addressed by (i) the EMF API implementation that allows to plug NEOEMF into existing EMF-based applications, (ii) the modular architecture presented in Section 3.2 that allows to switch from one data-store to another according to the modeling scenario, and (iii) an extensible architecture that eases new data-store connector integration. *Performance* requirements are addressed by NEOEMF 's *lazy-loading* mechanism that allows to manipulate large models, and our experiments have shown that NEOEMF/MAP is an interesting persistence solution that outperforms state of the art frameworks in highly constrained memory environments.

In the rest of this manuscript we show how NEOEMF can be complemented by caching and prefetching strategies to further improve performances, and we investigate the use of efficient graph query languages to efficiently compute model queries and transformations.

4

Model Prefetching and Caching

Prefetching and caching are two well-known techniques used to improve performance of applications that rely intensively on I/O accesses. Prefetching consists in bringing objects into memory before they are actually requested by the application to reduce performance issues due to the latency of I/O accesses. Fetched objects are then stored in memory to speed-up their access later on. In contrast, caching aims at speeding up the access by keeping in memory objects that have been already loaded.

Prefetching and caching have been part of database management systems and file systems for a long time and have proven their efficiency in several use cases [99, 90]. In particular, P. Cao et al. [26] showed that integrating prefetching and caching strategies together dramatically improves the performance of I/O-intensive applications. In short, prefetching mechanisms work by adding —dynamic or static— load instructions (according to prefetching rules derived by static [62] or execution trace analysis [33]) into an existing program. Caches are usually controlled by global policies such as Least Recently Used (LRU) or Most Recently Used (MRU) that define in which order elements are discarded (i. e. removed from memory) and replaced by new ones.

Currently, there is lack of support for prefetching and caching at the model level. As we stated in Chapter 3, model-driven engineering (MDE) is progressively adopted in the industry [54, 76], and such support is required to raise the scalability of MDE tools dealing with large models where storing, editing, transforming, and querying operations are major issues [68, 115].

Existing approaches, including the NEOEMF framework presented in the previous chapter, have proposed scalable model persistence frameworks on top of relational and NoSQL databases [43, 88, 63, 35]. The lazy-loading strategy usually provided by these approaches helps dealing with large models that would otherwise not fit in memory. However, it also adds an execution time overhead due to the latency of I/O accesses to load model excerpts from the database, specially when executed in a distributed environment, where this I/O delay is amplified by the network latency.

In this sense, we propose a new prefetching and caching framework for models. We present PREFETCHML , a domain specific language and execution engine, to specify prefetching and caching policies and execute them at run-time in order to optimize model access operations. This DSL allows designers to customize the prefetching rules to the specific needs of model manipulation scenarios, even providing several execution plans for different use cases. PREFETCHML also includes a monitoring component that provides insights on the execution performance to guide modelers on improving their prefetching plans. Finally our framework also embeds a set of caching strategies and consistency policies allowing modelers to finely adapt the cache behavior to a particular execution scenario. Our framework is built on top of the EMF infrastructure and therefore it is compatible with existing scalable model persistence approaches, regardless whether those backends also offer some kind of internal prefetching mechanism. A special version tailored to the NeoEMF/Graph [6] engine is also provided for further performance improvements. The empirical evaluation of PREFETCHML highlights the significant time benefits it achieves.

The remaining of this chapter is organized as follows: Section 4.1 introduces further the background of prefetching and caching techniques in the modeling ecosystem while Section 4.2 introduces the PREFETCHML DSL. Section 4.3 describes the framework infrastructure, its basic rule execution algorithm, and the consistency policies we have implemented. Section 4.4 presents our monitoring component and how it can be used to optimize an existing PREFETCHML plan. Section 4.5 introduces the editor that allows the designer to define prefetching and caching rules, and the implementation of our tool and its integration with the modeling environment. Finally, Section 4.6 presents the benchmarks used to evaluate our prefetching tool and associated results. Section 4.7 summarizes the key points of the chapter and draws conclusions.

4.1 State of the Art

Prefetching and caching techniques are common in relational and object databases [99] in order to improve query computation time. Their presence in NoSQL databases is much more limited, which contrasts with the increasing popularity of this type of databases as model storage solution. Moreover, database-level prefetching and caching strategies do not provide fine-grained configuration of the elements to load according to a given usage scenario —such as model-to-model transformation, interactive editing, or model validation— and are often strongly connected to the data representation, making them hard to evolve and reuse.

4.1.1 Prefetching and Caching in Current Modeling Frameworks

Most of the existing scalable model persistence frameworks are built on top of relational or NoSQL databases to store and access large models [35, 43]. These approaches are often based on lazy-loading strategies to optimize memory consumption by loading only the accessed objects from the database. While lazy-loading approaches have proven their efficiency in terms of memory consumption to load and query very large models [34, 88], they generate a lot of fragmented queries on the database, thus adding a significant execution time overhead. For the reasons described above, these frameworks cannot benefit from database prefetching solutions nor they implement their own mechanism, with the partial exception of CDO [43] that provides some basic prefetching and caching capabilities [1]. For instance, CDO is able to bring into memory all the elements of a list at the same time, or load nested/related elements up to a given depth. Nevertheless, alternative prefetching rules cannot be defined to adapt model access to different contexts nor it is possible to define rules with complex prefetching conditions.

Caching is a common solution used in current scalable persistence frameworks to improve query execution involving repeated accesses of model elements. They are integrated in several solutions such as CDO [43], Morsa [88], and NeoEMF [35]. However, these caches are tailored to a specific solution, and they typically lack of advanced configurations such as the replacement policy to use, the maximum size of the cache, or the number of elements to drop when the cache is full. In addition, persistence framework caches are usually defined as internal components, and do not allow client applications to access the content of the cache.

Hartmann et al. [49] propose a solution to tackle scalability issues in the context of models@run.time by splitting models into chunks that are distributed across multiple nodes in a cluster. A lazy-loading mechanism allows to virtually access the entire model from each node. However, to the best of our knowledge the proposed solution does not provide a prefetching mechanism, which could improve the performance when remote chunks are retrieved and fetched among nodes.

Optimization of query execution has also been targeted by other approaches not relying on prefetching but using a variety of other techniques. EMF-IncQuery [11] is an incremental evaluation engine that computes graph patterns over an EMF model. It relies on an adaptation of the RETE algorithm, and results of the queries are cached and incrementally updated when the model is modified using the EMF notification framework. While EMF-IncQuery can be seen as an efficient EMF cache, it does not aim to provide prefetching support, and cache management cannot be tuned by the designer. The Hawk [3] model indexer also aims at improving model query computation by providing an efficient backend-independent query language built on top of the Epsilon platform. However, Hawk has been primarily designed to handle model queries, and does not provide constructs to define prefetching and caching plans on top of the indexed models.

4.1.2 Problematic and Requirements

In the presented approaches, prefetching and caching components are usually designed (when they exist) to generically improve application's performance by relying on a preset cache policy, that can be complemented, in the case of CDO, with a predefined simple prefetching strategy. However, we believe that these generic, statically defined techniques should be adaptable regarding the expected modeling scenario and query access patterns on the model. Finally, the presented approaches are specific to the chosen persistence solution, and can not be reused from one solution to another.

Thus, we define a set of requirements to address in order to provide an efficient and configurable prefetching and caching component to complement modeling and persistence frameworks:

---

[1] `https://wiki.eclipse.org/CDO/Tweaking_Performance`

Rq1 Ability to define/execute prefetching rules independently of the database backend.

Rq2 Ability to define/execute prefetching rules transparently from the persistence framework layered on top of the database backend.

Rq3 A prefetching language expressive enough to define rules involving conditions at the metamodel and instance model levels (i.e. loading all instances of a class `A` that are linked to a specific object of a class `B`).

Rq4 A context-dependent prefetching language allowing the definition of alternative prefetching and caching plans for specific modeling scenarios.

Rq5 A readable prefetching DSL enabling designers to easily create, tune, and maintain prefetching and caching rules.

Rq6 A monitoring/quality component providing feedbacks on the prefetching and caching plan execution to guide modelers on improving their prefetching rules.

In the following sections, we present PREFETCHML , our prefetching and caching framework that tackles these challenges.

4.2 The PrefetchML DSL

The PREFETCHML DSL is a high-level, event-based language that describes prefetching and caching rules over models. Rules are triggered when an event satisfying a particular condition is received. These events can be the *initial model loading*, an *access* to a specific model element, the *update* of a value, or the *deletion* of an element. Events can be parameterized with OCL guards that express conditions are validated to trigger prefetching and caching instructions.

Loading instructions are also defined in OCL. The set of elements to be loaded as a response to an event is characterized by means of OCL expressions that navigate the model and select the elements to fetch and store in the cache. Not only loading requests can be defined, the language also provides an additional construct to control the cache content by removing specific elements from the cache when a particular event is received. We choose OCL as our model navigation language because it is a well-known OMG standard intensively used in the MDE community that expresses sophisticated model constraints, invariants, and queries independently of the low-level model persistence solution.

Prefetching and caching rules are organized in *plans*, that are sets of rules that should be used together to optimize a specific usage scenario for the model, since different kinds of model accesses may require different prefetching strategies. For example, a good strategy for an interactive model browsing scenario is to fetch and cache the containment structure of the model, whereas for a complex query execution scenario it is better to have a plan that fits the specific navigation path of the query.

Beyond a set of prefetching rules, each plan defines a cache that can be parametrized, and a consistency policy that defines the strategy to use to manage the life-cycle of cached elements when the model is updated.

In what follows, we formalize the abstract and concrete syntax of the PREFETCHML DSL and introduce them by extending the running example presented in section 2.2.1. Next Section will introduce how these rules are executed as part of the PREFETCHML engine.

4.2.1 Abstract Syntax

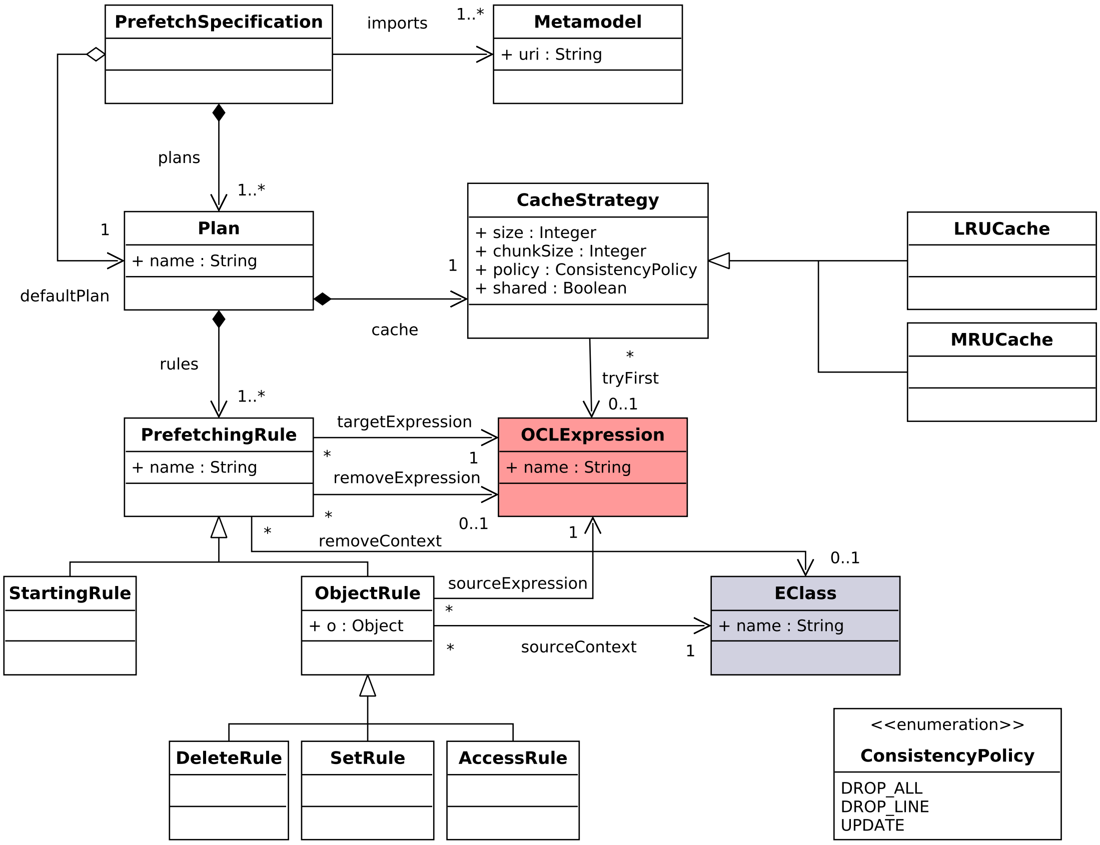

PrefetchSpecification
imports
1..*
Metamodel
+ uri : String
plans
1..*
1
Plan
+ name : String
defaultPlan
CacheStrategy
+ size : Integer
+ chunkSize : Integer
+ policy : ConsistencyPolicy
+ shared : Boolean
1
cache
LRUCache
MRUCache
rules
1..*
*
tryFirst
0..1
PrefetchingRule
+ name : String
targetExpression
*
1
removeExpression
*
0..1
OCLExpression
+ name : String
*
removeContext
1
0..1
StartingRule
ObjectRule
+ o : Object
sourceExpression
*
*
sourceContext
1
EClass
+ name : String
DeleteRule
SetRule
AccessRule
<<enumeration>>
ConsistencyPolicy
DROP_ALL
DROP_LINE
UPDATE

Figure 4.1 – Prefetch Abstract Syntax Metamodel

This section describes the main concepts of PREFETCHML focusing on the different types of rules it offers and how they can be combined to create a complete prefetching and caching specification.

Figure 4.1 depicts the metamodel corresponding to the abstract syntax of the PrefetchML language. A *PrefetchSpecification* is a top-level container that *imports* several *Metamodels*. These metamodels represent the domain on which prefetching and caching rules are described, and are defined by their *Unified Resource Identifier (URI)*.

The imported *Metamodel* concepts (classes, references, attributes) are used in prefetching *Plans*, which are named entities that group rules that are applied in a given execution context. A *Plan* can be the *default* plan to execute in a *PrefetchSpecification* if no execution information is provided.

Each *Plan* contains a *CacheStrategy*, which represents the information about the cache policy the prefetcher applies to keep loaded objects into memory. Currently, available cache strategies are *LRUCache* (Least Recently Used) and *MRUCache* (Most Recently Used). These *Caches* define four parameters: (i) the maximum number of objects they can store (*size*), (ii) the number of elements to free when the cache is full (*chunkSize*), (iii) the *consistency policy* used to manage model modifications, and (iv) the integration of the cache with the running application (details on cache consistency/integration are provided in Section 4.3). In addition, a *CacheStrategy* can contain a *tryFirst* *OCL* *expression* [2]. This expression is used to customize the default cache replacement strategy with additional knowledge: it returns a set of model elements that should be removed from the cache if it is full, overriding the selected caching policy.

---

[2] OCLExpression is defined in the Eclipse MDT OCL metamodel

Plans also contain the core components of the PrefetchML language: *PrefetchingRules* that describe tracked model events and the loading and caching instructions. We distinguish two kinds of *PrefetchingRules*:

- *StartingRules* that are prefetching instructions triggered a single time when the prefetching plan is loaded
- *ObjectRules* that are triggered when an element satisfying a given condition is accessed, deleted, or updated

ObjectRules can be categorized in three different types: *Access* rules, that are triggered when a particular model element is accessed, *Set* rules that correspond to the setting of an attribute or a reference, and *Delete* rules, that are triggered when an element is deleted or simply removed from its parent. When to fire the trigger is also controlled by the *sourceContext* class (from the *imported* metamodels), that represents the type of the elements that could trigger the rule. This is combined with the *sourceExpression* (i.e. the guard for the event) to decide whether an object matches the rule.

All kinds of *PrefetchingRules* contain a *targetExpression*, that represents the elements to load when the rule is triggered. This expression is an *OCLExpression* that navigates the model and returns the elements to load and cache. Note that if *self* is used as the *targetExpression* of an *AccessRule* the framework will behave as a standard cache, keeping in memory the accessed element without fetching any additional object.

It is also possible to define *removeExpressions* in *PrefetchingRules*, that are executed after *targetExpressions* to finely control the cache contents. When a *removeExpression* is evaluated, the prefetcher marks as free all the elements it returns from the cache. Each *removeExpression* is associated to a *removeContext Class*, that represents the context of the OCL expression. A *remove* expression can be coupled with the *tryFirst* expression contained in the *CacheStrategy* to finely tune the default replacement policy of the cache.

4.2.2 Concrete Syntax

We introduce now the concrete syntax of the PrefetchML language, which is derived from the abstract syntax metamodel presented in Figure 4.1. Listing 2 presents the grammar of the PrefetchML language expressed using XText [44], an EBNF-based language used to specify the grammar and generate an associated toolkit containing a metamodel of the language, a parser, and a basic editor. The grammar defines the keywords associated to the constructs presented in the PrefetchML metamodel. Note that *OCLExpressions* are parsed as strings, the model representation of the queries presented in Figure 4.1 is computed by parsing them using the Eclipse MDT OCL toolkit[3].

Listing 2 – PrefetchML Language Grammar

```
grammar fr.inria.atlanmod.Prefetching
with org.eclipse.xtext.common.Terminals
import "http://www.inria.fr/atlanmod/Prefetching"

PrefetchSpecification:
  metamodel=Metamodel
```

---

[3] http://www.eclipse.org/modeling/mdt/?project=ocl

```
  plans+=Plan+
;

Metamodel:
  'import' nsURI=STRING
;

Plan:
  'plan' name=ID (default?='default')? '{'
    cache=CacheStrategy
    rules+=(StartingRule | AccessRule)*
  '}'
;

CacheStrategy:
  (LRUCache{LRUCache} | MRUCache{MRUCache})
    (properties=CacheProperties)? ('when full remove' tryFirstExp=OCLExpression)?
;

LRUCache:
  'use cache' 'LRU'
;

MRUCache:
  'use cache' 'MRU'
;

CacheProperties:
  '[' 'size=' size=INT ('chunk=' chunk=INT)? shared='shared'? 'policy=' policy=ConsistencyPolicy ']'
;

enum ConsistencyPolicy:
  DROP_ALL = 'drop_all' |
  DROP_LINE = 'drop_line' |
  UPDATE = 'update'
;

PrefetchingRule:
  (StartingRule | AccessRule | DeleteRule | SetRule)
;

StartingRule:
  'rule' name=ID ':' 'on starting'
  'fetch' targetPatternExp=OCLExpression
  ('remove' 'type'removeType=ClassifierExpression removePatternExp=OCLExpression)?
;

AccessRule:
  'rule' name=ID ':' 'on access'
  'type'sourceType=ClassifierExpression (sourcePatternExp=OCLExpression)?
  'fetch' targetPatternExp=OCLExpression
  ('remove' 'type'removeType=ClassifierExpression removePatternExp=OCLExpression)?
;

DeleteRule:
  'rule' name=ID ':' 'on delete'
  'type'sourceType=ClassifierExpression (sourcePatternExp=OCLExpression)?
  'fetch' targetPatternExp=OCLExpression
  ('remove' 'type'removeType=ClassifierExpression removePatternExp=OCLExpression)?
;

SetRule:
  'rule' name=ID ':' 'on set'
  'type'sourceType=ClassifierExpression (sourcePatternExp=OCLExpression)?
  'fetch' targetPatternExp=OCLExpression
  ('remove' 'type'removeType=ClassifierExpression removePatternExp=OCLExpression)?
;

OCLExpression: STRING ;

ClassifierExpression: ID;
```

4.2.3 Running Example

In order to better illustrate the features of PrefetchML , we reuse the running example presented in Section 2.2.1. Listing 3 presents three sample OCL queries that can be computed over an instance of our example metamodel (Figure 6.2): the first one returns the *Package* elements that do not contain any *Class* through their *classes* reference. The second one returns from a given *Class* all its contained *Methods* that have a private *Modifier*, and the third one returns from a *Class* a sequence containing the *returnTypes* of all its *imported Methods* that contain *’Mock’* in their name.

```
context Package
def : isAnEmptyPackage : Boolean =
self.classes→isEmpty()

context Class
def : privateMethods : Sequence(Class) =
self.methods
→select(mm | mm.modifier = VisibilityKind::Private)

context Class
def : importedReturnTypes : Sequence(Type) =
self.imports.methods.returnType
→select(t | t.name.contains(’Mock’)))
```

Listing 3 – Sample OCL Query

Listing 4 provides an example of a *PrefetchSpecification* written in PrefetchML . To continue with our running example, the listing displays prefetching and caching rules suitable for a scenario where all the queries expressed in Listing 3 are executed in the order they are defined.

The *PrefetchSpecification* imports the Java *Metamodel* (line 1). This *PrefetchSpecification* contains a *Plan* named *samplePlan* that uses a *LRUCache* that can contain up to 100 elements and removes them by chunks of 10 (line 4). The cache also defines the *shared* property, meaning that elements computed by the prefetching rules and the running application will be cached together. Finally, the cache uses the *drop_line* consistency policy, that removes lines from the cache corresponding to updated elements. Note that the consistency policy is not important in this example, because OCL expressions are side-effect free and do not generate update notifications.

The *plan* also defines three *PrefetchingRules*: the first one, `r1` (5-6), is a starting rule that is executed when the plan is activated, and loads and caches all the *Package* classes. The rule `r2` (7-8) is an access rule that corresponds to the prefetching and caching actions associated to the query *PrivateMethods*. It is triggered when a *Class* is accessed, and loads and caches all the *Methods* and *Modifiers* it contains. The rule `r3` (9-11) corresponds to the query *ImportedReturnTypes*: it is also triggered when a *Class* is accessed, and loads the type name of each *Method* of its imported *Classes*. The rule also defines a `remove` expression, that removes all the *Package* elements from the cache when the loading instruction is completed.

```
import "http://www.example.org/Java"

plan samplePlan {
  use cache LRU[size=100,chunk=10, shared, policy=drop_line]
  rule r1 : on starting fetch
    Package.allInstances()
  rule r2 : on access type Class fetch
    self.methods.modifier
  rule r3 : on access type Class fetch
    self.imports.methods.returnType.name
    remove type Package
}
```

Listing 4 – Sample Prefetching Plan

4.3 PrefetchML Framework Infrastructure

Prefetching and caching plans defined from the grammar presented in the previous section constitute the input of an execution engine that parses, evaluates, and triggers PREFETCHML rules over models. In this section we present the infrastructure of the PREFETCHML engine and its integration in the modeling ecosystem (integration details into specific modeling frameworks are provided in Section 4.5). We also detail how prefetching rules are handled and executed using the running example presented in the previous section, and we present the different cache consistency policies and integration levels that can be defined to tune the prefetching and caching behavior.

4.3.1 Architecture

Figure 4.2 shows the integration of the PREFETCHML framework in a typical modeling framework infrastructure: orange nodes represent standard model access components: a **User** *uses* a model-based tool that accesses a model through a modeling API, which delegates to a persistence framework in charge of handling the physical storage of the model (for example in XML files, or in a database). The elements in this modeling stack are typically set-up by a **Modeler** who configures them according to the application's workload (for example by selecting a scalable persistence framework if the application aims to handle large models).

The PREFETCHML framework (red nodes) receives events from the modeling framework. When an event triggers a prefetching rule, the framework delegates the actual computation to its **Model Connector**. This component interacts with the modeling framework to retrieve the objects to load and cache, typically by translating the OCL expressions in the PREFETCHML rules into lower level calls to the framework API[4]. The PREFETCHML framework also provides *monitoring information* that gives useful insights on the execution to help the **Modeler** to customize the persistence framework and the prefetching and caching plans. The monitoring component and the feedbacks it provides are detailed in the next section.

The framework also intercepts model element *accesses*, in order to search first in its **Cache** component if the requested objects are already available. If the cache already contains the requested information (i. e. if it has been prefetched before), it is returned to the modeling framework, bypassing the persistence layer and improving execution time. Model modification events are also intercepted by the framework to update/invalidate cached values in order to keep the cache content consistent with the model state.

---

[4] Section 4.5 discusses two specific implementations of this component

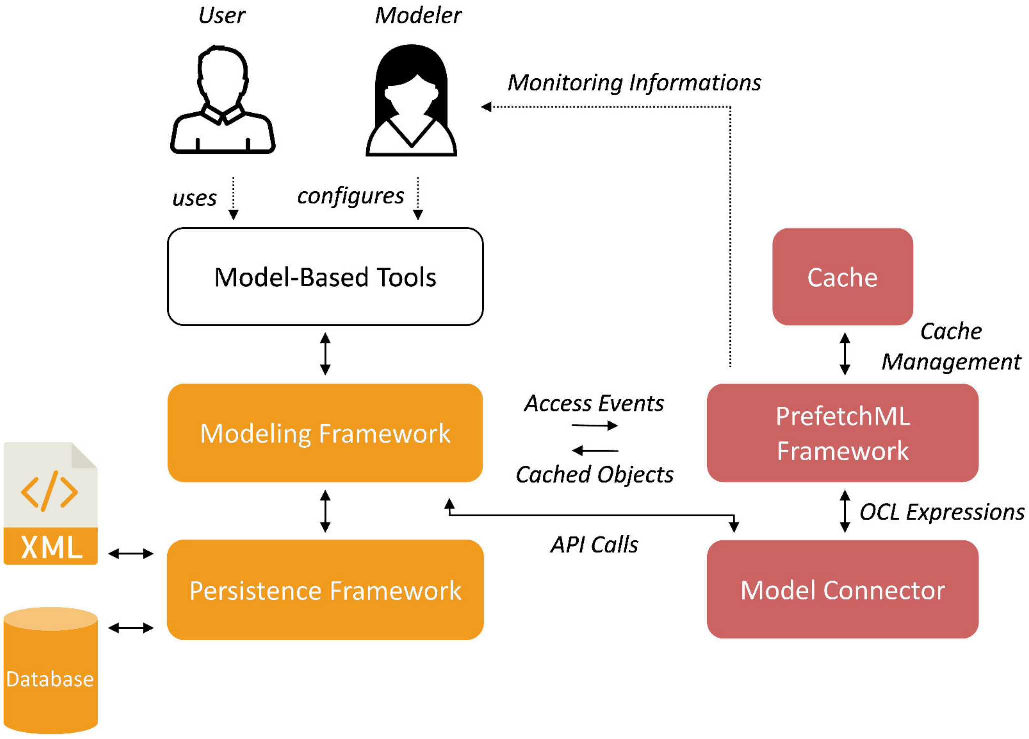

User
Modeler
Monitoring Informations
uses
configures
Model-Based Tools
Cache
Cache Management
Modeling Framework
Access Events
Cached Objects
PrefetchML Framework
XML
OCL Expressions
API Calls
Persistence Framework
Model Connector
Database

Figure 4.2 – PREFETCHML Integration in MDE Ecosystem

Figure 4.3 describes the internal structure of the PREFETCHML Framework.

As explained in Section 4.2, a **PrefetchMLSpecification** *conforms to* the PREFETCHML metamodel. This specification *imports* also the metamodel/s for which we are building the prefetching plans.

The **Core** component of the PrefetchML framework is in charge of loading, parsing and storing these *PrefetchMLSpecifications* and use them to find and retrieve the prefetching / caching rules associated with an incoming event, and, when necessary, execute them. This component also contains the internal *cache* that retains fetched model elements in memory. The core component ensures cache consistency, by invalidating part or all cached records when update, create, or delete events are received. The **Rule Store** is a data structure that stores all the object rules (access, update, and delete) contained in the input *PrefetchML description* and allows to easily retrieve rules that can be applied for a given object and event.

The **Model Connector** component is in charge of the translation and the execution of *OCLExpressions* in the prefetching rules. This connector can work at the modeling framework level, meaning that it executes fetch queries using the modeling API itself, or at the database level, translating directly OCL expressions into database queries.

The **CacheAPI** component gives access to the cache contents to client applications.

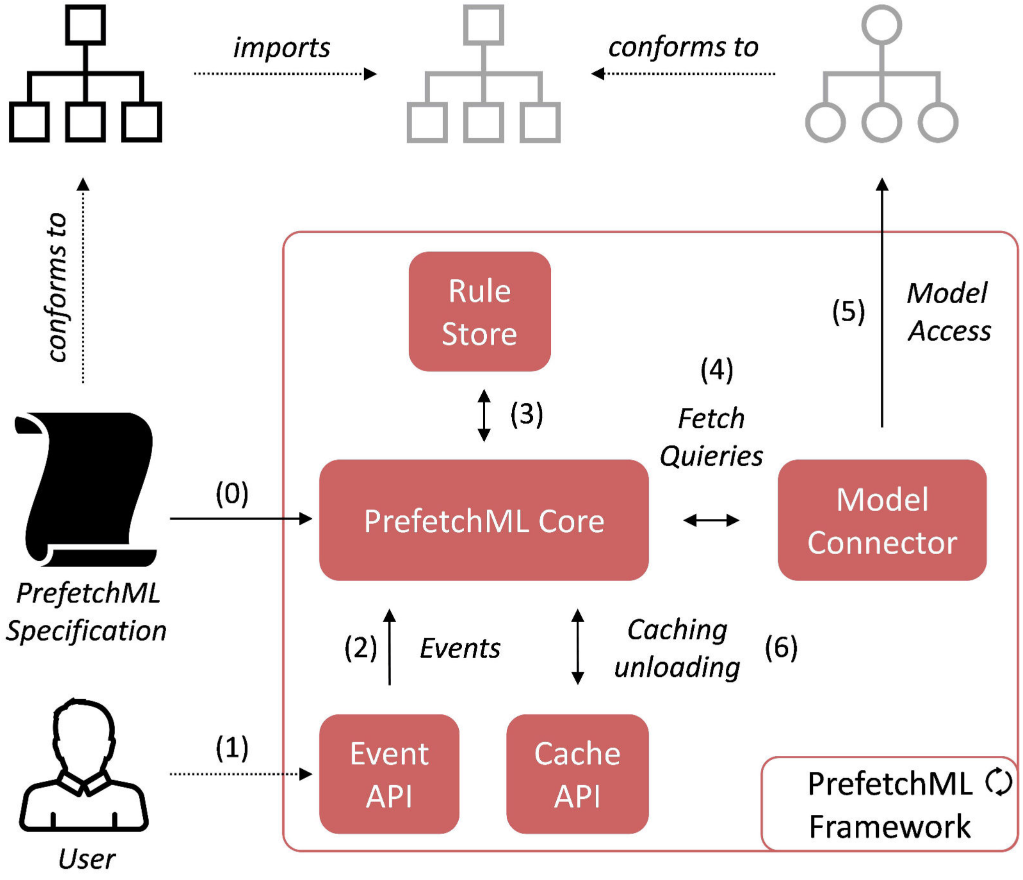

imports
conforms to
conforms to
Rule Store
(3)
(4)
Fetch Quieries
(5)
Model Access
(0)
PrefetchML Specification
PrefetchML Core
Model Connector
(2)
Events
Caching unloading
(6)
(1)
User
Event API
Cache API
PrefetchML Framework

Figure 4.3 – PREFETCHML Framework Infrastructure

It allows manual caching and unloading operations, and provides configuration facilities. This API is an abstraction layer that unifies access to the different cache types that can be instantiated by the Core component. By default, the core component manages its own cache where only prefetched elements are stored, providing a fine-grain control of the cache content. While this may result in keeping in the cache objects that are not going to be recurrently used, using a LRU cache strategy allows the framework to get rid off them when memory is needed. In addition, the grammar allows to define a minimal cache that would act only as a storage mechanism for the immediate prefetched objects.

The **EventAPI** is the component that receives events from the client application. It provides an API to send access, delete, and update events to the core component. These events are defined at the object level, and contain contextual information of their encapsulated model element, such as its identifier, the reference or attribute that is accessed, and the index of the accessed element. This information is then used by the **Core Component** to find and execute the rules that match the event.

In particular, when an object event is sent to the PrefetchML framework (1), the *Event API* handles it and forwards it to the *Core Component*, which is in charge of triggering the associated prefetching and caching rule. To do that, the *Core Component* searches in the *Rule Store* the rules that correspond to the event and the object that triggered it (3). Each *OCLExpression* in the retrieved rules is translated into fetch queries sent to the

Model Connector (4), which is in charge of the actual query computation over the model (5). Query results are handled back by the PREFETCHML *Core*, which caches them and frees the cache from previously stored objects if necessary (6).

As prefetching operations can be expensive to compute, the PREFETCHML Framework runs in the background, and contains a pool of working threads that perform the fetch operations in parallel with the application execution. Model elements are cached asynchronously and are available to the client application through the *CacheAPI*.

Note that this infrastructure is not tailored to any particular data representation and can be plugged in any kind of model persistence framework that stores models conforming to the Ecore metamodel and provides an API rich enough to evaluate OCL queries. This includes for example EMF storage implementations such as XMI, but also scalable persistence layers built on top of the EMF, like NeoEMF [35], CDO [43], and Morsa [88]. However, the efficiency of PrefetchML (in particular the prefetcher throughput) can vary from one persistence solution to another because of synchronization feature and the persistence framework/database ability to handle multiple queries at the same time. This differences are highlighted in the experiments we discuss in Section 4.6.

4.3.2 Rule Processing

We now look at the PREFETCHML engine from a dynamic point of view. Figure 4.4 presents the sequence diagram associated with the initialization of the PrefetchML framework. When initializing, the prefetcher starts by loading the *PrefetchSpecification* to execute (1). To do so, it iterates through the set of plans and stores the rules in the *RuleStore* according to their type (2). In the example provided in Listing 4 this process saves in the store the rules `r2` and `r3`, both associated with the *Class* type. Then, the framework creates the cache (3) instance corresponding to the active prefetching plan (or the default one if no active plan is provided). This creates the `LRU` cache of the example, setting its `size` to 100, its `chunkSize` to 10, and the *drop line* consistency policy.

Next, the PREFETCHML framework iterates over the *StartingRules* of the specification and computes their *targetExpression* using the *Model Connector* (4). Via this component, the OCL expressions are evaluated (in the example the target expression is `Package.allInstances()`) and the traversed elements are returned to the *Core* component (5) which creates the associated identifying keys (6) and stores them in the cache (7). Note that starting rules are not stored in the Rule Store, because they are executed only once when the plan is activated, and are no longer needed afterwards.

Once this initial step has been performed, the framework awaits object events. Figure 4.5 shows the sequence diagram presenting how PREFETCHML handles incoming events. When an object event is received (8), it is encapsulated into a working task which contains contextual information of the event (accessed object, navigated feature, and index of the accessed feature) and asynchronously sent to the prefetcher (9) that searches in the *RuleStore* the object rules that have the same type as the event (10). In the example, if a *Class* element is accessed, the prefetcher searches the corresponding rules and returns `r2` and `r3`. As for the initialization diagram, the next calls involve the execution of the target expressions for the matched rules and saving the retrieved objects in the cache for future calls. Finally, the framework evaluates the remove OCL expressions (17) and frees the matching objects from the memory. In the example, this last step removes from the cache all the instances of the *Package* type.

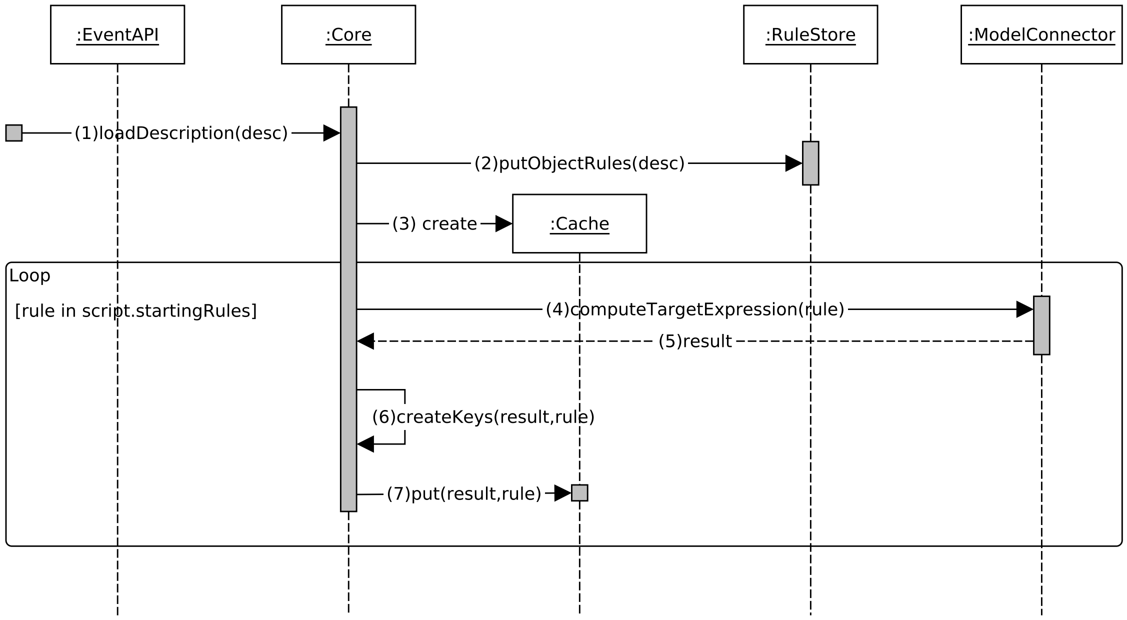

:EventAPI
:Core
:RuleStore
:ModelConnector
(1)loadDescription(desc)
(2)putObjectRules(desc)
(3) create
:Cache
Loop
[rule in script.startingRules]
(4)computeTargetExpression(rule)
(5)result
(6)createKeys(result,rule)
(7)put(result,rule)

Figure 4.4 – PREFETCHML Initialization Sequence Diagram

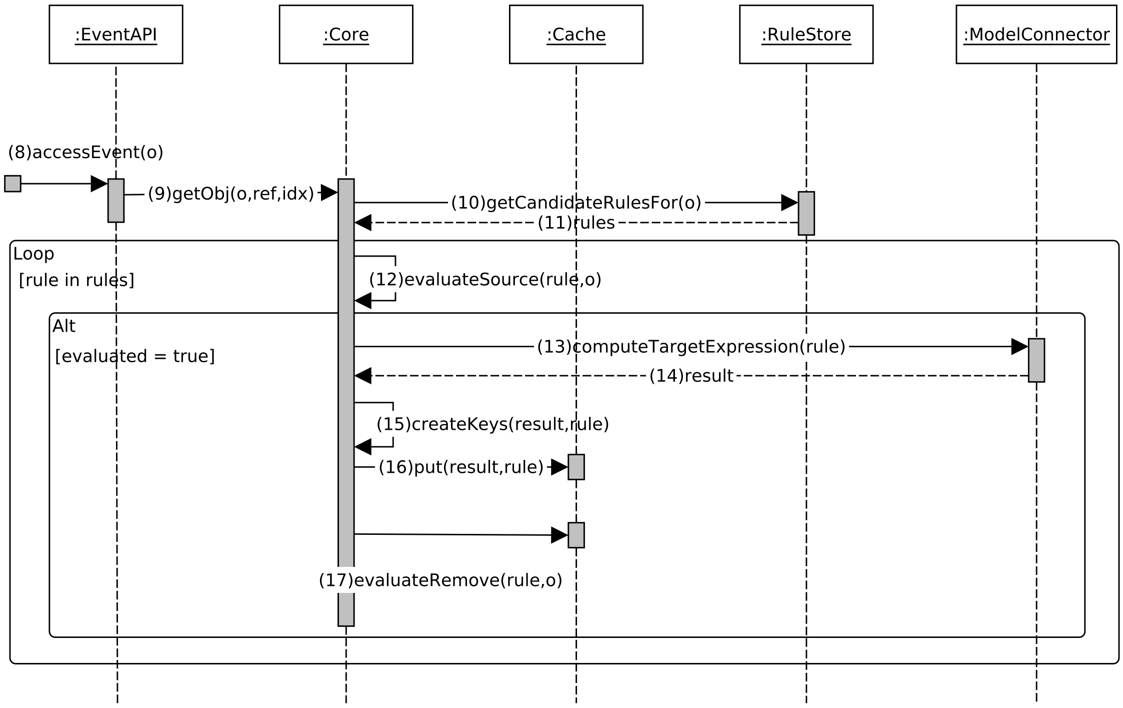

:EventAPI
:Core
:Cache
:RuleStore
:ModelConnector
(8)accessEvent(o)
(9)getObj(o,ref,idx)
(10)getCandidateRulesFor(o)
(11)rules
Loop
[rule in rules]
(12)evaluateSource(rule,o)
Alt
[evaluated = true]
(13)computeTargetExpression(rule)
(14)result
(15)createKeys(result,rule)
(16)put(result,rule)
(17)evaluateRemove(rule,o)

Figure 4.5 – PREFETCHML Access Event Handling Sequence Diagram

4.3.3 Cache Consistency

The PrefetchML DSL presented in Section 4.2 allows to define prefetching rules when an element in the model is *Accessed*, *Set*, and *Deleted*. However, these events are simply used to trigger prefetching rules, and updating the model may create inconsistencies between the PrefetchML cache and the actual model state. While this is not a problem for side-effect free computation such as OCL queries (where no element is modified), it becomes an issue when using PrefetchML on top of model-to-model transformation frameworks, or EMF-API based applications.

To overcome this limitation we have defined a set of cache consistency policies that can be plugged to tune how the engine keeps the cache consistent with the running application. They all ensure that the content of the cache is consistent w.r.t the model, by handling updates with different strategies in order to improve the prefetching throughput or increase cache hits. Available policies retrieve:

— **Drop all**: drop the entire cache every time the model is updated
— **Drop line**: drop the cache lines corresponding to the updated element and all its cached references
— **Update**: update the cache lines corresponding to the updated element with the new value, including referenced elements

Drop all is the simplest cache consistency policy: it drops the entire cache each time a model update event is received. Dropping the entire cache is fast and does not have a significant impact on the prefetcher throughput. However, this policy drops elements that are still consistent with the model, and has an important impact on the prefetcher hit score. Full drop policy is typically used when model modifications are localized at a specific point of the execution, and concern an important part of the model. This consistency strategy can be specified in the cache parameters of a prefetching plan with the *drop-all* keyword.

Drop line removes from the cache the updated element and all the elements referencing it. This approach is adapted to query scenarios where few model modifications are performed at multiple steps of the execution, and dropping the entire cache would have an important impact on the number of hits. However, dropping multiple lines is more expensive in terms of execution time because the framework has to inspect the cache to find all the elements to remove. This policy is used by default if no consistency policy is defined in the executed PrefetchML plan.

Update policy keeps the cache consistent with the model by updating all the lines corresponding to the modified objects. This policy is interesting if a small amount of model modifications are performed, and the updated objects are reused later and should stay in the cache. Updating the cache requires to find the cache lines to update, and navigate the model to find the updated values. This operation is costly (especially because it requires additional model navigations), and may have a significant impact on the prefetcher performances if too many objects are updated during the query execution.

These different cache policies can be selected by the modeler to tune PrefetchML according to its application workload. For example, an interactive model editor can benefit from the *Update* policy, because this kind of application usually has a low workload, with localized model modifications. On the other hand, in the context of a model-to-model transformation that typically creates and updates a lot of model elements, using a lightweight policy such as *drop line* is more appropriated.

Figure 4.6 shows the sequence diagram presenting how PREFETCHML handles model modifications. When an element is updated, an *updateEvent* describing the old (o) and new (n) versions of the updated element is sent to the *EventAPI* (8). This event is forwarded to the *Core* component (9) that retrieves the consistency policy to use (10), and tells the *Cache* to update its content according to it (11). Depending on the used policy, the *Cache* will drop all its content, invalidate the lines corresponding to the updated element, or update its content. The rest of the sequence diagram is similar to the one presented in Figure 4.5, with the particularity that rules are found and computed from the new version of the element instead of the old one.

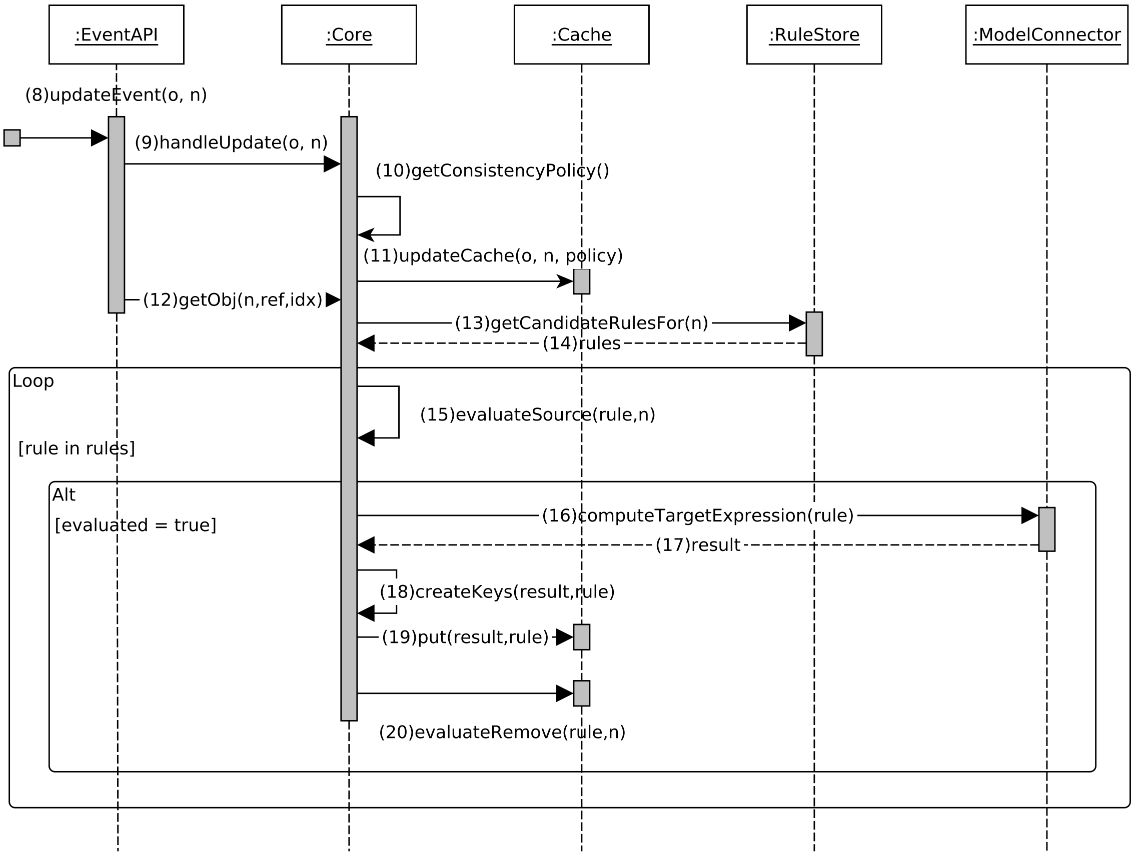

:EventAPI
:Core
:Cache
:RuleStore
:ModelConnector
(8)updateEvent(o, n)
(9)handleUpdate(o, n)
(10)getConsistencyPolicy()
(11)updateCache(o, n, policy)
(12)getObj(n,ref,idx)
(13)getCandidateRulesFor(n)
(14)rules
Loop
[rule in rules]
(15)evaluateSource(rule,n)
Alt
[evaluated = true]
(16)computeTargetExpression(rule)
(17)result
(18)createKeys(result,rule)
(19)put(result,rule)
(20)evaluateRemove(rule,n)

Figure 4.6 – PrefetchML Update Event Handling Sequence Diagram

4.3.4 Global shared cache

PREFETCHML embeds a cache that is dedicated to prefetched elements, providing a lot of control on the cache content to the modeler who knows that every object in the cache has been loaded by a prefetching rule. This approach is interesting when designers want to choose a cache size that perfectly fits their needs, and are not concerned by persistence and application level caches. However, this strict distinction between application and PREFETCHML caches relies on the correctness of the prefetching plan: if the plan is good the cache contains elements that will improve the computation time, if not the application could not benefit from the cached elements.

To overcome this limitation, we have defined a *shared cache* strategy, that contains elements loaded by prefetching rules and by the application itself. It can be enabled by setting the cache parameter *shared* in a PREFETCHML plan. Sharing the cache between the prefetcher and the application provides two benefits: (i) elements that are accessed multiple times are cached even if they are not part of a prefetching rule, improving query execution time, and (ii) the prefetcher throughput is optimized when both prefetching rules and application-level queries are loading the same elements. In this last scenario the PREFETCHML algorithm will be notified that the element has been cached by the application, allowing it to move on the next rule to compute, reducing concurrency issues and improving the prefetching algorithm efficiency. We show in our experiments (Section 4.6) that sharing the cache in the context of OCL query computation has a positive impact on query execution time.

4.4 Plan Monitoring

The presented DSL and execution engine provides several constructs to create and tune PREFETCHML plans according to an expected modeling scenario. However, our experiments when using the framework have shown that an important knowledge of its internal components is required to efficiently tune an existing plan. In this section we introduce the monitoring component we have integrated into the PREFETCHML framework to help modelers tune their prefetching plans by providing feedbacks on the execution. We first introduce the new language constructs and framework updates, then we present an example of the information a modeler can get from the framework and how it can be used to customize an existing PREFETCHML plan. Finally, we show how this same monitoring information can be employed to dynamically adapt the PREFETCHML algorithm and automatically define an appropriate cache integration.

4.4.1 Language Extensions for Plan Monitoring

As we demonstrated in our previous work [36], prefetching and caching can significantly improve model query computation, but this improvement is tightly coupled to the quality of the plan to execute. Intuitively, a *good* prefetching plan is a plan that loads elements before they are needed by the application, and keeps them in memory for a sufficiently long time to make later accesses faster, without polluting cache content with irrelevant objects.

While this intuitive approach is easy to conceptualize, it can be hard to apply in real-life scenarios: the modeler does not know the exact content of the cache, and multiple rules may interact with each other, filling/freeing the cache with different expressions at the same time. Moreover, comparing the quality of two prefetching plans and/or the impact of an update on a specific rule is not a straightforward task, and requires to have a close look at the cache content and a deep knowledge on how PREFETCHML rules are evaluated and executed. To help designers evaluate the quality of their prefetching plans we have defined a monitoring component that presents execution information allowing them to detect problematic and missing rules, guards, and interaction between prefetching and caching instructions.

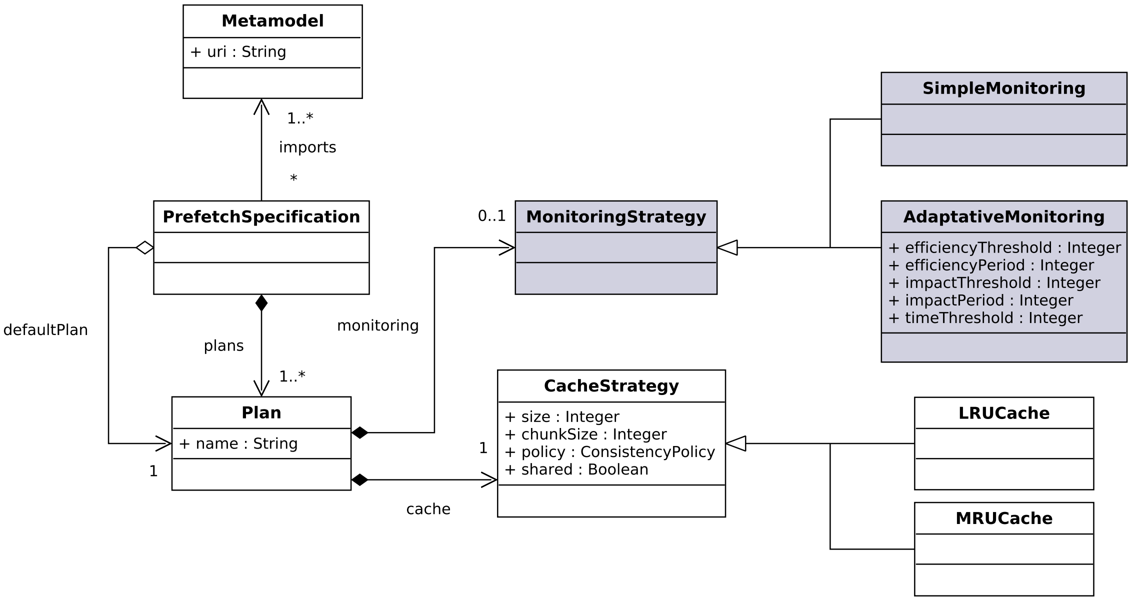

Metamodel
+ uri : String
1..*
imports
*
PrefetchSpecification
defaultPlan
plans
1..*
Plan
+ name : String
1
monitoring
0..1
MonitoringStrategy
SimpleMonitoring
AdaptativeMonitoring
+ efficiencyThreshold : Integer
+ efficiencyPeriod : Integer
+ impactThreshold : Integer
+ impactPeriod : Integer
+ timeThreshold : Integer
CacheStrategy
+ size : Integer
+ chunkSize : Integer
+ policy : ConsistencyPolicy
+ shared : Boolean
1
cache
LRUCache
MRUCache

Figure 4.7 – PREFETCHML Abstract Syntax Metamodel with Monitoring Extensions

Figure 4.7 shows the extended abstract syntax of the PREFETCHML DSL with the new constructs dedicated to monitoring (grey nodes). In addition to its *CacheStrategy*, now a PREFETCHML *Plan* can define an optional *MonitoringStrategy* that collects execution information such as the number of hits and misses for each rule. Current available monitoring strategies are *SimpleMonitoring* that provides these metrics to the modeler under request (Section 4.4.2), and *AdaptativeMonitoring* that uses them together with a set of user-defined thresholds to optimize the prefetching algorithm at runtime (Section 4.4.3).

These new language constructs are used to initialize a new monitoring layer integrated into the PREFETCHML core component through the **MonitorAPI**. This API defines a set of methods to instantiate and parameterize a monitor, and to access computed metrics. These metrics are updated each time an element is loaded by a prefetching rule or accessed from the cache. Monitoring information can be displayed to end-users to help them improve their PREFETCHML plans, or used at runtime by the framework itself to adapt the plan dynamically.

In the following, we detail the metrics computed by the monitoring component and how they can be used by a modeler to improve her PREFETCHML plans.

4.4.2 Simple Monitoring

SimpleMonitoring is the first monitoring strategy we have added to the PREFETCHML grammar (Figure 4.7). It can be defined in a PREFETCHML plan by using the keywords `use simple monitoring`. Once activated, the framework will collect information during the execution, and computes a set of metrics that will be presented on demand to the modeler to help in the quality evaluation of the plan. The metrics are the following:

1. **HitScore:** the total number of elements found and accessed from the cache
2. **MissScore:** the number of elements the persistence framework had to load because of cache misses
3. **MissPerFeature:** categorize the cache misses score per accessed element feature
4. **CachedByRule:** the number of elements cached by each prefetching rule
5. **HitPerRule:** the number of cache hits generated by each prefetching rule
6. **CachedTimestampPerRule:** the list of caching instruction timestamps for each prefetching rule
7. **HitTimestampPerRule:** the list of cache hit timestamps for each prefetching rule
8. **TotalPrefetchingTime:** the total time spent on prefetching/caching actions

Metrics 1-3 correspond to *global accuracy information* that represents the entire prefetching plan usefulness. A good plan will typically generate a high *HitScore* and a low *MissScore*. Misses are categorized by feature (attribute or reference), providing insights on a potential new rule to add to the plan. Metrics 4 and 5 provide fine information for each rule within the PREFETCHML plan: the number of cached elements per rule and the number of hits generated by each rule. This information can be used to evaluate the usefulness of a specific rule (for example by comparing the ratio $HitPerRule/CachedByRule$ to a given threshold). Finally, metrics 6-8 provide time-based information, showing the impact of a given rule over time. This information can be used to find rules that are applied at some point of the computation where they should not allowing modelers to tune the OCL conditions to control when they are triggered. The total prefetching time shows which part of the computation was dedicated to prefetching and caching instructions. This information is particularly interesting when PREFETCHML is applied on top of a backend that does not handle multi-threaded accesses, emphasizing execution time bottlenecks.

Listing 5 shows a possible output of the monitoring component after the execution of the queries presented in the running example (Listing 3) with the PrefetchML plan presented in Listing 4 enabled over a sample model. The table shows, for each rule, the number of executions, the total and average computation time, the number of cached elements, and the number of generated hits. This output format is the default one provided by PREFETCHML , note that time-based metrics are not displayed, but can be accessed through the monitor API.

The table shows that three rules were executed: *r1*,*r2*, and *r3*. Rule *r1* was executed one time, which is the expected behavior for starting rules, that are executed when the prefetching plan is loaded. The table also shows that *r1* cached 45000 elements, but only generated 3000 hits which is low compared to the total hit score (around 1%). Loading these 45000 elements required 6900 milliseconds (15% of the total execution time), which is high compared to the benefit. Removing the rule from the plan would allow the framework to use this execution time to increase the throughput of the other rules. Compared to *r1*, rules *r2* and *r3* cached less elements, but generated most of the global hit score (respectively 52% and 47%).

The last part of the presented listing shows the features that generated cache misses. In our example, there is only one feature (Package. classes) that generated all the misses. This information shows that adding a prefetching rule for this feature would improve the global hit score and thus improve the efficiency of the prefetching plan.

Based on the monitoring information, we were able to detect that *r1* should be removed, and that a new rule *r4* should be added to prefetch the feature that generated the misses. Listing 6 shows the new version of the PREFETCHML plan.

```
=== PrefetchML Monitoring ===
Monitoring started at 12:30:34:145
#Hits: 234 000
#Misses: 125000
#Total Prefetching Time: 45000 ms

== Rule  → #Execution | Tot. Time | Avg. Time | #Cached |  #Hits ==
r1       →          1 |      6900 |      6900 |   45000 |   3000
r2       →       1493 |     14500 |        10 |   12500 | 120000
r3       →       5890 |     23600 |         4 |   30456 | 111000

== Feature               → #Misses ==
Package.ownedElements    →   125000
```

Listing 5 – PREFETCHML Monitoring Example

```
plan samplePlan {
    use cache LRU[size=100,chunk=10]
    rule r2 : on access type Class fetch
        self.methods.modifier
    rule r3 : on access type Class fetch
        self.imports.methods.type.name
        remove type Package
    rule r4 : on access type Package fetch
        self.classes
}
```

Listing 6 – Tuned PREFETCHML Plan

4.4.3 Adaptative Monitoring

Adaptative Monitoring is the second monitoring strategy we have added to the PREFETCHML language (Figure 4.7). It can be defined within a PREFETCHML plan using the keywords `use adaptative monitoring`. When this strategy is set, the framework collects runtime information (as for the *SimpleMonitoring* strategy) and uses a set of heuristics and user-defined thresholds to dynamically adapt prefetching plans to the query computation.

We have defined five heuristics that are used by the framework to disable prefetching rules that are not benefitial for the application. We consider that a rule is harmful if it pollutes the cache content with useless objects and/or if it reduces the throughput of the prefetcher by spending execution time computing loading instructions that are not caching relevant elements. These heuristics can be parametrized by setting the *threshold values* of the *AdaptativeMonitoring* component, and retrieve:

1. **RuleEfficiency:** $Hit_r/Cache_r < threshold \rightarrow disable(r)$
2. **Time-based RuleEfficiency:** $Hit_r/Cache_r < threshold$ during a period of time $t \rightarrow disable(r)$
3. **RuleImpact:** $Hit_r/HitScore < threshold \rightarrow disable(r)$
4. **Time-based RuleImpact:** $Hit_r/HitScore < threshold$ during a period of time $t \rightarrow disable(r)$
5. **TimeImpact:** $TotalTime > threshold \rightarrow \forall r, disable(r)$

RuleEfficiency evaluates the rule efficiency by comparing the number of hits it has generated with the number of cached objects. The rule is disabled when this value goes under a given threshold, meaning that the rule cached too many objects compared to the number of hits it generated. While this strategy can be interesting for simple prefetching plan, it is not adapted to plan involving rules that cache elements that are accessed late in the query computation (typically *starting rules*). To handle this kind of rules we have defined **Time-based RuleEfficiency**, that extends **RuleEfficiency** by disabling a rule if its computed ratio is below a threshold for a given period of time t. The **RuleImpact** heuristic computes the impact of a rule by comparing the number of hits it generates w.r.t the global *HitScore*, and disables the rule if this value goes below a given threshold. This strategy disables low-impact rules, giving more execution time to other rules that are generating more hits. **Time-based RuleImpact** is similar, but it only disables a rule if its computed ratio is below a threshold for a given period of time. Finally, **TimeImpact** is a plan-level strategy that disables all rules if the prefetching time increases over a given threshold.

All the thresholds and time intervals used to define the presented heuristics can be configured in PREFETCHML plans as parameters of the monitoring strategy using their corresponding keywords: `efficiencyThreshold`, `efficiencyPeriod`, `impactThreshold`, etc.

Note that in this initial version of the *Adaptative Monitoring* component rules can only be disabled. Indeed, re-enabling rules is a more complicated task, because computed ratios do not evolve once a rule has been disabled. To allow rules re-activation, we plan to add another monitoring layer that keeps traces of accessed elements and computes which rules would have prefetched them. Monitoring information could also be used to create new rules based on the feature misses. While creating a rule for a single feature is simple, the key point is to find the optimal rule(s) to reduce the number of misses, without polluting the cache content and the prefetcher throughput. This could be done by using constraint solving techniques in order to find the optimal set of rules to create from a set of misses.

4.5 Tool Support

In this Section we present the tool support for the PREFETCHML framework. It is composed of two main components: a language editor (presented in Section 4.5.1) that supports the definition of prefetching and caching rules, and an execution engine with two different integration options: the EMF API and the NeoEMF/Graph persistence framework (presented in Sections 4.5.2 and 4.5.3). The presented components are part of a set of open source Eclipse plugins available on Github[5].

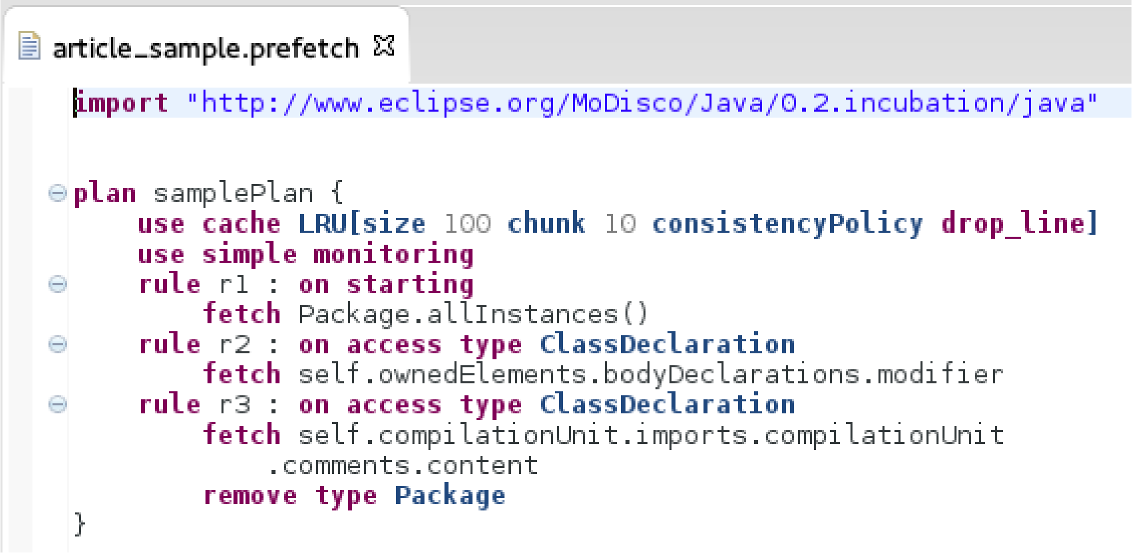
4.5.1 Language Editor

The PREFETCHML language editor is an Eclipse-based editor that allows the creation and the definition of prefetching and caching rules. It is partly generated from the XText grammar presented in Section 4.2.2 and defines utility helpers to validate and navigate the imported metamodel. The editor supports navigation auto-completion by inspecting imported metamodels, and visual validation of prefetching and caching rules by checking reference and attribute existence. Note that monitoring constructs defined in Section 4.4 are available in the editor, allowing to choose a monitoring strategy and define its optional thresholds.

---

[5] https://github.com/atlanmod/Prefetching_Caching_DSL

Figure 4.8 shows an example of the PrefetchML editor that contains the prefetching and caching plan defined in the running example of Section 4.2. The plan contains an additional `use simple monitoring` line that enables simple monitoring capabilities, providing execution information to the modeler.



```
import "http://www.eclipse.org/MoDisco/Java/0.2.incubation/java"


plan samplePlan {
    use cache LRU[size 100 chunk 10 consistencyPolicy drop_line]
    use simple monitoring
    rule r1 : on starting
        fetch Package.allInstances()
    rule r2 : on access type ClassDeclaration
        fetch self.ownedElements.bodyDeclarations.modifier
    rule r3 : on access type ClassDeclaration
        fetch self.compilationUnit.imports.compilationUnit
            .comments.content
        remove type Package
}
```

Figure 4.8 – PrefetchML Rule Editor

4.5.2 EMF Integration

Figure 4.9 shows the integration of PrefetchML into the EMF ecosystem. Note that only two components must be adapted (light grey boxes). The rest are either generic PrefetchML components or standard EMF modules.

In particular, orange boxes represent the standard EMF-based model access architecture: an *EMF-based* tool accesses the model elements through the *EMF API*, that delegates the calls to the *PersistenceFramework* of choice (XMI, CDO, NeoEMF,...), which is responsible of the model storage and element access.

The two added/adapted components are:

— An *Interceptor* that wraps the EMF interface and captures the calls (1) to the EMF API (such as `eGet`, `eSet`, or `eUnset`). EMF calls are then transformed into *EventAPI* calls (2) by deriving the appropriate event object from the called EMF method. For example, an `eGet` is translated into the *accessEvent* method call (8) in Figure 4.5. Once the event has been processed, the *Interceptor* also searches in the cache the requested elements (3). If they are available in the cache, they are directly returned to the EMF-based tool, avoiding a costly database access from the persistence framework. Otherwise, the Interceptor passes on the control to the EMF API to continue the normal process.

— An *EMF Model Connector* that translates the OCL expressions in the prefetching and caching rules into lower-level EMF API calls. The results of those queries are stored in the cache, ready for the Interceptor to request them when necessary.

This integration makes event creation and cache accesses totally transparent to the client application. In addition, it does not make any assumptions about the mechanism used to store the models, and therefore, it can be plugged on top of any EMF-based persistence solution such as CDO, NEOEMF , or the standard XMI persistence layer.

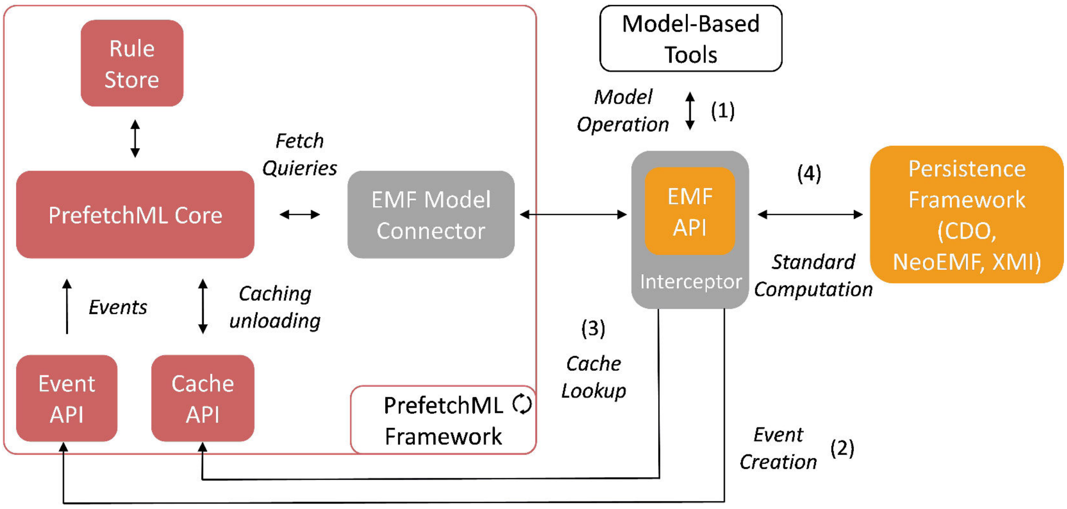

Rule Store
PrefetchML Core
Fetch Quieries
EMF Model Connector
Events
Caching unloading
Event API
Cache API
PrefetchML Framework
Model-Based Tools
Model Operation
(1)
EMF API
Interceptor
(4)
Persistence Framework (CDO, NeoEMF, XMI)
Standard Computation
(3)
Cache Lookup
Event Creation
(2)

Figure 4.9 – Overview of EMF-Based Prefetcher

4.5.3 NeoEMF/Graph Integration

The prefetcher implementation integrated in NeoEMF/Graph (Figure 4.10) uses the same mechanisms as the standard EMF one: it defines an *Interceptor* that captures the calls to the EMF API, and a dedicated *Graph Connector*. While the EMF Connector computes loading instructions at the EMF API level, the *Graph Connector* performs a direct translation from OCL into graph database queries, and delegates the computation to the database, enabling back-end optimizations such as uses of indexes, or query optimizers. The *Graph Connector* caches the results of the queries (i.e. database *vertices*) instead of the EMF objects, limiting execution overhead implied by object reifications. Since this implementation does not rely on the EMF API to navigate the model, it is able to evaluate queries significantly faster than the standard EMF prefetcher (as shown in our experimental results in Section 4.6), thus improving the throughput of the prefetching rule computation. Database vertices are reified into EMF objects when they are accessed from the cache, limiting the initial execution overhead implied by unnecessary reifications.

4.6 Evaluation

In this Section, we evaluate the performance of the PREFETCHML Framework by comparing the performance of executing a set of OCL queries on top of two different

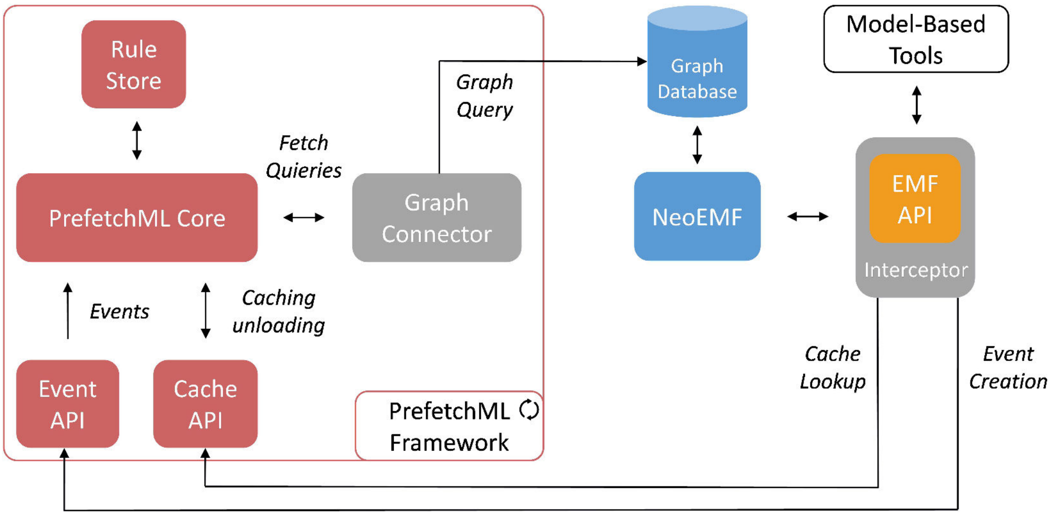

Rule Store
PrefetchML Core
Fetch Quieries
Graph Connector
Graph Query
Events
Caching unloading
Event API
Cache API
PrefetchML Framework
Graph Database
NeoEMF
Model-Based Tools
EMF API
Interceptor
Cache Lookup
Event Creation

Figure 4.10 – Overview of NeoEMF-Based Prefetcher

backends: NEOEMF/GRAPH and NEOEMF/MAP when (i) no prefetching is used and (ii) EMF-based prefetching is active. Models stored in NEOEMF/GRAPH are also evaluated with a third strategy using the dedicated graph-based prefetching presented in the previous section.

Queries are executed in two modeling scenarios: *single query execution* where queries are evaluated individually on the models, and *multiple query execution* where queries are computed sequentially on the models. The first one corresponds to the worst case scenario where the prefetcher and the query itself compete to access the database and retrieve the model elements. The second benchmarked scenario corresponds to the optimal prefetching context: rules target all the queries at once, and the workflow contains idling intervals between each evaluation (due to OCL constraint parsing and syntactic validation), giving more execution time to the prefetcher to load elements from the database.

Note that we do not compare the performance of our solution with existing tools that can be considered related to ours because we could not envision a fair comparison scenario. For instance, Moogle [73] is a model search approach that creates an index to retrieve full models from a repository, where our solution aims to improve performances of queries at the model level. EMF-IncQuery [11] is also not considered as a direct competitor because it does not provide a prefetch mechanism. In addition, EMF-IncQuery was primarily designed to execute queries against models already in the memory which is a different scenario with different trade-offs.

4.6.1 Benchmark Presentation

The executed queries are adapted from the Train Benchmark [103], which is a benchmark used to evaluate the performance of model transformation tools. It defines the *Railway* metamodel, which describes classes to represent railway networks, such as *Routes*, *Tracks*, *Semaphores*, and *Switches*. A complete description of this metamodel can be found on the benchmark repository [6] and in the corresponding publication [103]. In this experiment we use four queries adapted from the ones defined in the benchmark:

— **RouteSensors**: computes a subset of the sensors contained in a given route.
— **RegionSensors**: accesses all the sensors contained in a given region.
— **ConnectedSegments**: navigates all the track elements connected to a sensor.
— **SwitchSet**: retrieves for each entry of a route its corresponding switch elements.

The first query navigates multiple references from a *Route* element in order to retrieve the *Sensors* it directly and indirectly contains. The second one performs a simple navigation to retrieve all the *Sensor* elements contained in a *Region*. The third query performs a long navigation sequence to retrieve all the *Track* elements connected to a given *Route*. Finally, the last query (detailed in Listing 7) computes for a *Route* with a *Semaphore signal* set to *GO* the *Switch* elements that have a different *currentPosition* than the one prescribed in the *Route*'s *SwitchPosition* list [7].

Note that we choose these specific queries among the ones available in the benchmark repository because of their diversity in terms of their number of input, traversed, and returned elements.

```
context Route
def : SwitchSet : Set(Switch) =
self.entry→select(signal = Signal::GO)→collect(
  semaphore | self.follows→collect(
    switchPosition | switchPosition.target→select(switch | switch.currentPosition <>
        switchPosition.position)
  )
)
```

Listing 7 – SwitchSet OCL Query

The prefetching plans used in this benchmark have been created by inspecting the navigation path of the queries. The context type of each expression constitutes the source of *AccessRules*, and navigations are mapped to target patterns. The queries have been executed with a MRU cache that can contain up to 20% of the input model. We choose this cache replacement policy according to Chou and Dewitt [29] who stated that MRU is the best replacement algorithm when a file is being accessed in a looping sequential reference pattern. Another benchmark presenting different cache configurations is available in our previous work [36]. In addition, we compare execution time of the queries when the are executed for the first time and after a warm-up execution to consider the impact of the cache on the performance.

As an example, Listing 8 shows the PREFETCHML specification we have defined from the query shown in Listing 7. It imports the *TrainBenchmark* metamodel and defines a single plan *SwitchSetPlan*. This plan contains a *MRU* cache that can contain up to 20000 elements, and three prefetching rules based on the query navigation paths. The first one, *r1*, is triggered when a *Route* element is accessed and fetches the *Semaphore* it contains through the *entry* reference and its associated *signal*. The rule *r2* is also triggered when a *Route* element is accessed, and loads in memory the *currentPosition* of all the *Switch* elements it contains. Finally, *r3* is executed when a *SwitchPosition* element is accessed, and fetches its *position*. Note that comments showing the mapping between PREFETCHML rules and query navigation paths have been added for the sake of clarity.

---

[6] `https://github.com/FTSRG/trainbenchmark`

[7] Details of the queries can be found at `https://github.com/atlanmod/Prefetching_Caching_DSL`

Table 4.1 – Experimental Set Details

Query	**#Input**	**#Traversed**	**#Res**
RouteSensors	320	28 493	1296
RegionSensors	320	25 431	15 805
ConnectedSegments	15 805	98 922	67 245
SwitchSet	320	14 957	252

```
import "http://www.semanticweb.org/ontologies/2015/trainbenchmark"

plan SwitchSetPlan {
  use cache MRU[size = 20000]

  — self.entry→select(signal = Signal::GO)
  rule r1 : on access type Route
    fetch self.entry.signal

  — self.follows→collect(switchPosition | switchPosition.target
  —     →select(switch | switch.currentPosition [...]
  rule r2 : on access type Route
    fetch self.follows.target.currentPosition

  — [...] <> switchPosition.position)
  rule r3 : on access SwitchPosition
    fetch self.position
}
```

Listing 8 – PREFETCHML Plan for The SwitchSet Query

Prefetching plans are evaluated in two cache settings: a first one with an embedded cache dedicated to prefetched objects, meaning that only elements that have been loaded by the framework are in the cache, and a second one using a *shared* cache storing elements of both the prefetcher and the running application.

The experiments are run over one of the model provided with the benchmark, which contains 102 875 elements, and corresponds to a 19 MB XMI file. The model is initially stored in the benchmarked persistence layers, and queries are evaluated over all the instances of the model that conform to the context of the query. In order to give an idea of the complexity of the queries, we present in Table 4.1 the number of input elements for each query (**#Input**), the number of traversed element during the query computation (**#Traversed**) and the size of the result set for each query (**#Res**).

4.6.2 Results

This section describes the results we obtained by running the experimentation presented above. We first introduce the results for the *single query execution* scenario, then we present the results for the *multiple query execution* scenario. Note that the correctness of query results in both scenarios has been validated by comparing the results of the different configurations with the ones obtained by computing the queries on the initial XMI file without any prefetching and caching enabled using EMFCompare. [8] Presented results have been obtained by using the *Eclipse MDT OCL* toolkit to run the OCL queries on the different persistent frameworks.

---

8. https://www.eclipse.org/emf/compare/

Tables 4.2 and 4.3 present the average execution time (in milliseconds) of 10 executions of the presented queries over the benchmarked model stored in NeoEMF/Graph and NeoEMF/Map, using the *single query execution* scenario. Note that due to garbage collection operations and the concurrent nature of the PREFETCHML engine, as well as the impact of the operating system scheduler, the result for a single execution can vary from another one. However, we found that after five consecutive runs the results tend to stabilize, and we chose to report in this experimentation the average results for ten executions of each query.

we choose to run multiple times the queries to limit the impact of garbage collection operations and concurrency issues on the average result, as well as the impact of the operating system itself on the performances.

Each line presents the result for the kind of prefetching that has been used: no prefetching (*NoPref.*), EMF-Prefetching with dedicated cache (*EMF Pref.*), and EMF-Prefetching with shared cache (*EMF-Pref. (Shared)*). Note that Table 4.2 contains an additional line corresponding to the graph specific prefetcher.

Table 4.4 shows the average execution time (in milliseconds) of 10 executions of all the queries over NeoEMF/Graph and NeoEMF/Map in the *multiple query execution* scenario.

In the first part of the tables, the cells show the execution time in milliseconds of the query the first time it is executed (Cold Execution). In this configuration, the cache is initially empty, and benefits of prefetching depend only on the accuracy of the plan (to maximize the cache hits) and the complexity of the prefetching instructions (the more complex they are the more time the background process has to advance on the prefetching of the next objects to access). In the second part, results show the execution time of a second execution of the query when part of the loaded elements has been cached during the first computation (Warmed Execution).

Table 4.2 – NeoEMF/Graph Query Execution Time in milliseconds

(a) Cold Execution

	Route Sensors	**Region Sensors**	**Connected Segments**	**Switch Set**
No Pref.	30 294	1633	14 801	915
EMF Pref.	30 028	1982	14 625	1047
EMF Pref. (Shared)	28 902	1803	13 850	998
Graph Pref.	25 143	1477	11 811	830

(b) Warmed Execution

	Route Sensors	**Region Sensors**	**Connected Segments**	**Switch Set**
No Pref.	16 087	908	8887	528
EMF Pref.	259	183	874	130
EMF Pref. (Shared)	236	179	877	130
Graph Pref.	1140	445	2081	264

Table 4.3 – NeoEMF/Map Query Execution Time in milliseconds

(a) Cold Execution

	Route Sensors	Region Sensors	Connected Segments	Switch Set
No Pref.	33 770	1307	11 935	499
EMF Pref.	2515	1210	10 166	410
EMF Pref. (Shared)	1640	1090	7488	353

(b) Warmed Execution

	Route Sensors	Region Sensors	Connected Segments	Switch Set
No Pref.	33 279	1129	11 389	271
EMF Pref.	203	167	783	105
EMF Pref. (Shared)	221	161	837	105

Table 4.4 – Multiple Query Execution Time in milliseconds

(a) Cold Execution

	NeoEMF/Graph	NeoEMF/Map
No Pref.	47 312	45 965
EMF Pref.	38 597	16 897
EMF Pref. (Shared)	34 522	13 742
Graph Pref.	31 479	–

(b) Warmed Execution

	NeoEMF/Graph	NeoEMF/Map
No Pref.	23 471	47 823
EMF Pref.	1698	1896
EMF Pref. (Shared)	1681	1793
Graph Pref.	3489	–

4.6.3 Discussion

The main conclusions we can draw from these results (Tables 4.2 to 4.4) are:

- PREFETCHML improves the execution time of all the queries on top of NEOEMF/MAP for both scenarios. Execution time is improved by around 16 % for *RegionSensor*, and up to 95 % for *RouteSensors*. These results can be explained by the concurrent nature of the backend, that can be accessed by the query computation and the PREFETCHML framework at the same time without execution time bottleneck. In addition, NEOEMF/MAP does not contain any model element cache, and the second execution of the queries directly benefits from the cache embedded in PREFETCHML , showing execution time improvement up to 99 % for the *RouteSensor* query.
- EMF-based prefetching improves the execution time of first time computation of queries that perform complex and multiple navigations (*RouteSensors* and *ConnectedSegments* queries) on top of NEOEMF/GRAPH . The EMF-Prefetcher also drastically improves the performance of the second execution of the queries: an important part of the navigated objects is contained in the cache, limiting the database overhead. However, when the query is simple such as *RegionSensors* or only contains independent navigations such as *SwitchSet*, the EMF prefetcher results in a small execution overhead since the prefetching task takes time to execute and with simple queries it cannot save time by fetching elements in the background while the query is processed.
- Graph-based prefetcher is faster than the EMF one on the first execution of the queries in both scenarios because prefetching queries can benefit from the database query optimizations (such as indexes), to quickly load objects to be used in the query when initial parts of the query are still being executed, i.e. the prefetcher is able to run faster than the computed query. This increases the number of cache hits in a cold setup, improving the overall execution time. Conversely, graph-based prefetcher is slower than the EMF-based one on later executions because it stores in the cache the vertices corresponding to the requested objects and not the objects themselves, therefore an extra time is needed to reify those vertices into EMF-compatible objects.
- Sharing the cache between the PrefetchML framework and the running application globally improves the performances for all the queries, w.r.t the performances without sharing the cache. This is particularly true for simple queries such as *RegionSensors*, where the prefetcher and the query are computing the same information at the same time, and sharing the fetched elements reduces the concurrency bottlenecks.

To summarize our results, the PREFETCHML framework is an interesting solution to improve execution time of model queries over EMF models. The gains in terms of execution time are always positive for NEOEMF -based implementations. Using PREFETCHML on top of the standard EMF interface is also always better on a warmed execution scenario, but for ad hoc scenarios where most queries may be executed a single time and may not be related to each other, PREFETCHML adds sometimes a small overhead to the overall the query computation time. A tuning process, taking into account the kind of ad hoc queries typically executed (e.g. their likely footprint), may be needed to come up with an optimal prefetching strategy.

4.7 Conclusions

We presented the PREFETCHML DSL, an event-based language that describes prefetching and caching rules over models. Prefetching rules are defined at the metamodel level and allow designers to describe the event conditions to activate the prefetch, the objects to prefetch, and the customization of the cache policy. Since OCL is used to write the rule conditions, PREFETCHML definitions are independent from the underlying persistence back-end and storage mechanism.

Rules are grouped into plans and several plans can be loaded/unloaded for the same model, to represent fetching and caching instructions specially suited for a given usage scenario. Note that some automation/guidelines could be added to help on defining a good plan for a specific use-case in order to make the approach more user-friendly. PREFETCHML embeds a monitoring component that partially addresses this issue by helping modelers to detect those undesired scenarios and optimize their existing plans. The execution framework has been implemented on top of the EMF as well as on NEOEMF/GRAPH , and experimental results show a significant execution time improvement compared to non-prefetching use cases.

PREFETCHML satisfies all the requirements listed in Section 4.1. Prefetching and caching rules are defined using a high-level DSL embedding the OCL, hiding the underlying database used to store the model (**Rq1**). The EMF integration also provides a generic way to define prefetching rules for every EMF-based persistence framework (**Rq2**), like NEOEMF and CDO. Note that an implementation tailored to NEOEMF is also provided to enhance performance. Prefetching rules are defined at the metamodel level, but the expressiveness of OCL allows to refer to specific subset of model elements if needed (**Rq3**). In Section 4.2 we presented the grammar of the language and emphasized that several plans can be created to optimize different usage scenario (**Rq4**). The PREFETCHML DSL is a readable language that eases designers' tasks on writing and updating their prefetching and caching plan (**Rq5**). Since the rules are defined at the metamodel level, created plans do not contain low-level details that would make plan definition and maintenance difficult. Finally, we have integrated a monitoring component in our framework that can provide a set of metrics allowing modelers to finely optimize their PREFETCHML plans (**Rq6**). This monitoring component is also used to automatically disable harmful rules during the execution.

5

Efficient Queries

Model queries are one of the cornerstones of MDE processes. They constitute the basis of several modeling activities, such as model validation [10], where model queries are used to retrieve part of the model to verify and check constraints over, or model transformations [58], where queries are used to navigate source models, create target elements, and build the output model.

In the current modeling toolchains, queries are defined using high-level, expressive languages such as the OCL [83] standard. Existing technical solutions typically embed an editor that helps modelers to define their queries, and an interpreter that translates high-level language constructs into sequences of low-level API calls which are then handled by the modeling framework.

While the evolution of model persistence layers —such as NEOEMF — has improved the support for managing large models by using advanced storage mechanism, *lazy-loading* techniques, and prefetching/caching components to handle large models [88, 43, 35], they are just a partial solution to the scalability problem in current modeling frameworks. In its core, all frameworks are based on the use of low-level model handling APIs that are focused on manipulating individual model elements and do not provide support for generic model query computation. This is clearly inefficient because (i) the API granularity is too fine to benefit from the advanced query capabilities of the data-store and (ii) an important time and memory overhead is necessary to construct navigable intermediate objects that can be used to interact with the API.

Based on our experience on the alignment of modeling-level constructs and NoSQL database primitives, we have defined a set of requirements that should be addressed to overcome the modeling frameworks' API limitations and enable efficient query support for the novel generation of model persistence solutions:

Rq1 the solution should generate database queries in order to benefit from the NoSQL database structures and optimizations, and limit network overhead

Rq2 the framework should support queries expressed using a state of the art model query language such as OCL or EOL

Rq3 the framework must outperform existing querying solutions relying on low-level model handling APIs when applied on actual scalable model persistence techniques

Rq4 low-level details such as the generated database query and intermediate translation steps should be hidden from the end user

In the following, we present MOGWAÏ , an efficient and scalable query framework that addresses these requirements to enable complex query computation on top of large models stored in actual model persistence frameworks. MOGWAÏ translates model queries written in OCL into expressions of a graph traversal language, Gremlin [109] [1], which is directly used to query models stored in a NoSQL backend. A prototype implementation of MOGWAÏ integrated into the Eclipse Modeling Framework is also provided and evaluated. We show that bypassing the framework API (the EMF API in our case) to delegate the query to the database is more efficient and scalable than existing solutions relying on the EMF API when applied to large model and/or complex queries. To evaluate our solution, we perform a set of queries extracted from existing software modernization use-cases [19] and compare the results against full EMF API and existing query frameworks over several persistence solutions.

This chapter is organized as follows: Section 5.1 presents the existing model query solutions and emphasizes the research problem to address. Section 5.2.1 introduces Gremlin, a language to query multiple NoSQL databases, Section 5.2 presents the architecture of the MOGWAÏ framework and its query translation process. Section 5.3 introduces the implementation of our solution and its integration in the NEOEMF framework. We evaluate our approach and compare it with existing solutions in Section 5.4. Finally, Section 5.5 summarizes the contributions and draws conclusions.

5.1 State of the Art

Several solutions have been proposed to efficiently compute queries, define contraints, and express derived features over models. Most of the existing approaches rely on OCL, the OMG standard model query language, and are integrated in existing modeling frameworks. In this section we present the different model query solutions, focusing on their specific features and the way they access the underlying model. Finally, we summarize the benefits and drawbacks of existing approaches, and we propose a research problem to address in order to improve query computation over large models.

5.1.1 Model Query Solutions

Few solutions have been proposed to query models efficiently by translating high-level model query specifications into database-specific languages. These approaches usually target the database itself, bypassing the standard modeling APIs to enhance query performance and benefit from the low-level database optimizations.

The Model Query Translator (MQT) framework [37] is a persistence and query solution for EMF models built on top of the Epsilon platform. MQT is designed to compute EOL queries efficiently, by relying on a model to RDBMS mapping that stores models in relational databases, and analyzes input EOL expressions to create equivalent SQL queries. This query pre-processing analyses the entire EOL expression to compute, and group the query operations into SQL expressions that can be efficiently handled by the database, limiting the memory consumption of the application and improving query computation performances. However, MQT only targets relational databases, and therefore is not the best solution for the new generation of NoSQL persistence frameworks used nowadays.

---

[1] "Mogwaï" is inspired by the species of Gizmo, the main character of the Gremlins movie

Also relevant to our work are the approaches targeting the translation of OCL expressions to other languages/technologies [24]. For example, Heidenreichin et al. [40, 51] proposed a solution to automatically build a relational database from a UML representation of an application, and translates the OCL invariants into database constraints and triggers. A similar approach was proposed by Brambilla and Cabot [17] in the field of web applications. In that case, model-level queries and constraints could be translated into triggers or as views that are used to ensure data consistency at the database level. Note that in all these scenarios the goal is to use OCL for code-generation purposes as part of a data validation component but does not focus on computing model queries efficiently. In addition, the proposed solutions aim to generate code that ensures consistency at the data level, and are not designed to operate at the metamodel-level required by model query languages.

Alternative approaches relying on the use of modeling framework APIs to compute model queries have been proposed to decouple the data representation strategy from the actual query solution. The use of these APIs allows to evaluate model queries independently of the underlying storage solution, at the cost of an execution and memory overhead to align the intermediate model access layer to the concrete data-store interface.

The MDT OCL framework is an Eclipse project that provides an environment to define, parse, and evaluate OCL invariants and queries over models. It provides an EMF-based OCL metamodel and a query editor that allows to define and evaluate OCL expressions over an existing metamodel. The MDT OCL project is tightly integrated in the Eclipse platform, and provides a set of views and perspectives to interactively compute queries on an EMF model, check constraints on the fly, and validate models dynamically. The framework itself relies on the EMF API to navigate the underlying model, and therefore it is compatible with all the EMF-based persistence solutions listed in chapter 3, such as NeoEMF [35] and CDO [43].

In addition, CDO also embeds a server-side OCL query interface that takes benefit of the CDO database structure to evaluate OCL queries efficiently. CDO-OCL provides a server-side API that translates model queries into low-level modeling API calls that are computed by the data-store, limiting network overhead. The produced query fragments can benefit from the database internal structures (such as indexes), and generated queries are partially optimized by a simple translation mechanism (such as combining navigation operations into a single database query). However, CDO-OCL still relies on the low-level modeling API to compute the final query, that generates fragmented database accesses which cannot be handled efficiently and optimized by the database.

EOL is the base query language provided by the Epsilon modeling framework. It is designed as a superset of OCL that provides side-effect operations and an imperative syntax. EOL is the foundation language of a variety of Epsilon technologies, such as

EML [64] for model merging, and ETL [67] for model transformations [2]. The Epsilon framework provides an EOL interpreter that relies on the Epsilon API to compute and evaluate model queries and can be extended by data-store specific solutions.

Hawk [3] is a model indexer built on top of the Epsilon platform that computes EOL queries on top of EMF models stored in graph databases (Neo4j and OrientDB). The framework provides its own implementation of the EOL execution engine that directly targets the underlying database instead of using the EMF API. This approach allows to use database querying facilities (in particular indexes and caches) to improve execution performances and limit the memory consumption. However, Hawks suffers from the same issue as CDO-OCL: it still relies on the low-level EOL API to compute queries, that generates fragmented and inefficient database accesses. This is particularly true for the Neo4j backend, that is optimized to compute complex queries expressed with its dedicated pattern matching language Cypher, but is not designed to compute atomic, low-level accesses.

EMF Query [105] is a framework that provides an internal Java DSL to query a model with a SQL like language. It includes a set of tools to ease the definition of queries and manipulate results as a model. EMF Query can be seen as a wrapper around the standard EMF API: it allows to express complex queries with database-related constructs (such as *select*, *from*, *group by*, etc) that are then computed using the modeling framework API. As for MDT OCL, the solution strongly relies on the EMF API to access and manipulate the model, making it compatible with EMF-based persistence solutions, at the cost of a performance and memory overhead when combined with current *lazy-loading* persistence solutions.

Finally, EMF-IncQuery [11] is an incremental pattern matching engine to query EMF models. It relies on a RETE network [45] that provides efficient incremental (re)computation of model queries. EMF-IncQuery is integrated into the Viatra platform [9], a reactive model transformation framework that aims to optimize model transformations by providing event-based, incremental re-computation of transformations. This query solution tackles the modeling API limitations by only using its internal structures to evaluate model queries, propagate changes, and return results. While EMF-IncQuery has shown impressive results to perform queries multiple times on a model [10], it also presents two major drawbacks when moving to large models: (i) the framework still relies on the model handling APIs to initialize its internal data structures by performing a complete model traversal, and (ii) the cache that receives the events and propagates the changes is fully stored in memory to improve query re-computation. The former has a significant impact in terms of performances when coupled with *lazy-loading* solutions that are typically inefficient to handle model traversals [116], and the later limits the scalability of the approach to models that can actually fit in the available memory [3].

Table 5.1 summarizes the main features of the state of the art solutions. We base our analysis on four criterias: (i) if the solution is based on a modeling API (for example EMF) to access the underlying model or if (ii) it is based on a translational approach that generates low-level database queries from high-level modeling languages. We also distinguish if (iii) the approach computes queries in memory or (iv) if it delegates the compu-

---

[2] A complete list of Epsilon-based languages is available online at `https://eclipse.org/epsilon/`

[3] Note that partial solutions are provided to improve large model supports on the framework's wiki at `https://wiki.eclipse.org/VIATRA/Query/FAQ`

Query Solution	API-based	Translation	In-memory	In database	
				Relational	NoSQL
MDT-OCL	✅	❌	✅	✔	✔
EMF-Query	✅	❌	✅	✔	✔
EOL	✅	❌	✅	✔	✔
CDO-OCL	✅	✔	✅	✅	❌
Hawk	✅	✔	✅	❌	✅
EMF-IncQuery	✅	❌	✅	❌	❌
MQT	❌	✅	❌	✅	❌
SQL generators[1]	❌	✅	❌	✅	❌

[1] We generalize the features of the UML/OCL to SQL generative approaches detailed in [51] and [17].

Table 5.1 – Features of existing query solutions

tation to the database layer. Note that we differentiate solutions that target relational and NoSQL databases. For each presented tool, fully supported features are represented with green checks, partially supported features are shown as grey checks, and unsupported features are represented as red crosses.

5.1.2 Summary and Research Problem

Our analysis emphasizes that most of the proposed solutions rely on the usage of modeling API to compute queries. While this architecture allows to easily integrate query solutions into existing tool chains, the low-level model handling APIs usually provided by the modeling frameworks have not been designed to fit the new schema-less architecture of the current scalable persistence solutions. Query translation approaches address this issue by generating database queries that bypass the modeling layer to directly compute them on the backend side. However, to the best of our knowledge the existing solutions are either focused on ensuring consistency at the data level or are not designed to generate NoSQL query language expressions to target the current persistence solutions.

In addition, there is currently no querying solution that fully addresses the requirements listed above, and **there is a need to provide a translation solution that produces optimized NoSQL expressions from high-level model query languages**. In the following, we present MOGWAÏ , a query framework that aims to address the presented requirements by proposing a translational approach that takes as its input OCL expressions and translates them into efficient NoSQL database queries expressed in Gremlin. We detail the translation process from model-level queries to database languages, and present the integration of the framework in existing modeling tool chains. Finally, we compare the MOGWAÏ framework with state of the art solutions and discuss the benefits and drawbacks of the approach.

5.2 The Mogwaï Framework

The MOGWAÏ framework is our proposal for handling model queries on large models stored in NoSQL persistence solutions. It relies on a model-to-model transformation that generates queries expressed using the Gremlin traversal language, a high-level NoSQL query language that targets multiple databases . At the modeling level, we assume that model queries are expressed using the OCL language, the OMG standard to describe invariants, operation contracts, and query expressions[4]. Note that the approach we present in this section can also be applied to different input and output languages, such as EOL for the modeling side and Hibernate Object/Grid Mapper (OGM) JPQL on the database side.

In this Section, we first introduce Gremlin, the NoSQL query language we choose as the target for our MOGWAÏ framework, and we present our query approach and compare it with standard API-based query solution. Then we detail the different components of our proposal: the Gremlin metamodel we have defined to represent the abstract syntax of Gremlin traversals, the individual mapping from OCL expressions to Gremlin steps, and finally the transformation process that combines the generated steps into a complete Gremlin traversal.

5.2.1 The Gremlin Query Language

Motivation

As we emphasize in Chapters 2 and 3, NoSQL databases have proven their efficiency to store and manipulate large models. Nevertheless, their diversity in terms of primitive constructs (key-value pairs, documents, nodes and relationships, etc) and supported features makes them hard to unify under a standard interface to be used as a generic query solution. Specifically, some NoSQL databases such as Cassandra and Neo4j provide a high-level query language to manipulate data efficiently, while key-value stores such as Redis typically provide a simpler API to access values from specific keys.

Several approaches have been proposed to unify NoSQL database accesses. UnQL[5] is a prototype of language developed by CouchBase and SQLite that is designed to query document and RDBMS databases. The language extends SQL with additional constructs to query and navigate semi-structured data, and provides features to select and manipulate complex document structures. However, according to the UnQL wiki, the project has not been updated since 2013.

The Hibernate OGM[6] is a persistence framework that implements the Java Persistence API (JPA) for multiple NoSQL solutions. It is based on the well-known Hibernate ORM persistence framework [5] and supports several types of databases, such as key-value stores (Infinispan, EhCache), document databases (MongoDB), and graph database (Neo4j). Hibernate OGM provides an implementation of JPQL, a SQL-like query language that allows to access and update persisted elements. While this abstraction is a first step in bridging the gap between different NoSQL data representations, recent studies have shown that there is a mismatch between JPA and NoSQL that leads to significant execution time overheads compared to native solutions [92].

---

[4] Details on OCL are provided in Chapter 2.

[5] http://unql.sqlite.org

[6] `http://hibernate.org/ogm)`

Blueprints [107] is an initiative developed by the Apache Tinkerpop project [110], that aims to create an interface to unify NoSQL database accesses under a common API. Initially designed for graph databases, Blueprints has been implemented by a large number of databases such as Neo4j [7], OrientDB [8], and MongoDB [9]. Blueprints is, to our knowledge, the only active project unifying several NoSQL databases [10].

Blueprints is the base of the Tinkerpop stack, a set of tools to store, serialize, manipulate, and query databases based on a graph representation of their content. When a database implements the Blueprints API, it automatically benefits of these high-level features, and can be manipulate using Gremlin [109], the query language designed to access Blueprints databases. Gremlin relies on a lazy data-flow framework [108] and is able to navigate, transform, or filter elements in a graph. It can express graph traversals finely using navigation *steps*, and shows positive performance results when compared to Cypher, the pattern matching language used to query natively the well-established Neo4j graph database [52].

The generic nature of the Gremlin language, as well as our experience on model persistence in graph databases (detailed in section 3.3) has motivated our choice to use it as the target language in our query approach. Furthermore, the adoption of the Tinkerpop stack by several major actors in the NoSQL community makes Gremlin an interesting candidate to target additional model data stores such as MongoDB-based persistence frameworks [88, 21].

Language

Gremlin is a Groovy [11] domain-specific language built on top of *Pipes* [108], a dataflow framework based on process graphs. A process graph is composed of vertices representing computational units and communication edges which can be combined to create a complex processing. In the Gremlin terminology, these complex processing are called *traversals*, and are composed of a chain of simple computational units named *steps*. Gremlin defines four types of steps:

- **Transform steps:** functions mapping inputs of a given type to outputs of another type. They constitute the core of Gremlin: they provide access to adjacent vertices, incoming and outgoing edges, and properties. In addition to built-in navigation steps, Gremlin defines a generic *transformation* step that applies a function to its input and returns the computed results.
- **Filter steps:** functions to select or reject input elements w.r.t. a given condition. They are used to check property existence, compare values, remove duplicated results, or retain particular objects in a traversal.
- **Branch steps:** functions to split the computation into several sub-traversals and merge their results.
- **Side-effect steps:** functions returning their input values and applying side-effect

---

[7] `http://neo4j.com/`
[8] `http://orientdb.com/`
[9] `https://www.mongodb.org/`
[10] Implementation list is available at `https://github.com/tinkerpop/blueprints`
[11] `http://www.groovy-lang.org/`

operations (edge or vertex creation, property update, variable definition or assignation).

In addition, the *step* interface provides a set of built-in methods to access meta information: number of objects in a step, output existence, or first element in a step. These methods can be called inside a traversal to control its execution or check particular elements in a step.

Gremlin allows the definition of custom steps, functions, and variables to handle query results. For example, it is possible to assign the result of a traversal to a table and use it in another traversal, or define a custom step to handle a particular processing.

In what follows, we describe some simple Gremlin examples based on the graph representation of the running example presented in Chapters 2 and 3. A Gremlin traversal begins with a *Start* step. It gives access to graph level informations such as indexes, vertex and edge lookups, and property based queries. For example, the traversal below performs a query on the index *classes* that returns the vertices indexed with the name *Package*, representing the *Package* class in the Figure 6.2. In our example, this class corresponds to vertex `Package`. The results of a start step constitute the input of next steps in the traversal.

```
g.idx("classes")[[name:"Package"]]; // v(Package)
```

The most common steps are *transform* steps, that allow navigation in a graph. The steps *outE(rel)* and *inE(rel)* navigate from input vertices to their outgoing and incoming edges, respectively, using the relationship *rel* as filter. *inV* and *outV* are their opposite: they compute head and tail vertices of an edge. For example, the following traversal returns all the vertices that are related to the vertex 3 by the relationship *classes*. The *Start* step used to access it is a vertex lookup.

```
g.V(p1).outE("classes").inV; // [v(c1), v(c2)]
```

Filter steps are used to select or reject a subset of input elements given a condition. They are used to filter vertices given a property value, remove duplicate elements in the traversal, or get the elements of a previous step. For example, the following traversal returns all the vertices related to vertex 3 by the relationship *classes* that have a property *name* with a value longer than 10 characters. The local variable `it` in the filter closure is a Groovy feature representing the element the closure is applied to.

```
g.V(p1).outE("classes").inV
  .has("name").filter{it.name == 'c1'}; // [v(c1)]
```

Branch steps are particular steps used to split a traversal into sub queries, and merge their results. As an example, the following traversal collects all the *id* and *name* properties for the vertices related to vertex 3 by the relationship *classes*. The computation is split using the *copySplit* step and merged in the parent traversal using *exhaustMerge*.

```
g.V(p1).outE("classes").inV.copySplit(
  _().name,_().id).exhaustMerge(); // ['class1', 'class2', 'c1', 'c2']
```

Finally, *side-effect* steps modify a graph, compute a value, or assign variables in a traversal. They are used to map elements to computed values, fill collections with step results, update properties, or create elements. For example, it is possible to store the result of the previous traversal in a table using the step *Fill*.

```
def table = [];
g.V(p1).outE("classes").inV.copySplit(
  _().name,_().id).exhaustMerge().fill(table);
// table = ['class1', 'class2', 'c1', 'c2']
```

5.2.2 The Mogwaï Query Approach

Figure 5.1 shows the overall transformation and query process of the MOGWAÏ framework (top part) and compares it with standard EMF[12] API based approaches such as Eclipse MDT OCL or EMF Query (bottom part).

An initial textual OCL expression is parsed into a model of its abstract syntax conforming to the OCL metamodel. This model constitutes the input of a model-to-model transformation that analyzes the query and translates its operations to generates an equivalent Gremlin traversal. The resulting Gremlin query is then sent to the Blueprints database for its execution, and query results are returned to the modeler.

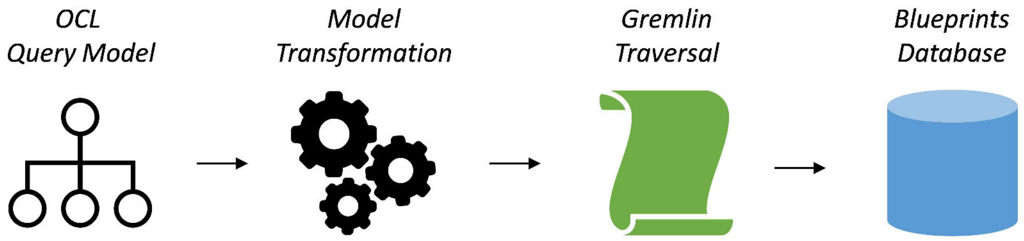
The main difference with existing query frameworks is that the MOGWAÏ framework does not rely on the EMF API to perform a query. As we detailed in Section 5.1, API based query frameworks generally translate OCL queries into a sequence of low-level API calls, which are then performed one after the other on the database. While this approach has the benefit to be compatible with every EMF-based application, it does not take full advantage of the database structure and query optimizations. Furthermore, each object fetched from the database has to be reified to be navigable, even if it is not going to be part of the end result. Therefore, execution time of the EMF-based solutions strongly depends on the number of intermediate objects reified from the database (which depends on the complexity of the query but also on the size of the model, bigger models will need a larger number of reified objects to represent the intermediate steps) while for the MOGWAÏ framework, everyhting is computed in the database itself, and there is no need to reify intermediate elements, limiting execution time and memory consumption overhead.

Once the Gremlin traversal has been executed on the database side, the results are handled by the MOGWAÏ framework that delegates their reification to the persistence side if necessary (i. e. if the query results can be reified as model elements). Note that this reification phase is performed after the entire query computation, and does not reify any intermediate element. Using this architecture, it is possible to plug the MOGWAÏ framework on top of any persistence framework that uses a Blueprints-compatible database and provides a mechanism to reify database records into modeling elements, such as NEOEMF and CDO.

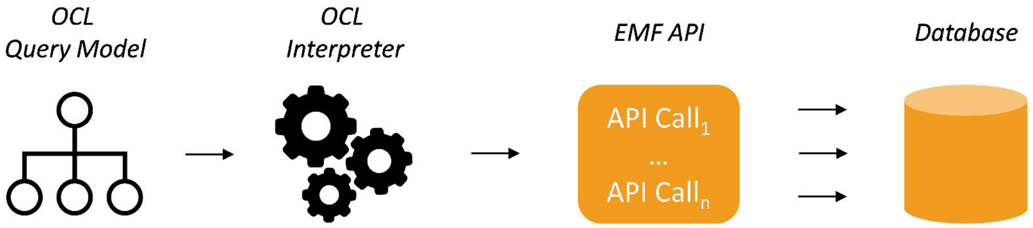
To sum up, the translation process embedded in our solution generates a single Gremlin traversal from an OCL query and runs it over the database. This solution provides three main benefits: (i) it delegates the query computation to the database, taking full advantage of the built-in caches, indexes, and query optimizers, (ii) queries are executed once and are not fragmented into low-level, atomic accesses, and (iii) it reduces the network overhead by sending an entire traversal at once instead of fragmented queries.

---

[12] We focus the explanation on the EMF framework but results are generalizable to all other modeling frameworks we are familiar with.



(a) Mogwaï Query Framework



(b) EMF-based Query Frameworks

Figure 5.1 – Comparison of OCL execution

5.2.3 Gremlin Metamodel

The output of our model-to-model transformation is a model representing the abstract syntax of the Gremlin traversal to compute. Since the Gremlin project does not provide a formal representation of its grammar we propose our own Gremlin metamodel (presented in Figure 5.2) based on the language constructs' documentation provided online [13]. Note that since Gremlin is a Groovy based language, it could have been possible to reuse existing Java or Groovy metamodels to express our queries, however, these metamodels are too complex for our needs, and they miss an easy way to define the step concept, a core feature in the Gremlin language.

Figure 5.2 presents an exerpt of the Gremlin metamodel we have defined. A *GremlinTraversal* contains a set of *Instructions* that can be either *TraversalElements* or *Expressions*. Supported *Expressions* retrieve *Literal* values, *UnaryExpressions*, and *BinaryExpressions* (such as boolean or mathematic operators). *UnaryExpressions* and *BinaryExpressions* contain respectively one and two inner *Instructions*. A *TraversalElement* is a single computation step in a Gremlin traversal that can be either a native Gremlin *Step* or a Groovy *MethodCall*. *TraversalElements* are organized through a composite pattern: each *Step* in a traversal has a *next* containment reference that links to the next *TraversalElement* to compute, allowing to create complex Gremlin traversal expressions by chaining simple computation *Steps* together. We have represented all the *Step* types of the grammar in our metamodel: *InEStep* and *OutEStep* are respectively incoming and outgoing edge navigation steps that retrieves for a given vertex the edges containing the label *relName*, *InVStep* and *OutVStep* perform that same kind of navigations from edges to vertices, and *FilterStep* instances allows to express filtering condition through a *Clo-*

---

[13] http://gremlindocs.spmallette.documentup.com/

sure that contains a set of *Instructions*. Note that for the sake of readability we do not present all the supported *Steps* and *MethodCalls* in this excerpt, but a complete definition is provided in the project repository [14].

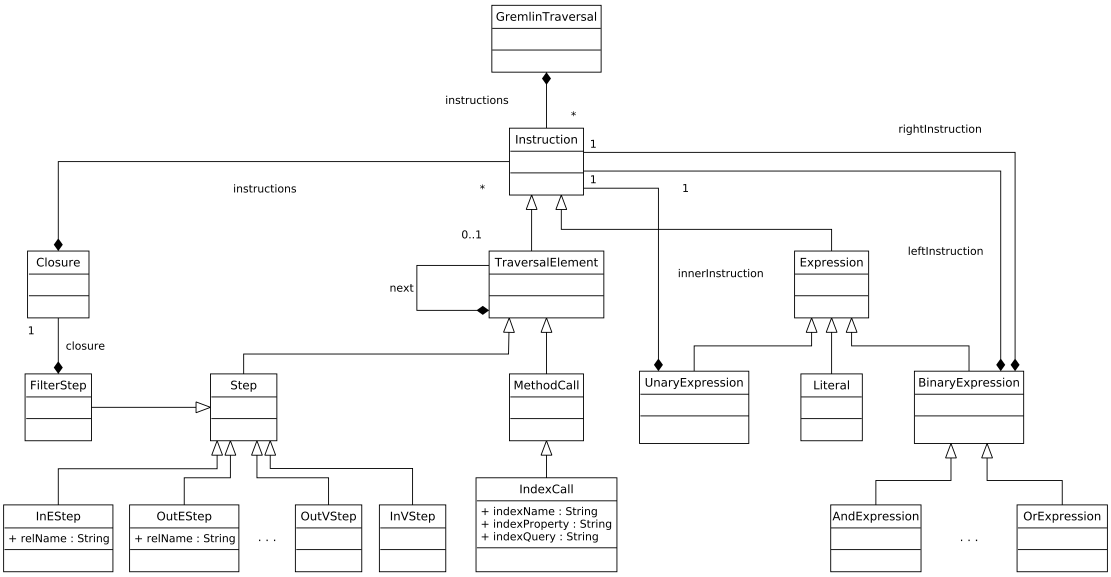

GremlinTraversal
instructions
*
Instruction
1
rightInstruction
1
instructions
*
1
0..1
Closure
TraversalElement
next
innerInstruction
Expression
leftInstruction
1
closure
FilterStep
Step
MethodCall
UnaryExpression
Literal
BinaryExpression
InEStep
+ relName : String
OutEStep
+ relName : String
. . .
OutVStep
InVStep
IndexCall
+ indexName : String
+ indexProperty : String
+ indexQuery : String
AndExpression
. . .
OrExpression

Figure 5.2 – Extract of Gremlin Metamodel

5.2.4 Mapping of OCL expressions

The MOGWAÏ transformation process that generates Gremlin traversal relies on an initial low-level translation of individual OCL expressions into their corresponding Gremlin steps. This mapping is presented in Table 7.1 and shows, for each supported OCL expression, the corresponding Gremlin statement(s). We have divided the supported expressions into four categories based on Gremlin step types: transformations, collection operations, iterators, and general expressions. Note that other types of OCL expressions not explicitly listed in the table can be first expressed in terms of those that have been defined by Cabot and Teniente [25] and therefore be also covered by our mapping.

Expressions in the first group, transformation expressions, return a computed value from their input. OCL operations that navigate the model elements are mapped to navigation steps: `Type` access is translated into an index call returning the vertex representing the type, assuming the type exists. `AllInstances` collection is mapped to a traversal returning adjacent vertices on the Type vertex having an *instanceof* outgoing edge. Reference and attribute `collect` operations are respectively mapped to an adjacent vertex collection on the reference name and a property step accessing the attribute. Type conformance is checked by comparing the adjacent *instanceof* vertex with the type one using a generic *transform step*. Finally, attribute and reference `collects` applied after a type casting operation are mapped as regular `collect` operations, because each vertex in the database contains its inherited attributes and edges.

The second group, operations on collections, has a particular mapping, because Gremlin is an iterator-based language that does not allow to modify the content of a step during the computation of a traversal. Therefore, OCL expressions that change the content of a collection have to be handled in a specific way to provide the same behavior at the Gremlin level. `Union`, `intersection` and `set difference` expressions are mapped to the *fill* step, which puts the result of the traversal into a variable. We have extended the Gremlin language by adding *union*, *intersection*, and *subtract* methods that compute the result of those operations from the variables storing the traversed elements. These additional methods return a step instance, that allows to manipulate their result in a new traversal if necessary. `Including` and `excluding` operations can be computed by transforming the step content into a collection and by calling the corresponding Groovy operation. The same collection transformation is done to handle `includes` and `excludes` operations, which are translated into containment checks. Finally, functions returning the size and the first element of a collection are mapped to *count* and *first* step methods. Note that there is no specific method to check if a collection is empty in Gremlin but this can be achieved by calling a Groovy collection transformation.

---

[14] https://github.com/atlanmod/Mogwai

Iterator expressions are OCL operations that evaluate a condition over a collection, and return either the filtered collection or boolean value. The `Select` operation is mapped to a *filter step* that defines a closure containing the translation of the select condition. `Reject` expressions are mapped the same way with a negation of its condition. The mapping of `exists` and `forAll` operations follows the same schema: a *filter* step with the condition or its negation is generated and the number of results is analyzed using an *hasNext* step method call.

Finally, general expressions such as arithmetic and boolean operations, variable declarations, and literal values are simply mapped to their Groovy equivalent. Note that the transformation takes care of mapping OCL primitive types (such as String, Integer, Sequence, etc) into equivalent Groovy types when necessary.

5.2.5 Transformation process

Once the initial low-level mapping has been applied on the input OCL expression, the MOGWAÏ frameworks needs to combine the generated steps to create a complete Gremlin query. This operation is performed by analyzing the input OCL syntax tree and linking steps into traversal chains. To better illustrate this combination process, we introduce an example OCL query (Figure 5.3) and show how it is transformed into the final Gremlin expression shown in Figure 5.4 (abstract syntax tree) and Listing 10 (final textual expression). Listing 9 shows a simple query that selects the *Packages* instances which are not empty (i. e. does not contain any element through its *classes* association). Figure 5.3 shows an excerpt of the abstract syntax tree for this query. The top level element *Constraint* contains the context of the query, its return type and an *ExpressionInOCL* element representing the query itself. Its *body* contains the root expression of the query. Each expression in the OCL metamodel has a link to its source. In the example, the source chain starts with the *select* iterator, that has the *allInstances* operation as its source, which is linked to the *type* Package. *Iterators* are particular expressions that contains an iterator *variable* and a *body* representing the expression to apply on each element. As other expressions, the *body* tree also starts with the root operation in the expression, in our example the *isEmpty* operation inside the *select*.

As an initial step, the transformation pre-processes the OCL query model to first col-

Table 5.2 – OCL to Gremlin mapping

OCL expression	**Gremlin step**
Type	g.idx('classes')[[name:'Type']]
allInstances()	inE('instanceof').outV
collect(attribute)	attribute
attribute (implicit collection)	attribute
collect(reference)	outE('reference').inV
reference (implicit collection)	o.outE('reference').inV
oclIsTypeOf(C)	o.outE('instanceof').inV.transform(it.next() == C)
oclAsType(C).attribute	attribute
oclAsType(C).reference	outE('reference').inV
col$_1$ →union(col$_2$)	col$_1$.fill(var$_1$); col$_2$.fill(var$_2$); union(var$_1$, var$_2$);
col$_1$ →intersection(col$_2$)	col$_1$.fill(var$_1$); col$_2$.fill(var2); intersection(var$_1$, var$_2$);
col$_1 -$col$_2$ (Set subtraction)	col$_1$.fill(var$_1$); col$_2$.fill(var$_2$); subtract(var$_1$, var$_2$);
including(object)	toList().add(object)
excluding(object)	toList().removeAll(object)
includes(object)	toList().contains(object)
excludes(object)	!(toList().contains(object))
size()	count()
first()	first()
isEmpty()	toList().isEmpty()
select(condition)	c.filter{condition}
reject(condition)	c.filter{!(condition)}
exists(expression)	filter{condition}.hasNext()
forAll(expression)	!(filter{!condition}.hasNext())
$=, >, >=, <, <=, <>$	$==, >, >=, <, <=, !=$
$+, -, /, \%, *$	$+, -, /, \%, *$
and,or,not	&&,||,!
variable	variable
literals	*literals*

lect all the accessed *types* and creates variables storing the result of the corresponding index calls generated by the mapping in order to optimize index lookups when a type is accessed multiple times. In our example, *Package* type is transformed into an instruction that declares a variable (`packageV`) containing the corresponding vertex retrieved from the index. Besides this, *union*, *intersection*, and *set difference* operations are also collected to generate the intermediate variables used to store their results.

Once this initial processing has been performed, the transformation navigates in the OCL query model to find the first *step* of the traversal to build. This operation searches for the root expression in the OCL expressions' *source* chain and transforms it according to the mapping presented in Table 7.1. In our example this operation maps the root *Package* type to an access to the variable `packageV` (defined during the pre-processing phase). Then, the transformation navigates in the *source* containment tree in a postorder traversal and transforms each OCL operation into its Gremlin equivalent, and links it to the previously generated step using the *next* association. In our example, this processing generates the Gremlin nodes *inE('instanceof')* and *outV* corresponding to the *allInstance* expression.

```
sampleSelect : Set(Package) =
    Package.allInstances()
        →select(e | e.classes→isEmpty())
```

Listing 9 – Sample OCL Query

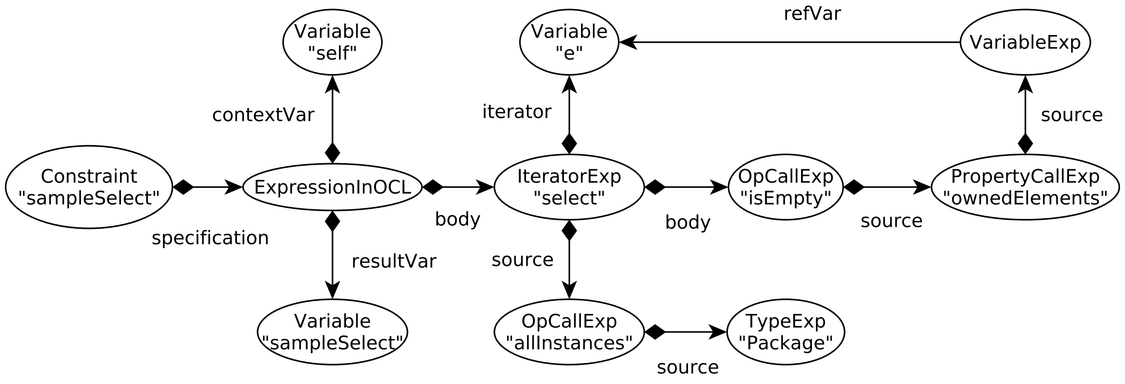

refVar
Variable "self"
Variable "e"
VariableExp
contextVar
iterator
source
Constraint "sampleSelect"
ExpressionInOCL
IteratorExp "select"
OpCallExp "isEmpty"
PropertyCallExp "ownedElements"
specification
body
body
source
resultVar
source
Variable "sampleSelect"
OpCallExp "allInstances"
TypeExp "Package"
source

Figure 5.3 – OCL Syntax Tree

OCL iterators have to be processed in a dedicated way because they define a *body* expression that is not part of the *source* containment tree. In our example the *select* iterator is transformed to a *filter* step containing a *closure* that represents its *body*. The *body* expression is parsed as a regular OCL expression by starting from the root element and generated *steps* are linked together. In Figure 5.4, *body* expression is mapped to *variableAccess*, *outE('classes')* and *inV*, *toList* and *isEmpty*, corresponding respectively to the iterator access, *collect(classes)*, and *isEmpty* OCL expressions. Note that if the OCL expression defines an explicit iterator variable (`e` in our example), a corresponding Gremlin variable declaration instruction is created in the closure with the same label. This variable contains the closure *it* value, that represents the current element processed. This dedicated variable is used to support iterator scoping in nested closures.

Finally, each expression in *union*, *intersection*, and *set subtraction* operations generates a single traversal that ends with a *fill* step that puts the results in the dedicated variable previously defined. The result of the OCL operation is computed from those variables by calling the additional Gremlin method we have defined to support collection operations. These intermediate accumulators are necessary because Gremlin branching mechanism requires that each sub-traversal starts with the same input, which is not always true for OCL collection operations.

Once the abstract syntax tree of the Gremlin traversal to compute has been created (Figure 5.4), it is sent to a model-to-text transformation that generates the final textual representation query (Listing 10). Finally, the MOGWAÏ framework delegates the query computation to the Blueprints database and reifies the results into model-level objects if necessary.

```
var packageV = g.idx("classes")[[name:Package]];
packageV.inE("instanceof").outV.filter{e=it;
  e.outE("classes").inV.toList().isEmpty()};
```

Listing 10 – Generated Gremlin Textual Traversal

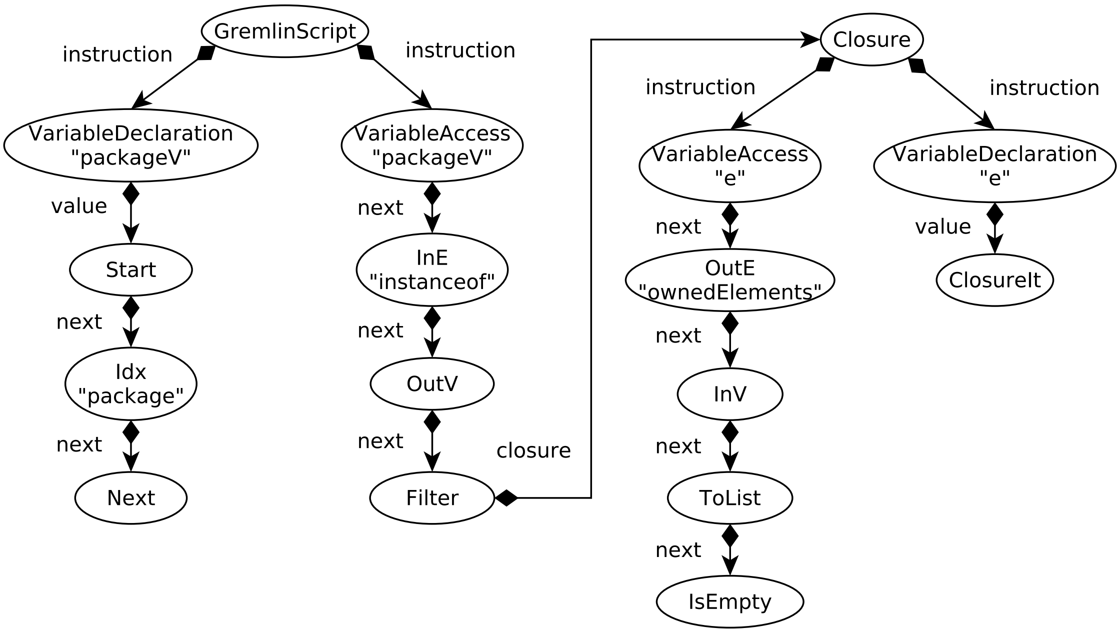

GremlinScript
instruction
instruction
VariableDeclaration "packageV"
VariableAccess "packageV"
value
next
Start
InE "instanceof"
next
next
Idx "package"
OutV
next
next
closure
Next
Filter
Closure
instruction
instruction
VariableAccess "e"
VariableDeclaration "e"
next
value
OutE "ownedElements"
ClosureIt
next
InV
next
ToList
next
IsEmpty

Figure 5.4 – Generated Gremlin Syntax Tree

5.3 Tooling

The MOGWAÏ framework is distributed as a set of open source Eclipse plugins under the EPL license [15]. The source code repository and the framework's documentation are publicly available on Github [16]. MOGWAÏ is integrated in the NEOEMF [35] environment, and provides its own implementation of the *PersistentResource* interface with dedicated methods to support query translation, execution, and result reification from Blueprints' persisted models.

Initial OCL queries are parsed using the Eclipse MDT OCL toolkit [17] and the output OCL models constitute the input of a set of 70 ATL [58] transformation rules and helpers implementing the mapping presented in Table 7.1 and the associated transformation process.

As an example, Listing 11 shows the transformation rule that generates a *filter* step from an OCL *select* operation. The *next* step is computed by the `getContainer` helper, which returns the parent of the element in the OCL *source* containment tree. The *instructions* of the *closure* are contained in an ordered set, to ensure the instruction defining the iterator variable is generated before the body instructions. Finally, the *select* body is generated, using the helper `getFirstInstruction` that returns the root element in an OCL *source* tree.

---

[15] https://www.eclipse.org/legal/epl-v10.html
[16] https://github.com/atlanmod/Mogwai
[17] www.eclipse.org/modeling/mdt/?project=ocl

```
rule select2filter {
    from
    s : OCL!IteratorExp (s.getOpName() = 'select')
    to
    f : Gremlin!FilterStep (
        closure ← cl,
        next ← select.getContainer()),
    cl : Gremlin!Closure(
        instructions ← OrderedSet{}
            .append(thisModule.var2def(select.iterator.first()))
            .append(select.body.getFirstInstruction()))
}
```

Listing 11 – Select to Filter ATL Transformation Rule

Once the Gremlin model is generated by the transformation, it is expressed using its textual concrete syntax and the resulting script is sent to an embedded Gremlin engine, which executes the traversal on the database and returns the result back to NeoEMF that reifies it to create a navigable EMF model. The reification process is done once the query has been entirely executed, and the constructed model only contains the result of the query, limiting the memory consumption implied by intermediate object reifications performed in API-based query solutions.

Finally, the framework also allows to define query parameters to check invariants, compute a value, or navigate a model from a particular model element. An additional binding mechanism is also provided to set the value of OCL variables (such as `self`) in order to evaluate queries on specific model element instances.

5.4 Evaluation

In this section, we evaluate the performance of the Mogwaï framework in terms of execution time and memory consumption and compare it with the state of the art solutions presented in Section 5.1. Note that we have discarded existing solutions that provide their own dedicated backend (such as CDO and MQT), because we could not envision a fair comparison scenario that would differentiate the generic data-store scalability improvements from the query solution itself. In addition, we have also discarded existing solutions that focus on database consistency (such as UML/OCL to SQL generative approaches) and does not primarily aim to compute queries efficiently.

The considered solutions (MDT OCL, EMF-IncQuery, and Mogwaï) are evaluated on top of models stored in a common persistence solution —NeoEMF/Graph — in order to provide a comparison that is not biased by the underlying data-store performances. EMF-based query solutions such as Eclipse MDT OCL and EMF-IncQuery manipulate the model through the standard EMF API provided by the persistence framework. Note that the evaluations of model data-store and persistence strategies have been covered in Chapter 3, and therefore are not presented in this query framework comparison.

5.4.1 Benchmark presentation

Our benchmark complements the one presented in Section 3.5 and reuses the input models and queries we designed to evaluate model persistence solutions. The queries have been implemented in the native frameworks' languages (OCL and EMF-IncQuery graph patterns), and retrieve:

- **ClassAttributes** computes the attributes of all the *Class* instances in the model
- **SingletonMethods** finds static methods returning their containing *Class* (singleton pattern)
- **InvisibleMethods** finds all the methods that have a private or protected modifier
- **UnusedMethods** computes the set of methods that are private and not internally called

Note that *SingletonMethods*, *InvisibleMethods*, and *UnusedMethods* have been implemented with an initial `allInstances` call, which is an important bottleneck for EMF API based query frameworks [116]. The *ClassAttributes* query has been implemented using a partial model navigation from the input *Model* element in order to compare the behavior of the query solution when only part of the model needs to be processed. Table 5.3 shows the number of intermediate objects traversed to compute each query and the size of their result sets to give an intuition on the complexity of the executed queries.

The experiments are run over the *set2* and *set3* models used in the benchmark presented in Section 3.5. All the queries are executed under two memory configurations: the first one uses a large virtual machine of 8 GB and the second a small-one of 250 MB. These configurations allow us to compare the different approaches both in normal and stressed memory conditions.

Table 5.3 – Query Intermediate Loaded Objects and Result Size per Model

	#Interm. MoDisco	**#Res. MoDisco**	**#Interm. JDT**	**#Res. JDT**
ClassAttributes	28 505	12 359	136 753	54 201
SingletonMethods	80 664	0	1 557 006	92
InvisibleMethods	80 664	134	1 557 006	3927
UnusedMethods	80 664	0	1 557 006	1155

5.4.2 Results

Experiment results for both benchmarked sets are listed in Tables 5.4 and 5.5. Each table is divided into two parts, showing respectively the time in seconds to perform the queries, and the memory consumption implied by the query computation. Table cells present the results for both the large and small JVM configurations.

5.4.3 Discussion

Experiment results show that the MOGWAÏ framework outperforms all the query frameworks executed over NEOEMF/GRAPH both in terms of memory consumption and

Query	**Execution Time** (s)			**Mem. Consumption** (MB)		
	OCL	**IncQuery**	**Mogwaï**	**OCL**	**IncQuery**	**Mogwaï**
ClassAttributes	6/6	13/14	5/5	9/11	49/48	12/12
SingletonMethods	11/11	14/14	5/4	21/24	57/54	19/18
InvisibleMethods	12/11	14/15	4/4	25/26	62/63	18/19
UnusedMethods	12/12	14/14	4/4	22/26	55/54	17/18

Table 5.4 – Query Results on Set 2

Query	**Execution Time** (s)			**Mem. Consumption** (MB)		
	OCL	**IncQuery**	**Mogwaï**	**OCL**	**IncQuery**	**Mogwaï**
ClassAttributes	18/18	241/635	10/10	41/43	606/161	31/32
SingletonMethods	136/174	238/1068	8/7	392/143	614/197	47/48
InvisibleMethods	141/175	230/606	8/8	392/143	614/197	47/48
UnusedMethods	136/182	221/432	6/6	395/111	566/165	45/51

Table 5.5 – Query Results on Set 3

execution time. The difference in terms of execution time is up to 20 times better than the Eclipse MDT OCL framework, and up to 8 times better in terms of memory consumption. This difference is reduced for small models and simple queries (such as *ClassAttributes*), where the number of intermediate objects loaded from the data-store is not significant.

Comparing the presented results with the ones obtained by computing the same queries over multiple persistence solutions (Section 3.5) shows that the MOGWAÏ framework is faster and consumes less memory than NEOEMF/MAP —the best solution to evaluate EMF API based queries— when the query implies a lot of intermediate object creation and the targeted model is large. This improvement is explained by (i) the absence of intermediate objects creation that consume time and memory and (ii) the use of indexes and query optimizations on the database side, avoiding a complete traversal of the model elements.

Conversely, if the query traverses a small subset of the model and has an important result set (such as the *ClassAttributes* query), the benefits of using MOGWAÏ are reduced compared to existing solutions. The result of the first query confirms this observation, where an important part of the intermediate elements are part of the final result set. The memory consumption may be even more important than other approaches for small models because the framework consumes memory to instantiate the transformation engine that handle the OCL to Gremlin translation.

Note that the comparison only considers a single execution of each query over the models. In case where the query is executed several times over a slightly different version of the same model, an incremental approach like the one provided by EMF-IncQuery could complement our approach, for example by using the MOGWAÏ framework to perform the initialization queries of the incremental engine, which are an important bottleneck in terms of memory consumption and execution time when applied to a *lazy-loading* persistence framework.

To summarize these results, the MOGWAÏ approach is an interesting solution to perform complex queries over large models. Using the query translation approach, gains in terms of execution time and memory consumption are positive, but the results also show that our approach is not the best solution for all kinds of queries. For example, it is more interesting to use API based queries on NEOEMF/MAP if the model is relatively small and/or if the query does not traverse an important part of the model.

The main disadvantage of the MOGWAÏ framework concerns its integration to the standard EMF environment. While persistence frameworks can be plugged transparently to EMF-based applications to improve their scalability, our solution requires to update the application code to translate EMF API calls into OCL expressions. However, we believe that this trade-off can be interesting for critical queries that need to be computed efficiently. In addition, we designed our query API to transparently reify results into EMF compatible objects, reducing the cost of integrating the MOGWAÏ framework in existing applications.

5.5 Conclusion

In this chapter we presented MOGWAÏ , an efficient query framework that translates model queries expressed in OCL into Gremlin expressions that can be directly computed by a Blueprints-compatible data-store. Our solution relies on a model-to-model transformation that maps OCL operations into Gremlin steps, and compose them to create a single query that can be optimized by the underlying database. Our experiments show that the MOGWAÏ query solution outperforms alternative approaches in terms of memory consumption and execution time to perform complex queries over large models.

MOGWAÏ addresses all the requirements introduced in 5.1: our solution generates Gremlin traversals —a NoSQL query languages— that manipulate database internal structures such as indices to improve computation performances (**Rq1**). The MOGWAÏ API accepts queries expressed in the OCL language, the OMG standard to define model invariants and queries (**Rq2**), and our evaluation shows that our query translation approach outperforms existing EMF API-based solutions as well as other query solutions (**Rq3**). Finally, the architecture of the framework makes the low-level Gremlin query execution totally transparent to the end user, because of the reification feature that convert returned database records into model-level elements that can be manipulated using the standard modeling APIs.

In the following we show how the MOGWAÏ approach can be extended to improve the performance of computing complex model transformations on top of large models stored in NoSQL data-stores.

6

Efficient Transformations

Model transformations have been characterized by Sendall and Kozaczynski as *the heart and soul of Model Driven Development* [97]. Indeed, model transformations are one of the core concept of most MDE processes: they are used to refine models, extract view of complex systems that can be understood by a modeler, generate code, or perform formal verification on an input model [58]. For exemple, model transformations constitute the operational part of the OMG's MDA development methodology [80], that promotes the use of a high-level Platform Independent Model (PIM) representing the specification of a system that is refined using model transformations into several Platform Specific Model (PSM) used to generate the final software artifacts. In the field of MDRE, model transformations are vastly used to detect problematic code, apply refactoring operations on existing applications, or generate documentation [19] from an existing code base.

As we stated in the previous chapters, the progressive adoption of MDE techniques in the industry [55, 117], coupled with the growing accessibility of big data (such as national open data programs [53]) has led to a situation where the volume and diversity of data to model has grown to such an extent that the scalability of existing technical solutions to store and manipulate models has become a major issue [68]. This is particularly true in the context of model transformations, that often define global processes that are applied on an entire (and potentially large) model.

Model transformation tools suffers from the same issues as query frameworks presented in the previous chapter: they rely on the low-level model handling API provided by the modeling framework that are focused on manipulating individual model elements and do not offer support for generic queries and transformation operations. As a result, model transformation tools are not efficient with current model persistence framework based on *lazy-loading* mechanisms, because (i) the granularity of the API is too fine and is not able to express complex model transformation patterns efficiently on the database side, and (ii) an important time and memory overhead is necessary to construct navigable objects that can be manipulated by the modeling framework. As shown in Figure 6.1, this is particularly true in the context of model transformations, which heavily rely on high-level model navigation queries (such as the `allInstances()` operation returning all instances of a given type) to retrieve source elements to transform and create the corresponding target model. This mismatch between high-level modeling languages and low-level model access APIs generates a lot of fragmented queries that cannot be optimized and computed efficiently by the database[116].

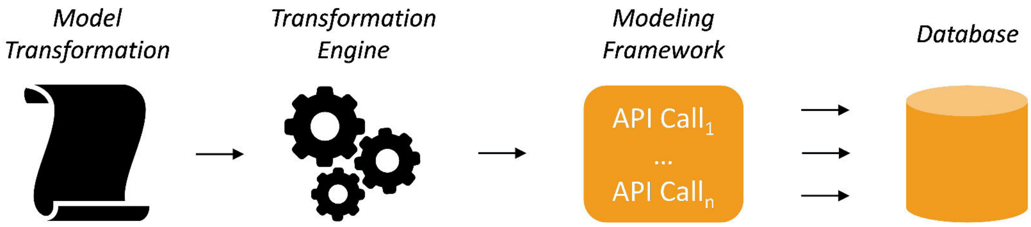


Figure 6.1 – Model Transformation Engine and Modeling Framework Integration

Based on our experience on using NoSQL data-stores to persist and query large models (emphasized by the approaches presented in Chapter 3 and 5), we have defined five requirements that need to be addressed in order to provide an efficient transformation solution for the novel generation of NoSQL model persistence frameworks. Note that these requirements are similar to the ones presented in Chapter 5, because model transformation are intrinsically related to model query computation, and they suffer from the same alignment issue between high-level model operations and low-level modeling APIs:

Rq1 the solution should generate database queries in order to benefit from the database structures and optimizations, and limit network overhead

Rq2 the framework should support transformation expressed using a state of the art model query language such as the ATL or the QVT standard

Rq3 the framework must outperform existing transformation solutions relying on low-level model handling APIs

Rq4 the framework should be extensible to support new persistence solutions and model mappings

Rq5 the transformation computation should be customizable to fit the modeler constraints in terms of execution time and memory consumption

In this chapter, we propose GREMLIN-ATL , that aims to address these issues by providing a novel transformation framework to compute complex model transformations on top of actual persistence frameworks efficiently. GREMLIN-ATL is based on a translation from high-level model transformation specification into efficient database queries that are directly executed on the underlying persistence framework's database. Our approach is generic and aims to target multiple data representation, using a flexible architecture that allows to define mappings that describe the alignment between high-level, transformation specific constructs and existing database APIs. Our experiments show that GREMLIN-ATL can bring a significant execution time and memory improvement when applied to large and complex models.

The rest of this chapter is structured as follow: Section 6.1 introduces the state of the art of model transformation tools, presenting their benefit and drawbacks when moving to large models. Section 6.2 presents GREMLIN-ATL and its key components, Section 6.4

presents how a transformation is executed from a user point of view. Sections 6.5 and 6.6 present our prototype and the benchmarks used to evaluate our solution. Finally, Section 6.7 summarizes the key points of the paper and draws conclusion.

6.1 State of the Art

Model transformation languages and engines have been an active field of research for the last 15 years, and several solutions have been proposed to define, evaluate, and execute model transformation efficiently. Existing approaches usually provide a DSL that lets modelers define their transformation using a high-level language that is used to access a source model, express transformation conditions, and create the target elements. In this section we first review the most well-known model transformation approach nowadays, then we detail how these solutions have been extended and complemented with enhanced support for large models and complex transformation computations. Finally, we summarize the benefits and issues of the existing solutions and propose a research problem to address in order to improve the computation of model transformations over large models.

6.1.1 Model Transformation frameworks

Originally, model transformations were expressed using general purpose programming languages (such as Java or XTend [12] for the EMF ecosystem) manipulating model elements through the modeling framework APIs. While these low-level approaches allow to precisely define the transformation execution, the lake of abstraction and language construct to define transformation operations make them hard to reuse and maintain. Multiple solutions have been proposed to tackle these issues by providing high-level DSLs focused on transformation definition and hiding execution details from the modeler.

ATL [58] is a declarative, rule-based language that defines transformations over an arbitrary number of source and target metamodels. An ATL transformation is composed of a set of *transformation rules* that map source elements from an input model according to a given condition and produce *target* elements stored in an output model. The ATL framework is integrated in the Eclipse ecosystem, and provides native support to EMF models. However, the current implementations rely on the low-level model handling APIs to navigate the source and the target models, and keeps them in memory along with transformation specific informations, limiting the application performances in modeling scenarios involving large models and scalable persistence frameworks.

The QVT standard [85] is an OMG specification that defines the architecture, the language, and the operational mapping of the QVT transformation language. It has been implemented in multiple frameworks, such as QVTo and QVTd, that respectively provide an imperative and a declarative implementation of the language. The internal transformation algorithm embedded in existing QVT engines is designed to be efficient in terms of memory consumption and execution time, and recent work [118] reported that QVT scales better than ATL when moving to large models. However, in its core QVT relies on the same low-level modeling APIs as ATL to access and manipulate models, reducing its performances on top of existing scalable persistence solutions.

The Epsilon Transformation Language (ETL) [67] is a transformation language built on top of the Epsilon platform that aims to provide enhanced integration with languages that support additional modeling tasks, such as model validation [65] or merging [64]. ETL has been designed as a hybrid language that implements a task-specific rule definition and execution scheme, but which also inherits the imperative features of EOL to handle complex transformations when necessary. ETL enables specification of transformations involving an arbitrary number of source and target models. Compared to ATL and QVT, ETL is based on the Epsilon infrastructure, that provides enhanced support for high-level modeling operations compared to EMF. However, the Epsilon API is not dedicated to model transformation computation, and also suffers of the low-level definition of its model handling API, that generates a lot of fragmented queries and intermediate object creations that have a significant impact on the memory consumption when moving to larger models.

Finally, The Viatra project [31] is an event-driven and reactive model transformation platform that relies on an incremental pattern matching language to access and transform models. The framework benefits of the incremental feature of EMF-IncQuery [11] to efficiently recompute model navigation and update transformed models. Viatra receives model update notifications and incrementally re-compute queries and transformations in an efficient way at the cost of a higher memory consumption. Compared to other approaches, Viatra proposes a different solution that only requires to use low-level modeling APIs to initialize its internal structure: it performs a traversal of the source model to transform and initialize its caches with the objects that have to be monitored and incrementally updated. These internal structures are then manipulated to compute model transformations, decoupling the computation from the low-level modeling APIs. However, the framework presents two issues that limits its performance in the context of large models: (i) the internal structures providing the incremental feature of the engines are stored in the memory, and are not designed to scale to very large models, and (ii) while the transformation computation is not tailored to a specific modeling APIs, the cost of the initial traversal used to initialize the Viatra structures can be an important bottleneck, especially in *lazy-loading* persistence solutions.

6.1.2 Towards Scalable Model Transformations

While the presented solutions have been used for years in MDE processes, several studies reported that the scalability of MDE tools (and especially model transformation engines) is an important factor that hampers the adoption of MDE techniques in industrial processes involving large and complex models [54, 55].

Several solutions have proposed to parallelize and distribute model transformations to tackle this issue and improve the efficiency and scalability of existing transformation engines. ATL-MR [7] is a map-reduce based implementation of the ATL engine that computes transformations on top of models stored in a distributed cluster running HBase[1]. The approach is based on a data partitioning strategy that assign each model element to a node in the cluster in order to maximize data locality and reduce network communication overhead [8]. The tool benefits from the distributed nature of the data-store and its integration in the Map/Reduce ecosystem to distribute the computation, improving the overall execution time.

---

[1] `https://hbase.apache.org/`

Query Solution	API-based	Translation	In-memory	In database	
				Relational	NoSQL
Java / XTend	✅	❌	✅	☑	☑
ATL	✅	❌	✅	☑	☑
QVT	✅	❌	✅	☑	☑
ETL	✅	❌	✅	☑	☑
Viatra	✅	❌	✅	❌	❌
Parallel-ATL	✅	❌	✅	☑	☑
ATL-MR	✅	❌	✅	❌	✅
Lintra	❌	❌	✅	❌	✅

Table 6.1 – Features of existing transformation solutions

Parallel-ATL [113] is an alternative implementation of the ATL engine that enables parallel computation of ATL rules. Compared to ATL-MR, Parallel-ATL is based on a task distribution algorithm that takes benefit from the multicore environment provided by current machines by inspecting the input transformation and splitting the execution into several workers that access a global shared model asynchronously. The framework relies on the declarative aspect of ATL to compute transformation rules in parallel, and as shown positive results compared to the standard ATL execution engine.

The LinTra [23] platform is another solution that relies on the Linda coordination model to parallelize and distribute model transformations. Compared to Parallel-ATL and ATL-MR, the framework does not rely on the low-level modeling framework API to compute the transformation, but provides an intermediate access API that enable support of various NoSQL data-stores, such as Oracle Coherence[2] or Infinispan[3]. While this flexible approach allows to compute transformation on multiple data-sources, it assumes that the input models are provided in a Lintra-compatible format, and does not specify how to build these models from state of the art modeling languages such as UML or Ecore. In addition, the transformation itself is expressed using a Java internal DSL, that can be hard to reuse and modularize, and requires some knowledge on the underlying coordination model to fully benefit from the approach.

Table 6.1 summarizes the main features of the state of the art solutions listed above. We classify them using the same four criterias as the one presented in Chapter 5: (i) if the solution is based on a modeling API (for example EMF) to access the underlying model or if (ii) it is based on a translational approach that generates low-level database queries from high-level transformation languages. We also distinguish if (iii) the approach computes the transformation in memory or (iv) if it delegates the computation to the database layer. For each presented tool, fully supported features are represented with green checks, partially supported features are shown as grey checks, and unsupported features are represented as red crosses.

---

[2] `http://www.oracle.com/technetwork/middleware/coherence/overview/index.html`

[3] `http://infinispan.org`

6.1.3 Summary and Research Problem

The vast majority of existing solutions rely on the usage of modeling APIs to compute transformations and directly suffers of the alignment mismatch between transformation primitive and model persistence frameworks. Furthermore, existing solutions usually use an in-memory transformation engine that keeps in memory part of the manipulated model along with transformation specific constructs, and are not designed to scale to large models.

In addition, there is no solution that currently fulfills all the requirements listed above, especially *Rq1*. Based on these observations, we state that **there is currently a need to provide a translation approach that handles model transformations on top of large models efficiently by generating optimized NoSQL database scripts**.

In the following we present GREMLIN-ATL , our scalable transformation engine that aims to address these requirements by generating low-level database scripts from high-level model transformations expressed in ATL.

6.2 Gremlin-ATL Framework

The approach we propose relies on a two step process: (i) a model-to-model transformation that takes as its input an ATL transformation and generates the corresponding Gremlin traversals (Section 6.3), and (ii) an execution envinronment that allows to plug GREMLIN-ATL on top of various data-store and tune the transformation execution (Section 6.4). Once the transformation has been executed, the resulting model can be translated back to the modeling framework level, allowing client applications to manipulate it transparently.

In this section we first introduce and motivate the choice of ATL as our input language, then we present an overview of the framework and its query translation process. Note that the motivations for using Gremlin as our target language are similar to the ones presented in Section 5.2.1.

6.2.1 The ATL Transformation Language

Before introducing the internal components of our approach, we detail the ATL [58] language, that is used to define the transformation to compute with the GREMLIN-ATL framework. The choice of using ATL as our input language is motivated by the fact that it is a wide-spread technology in the EMF ecosystem used to express and compute model transformations, as emphasized by the various frameworks based on the ATL engine [7, 113], as well as the growing amount of transformations referenced in the ATL Zoo[4]. Note that while this chapter focuses on ATL, our solution could be adapted to other rule-based transformation languages, notably by reusing the work proposed by Jouault and Kurtev [59] on the architectural alignment of ATL and the QVT standard [85].

ATL is a declarative, rule-based model transformation language that lets modelers express *transformation rules* defined at the metamodel level. An ATL transformation is defined between an arbitrary number of *source* and *target* metamodels, and contains a set of rules matching source elements and creating the corresponding target model. Transformations are organized in *modules*, that represent semantic containers used to group related rules and define libraries. ATL provides three types of rules: (i) *matched rules* that are declarative and automatically executed by the engine, (ii) *lazy rules* that are also declarative but are explicitly invoked from another rule, and (iii) *called rules* which contain imperative code [5] used to provide additional behavior such as logging execution traces, code generation, or specific element modifications.

---

[4] `https://www.eclipse.org/atl/atlTransformations/`

The language embeds its own implementation of the OCL standard [83] as its language to express transformation rule conditions, source and target model navigations, and attribute/reference bindings in the target model. These OCL expressions can be modularized in *helpers*, that are reused along the transformation to compute information in a specific context, provide global functions, and runtime attributes computed on the fly. OCL helpers can be invoked multiple times in a transformation, and related helpers can be grouped in library *modules* that can be shared between multiple transformations.

Finally, ATL programs are themselves described as models that conform to the ATL metamodel representing the grammar of the language. This feature allows to define higher-order transformations [112], that take an ATL transformation as their input and manipulate it to check invariant properties that should hold during the transformation [28], verify the transformation by dynamically inferring and checking types [32], or refine it into another language or technology [111]. GREMLIN-ATL relies on this higher-order transformation capabilities to refine existing ATL transformations and produce efficient database queries.

Note that the semantic of the language does not assume any order in the matched rule executions, meaning that a target model can contain proxy values that will be resolved later in the process. Current implementations of the ATL engine rely on a *trace link* mechanism to provide this feature by keeping traces between the source and target model elements in order to resolve proxies and set target values that have not been transformed yet. Tisi et al. [113] have shown that this independence between rules can be used, for example, to efficiently parallelize ATL transformations.

To better illustrate the ATL concepts that are used in this chapter, we introduce a simple transformation example that will be used through the following sections to explain the GREMLIN-ATL approach. Our example transformation is based on the metamodel presented in this manuscript's running example (Chapter 2), as a remainder, Figure 6.2 presents our simple metamodel representing Java programs in terms of a hierarchy of *Packages*, *Classes*, *Methods*, *Constructors* (a subclass of methods), and *Types* that are returned by *Methods*. A simple instance of this metamodel is provided in Figure 2.3.

Listing 12 shows a simple ATL transformation defining the module *example* (line 1) that takes as its input an instance of the Java metamodel shown in Figure 6.2, and creates an output model conforming to the Knowledge Discovery Model (KDM) metamodel [81], an OMG standard that represents software artifacts independently of their platform (line 3). The *example* module contains a single transformation rule *MethodToMethodUnit* (line 15) that matches all the *Method* elements from the input model that does not contain the *Test* string in their name and creates the corresponding *MethodUnit* in the target model. The attributes and references of the created elements are set using *binding specifications*:

---

[5] Note that imperative constructs are not discussed in this chapter

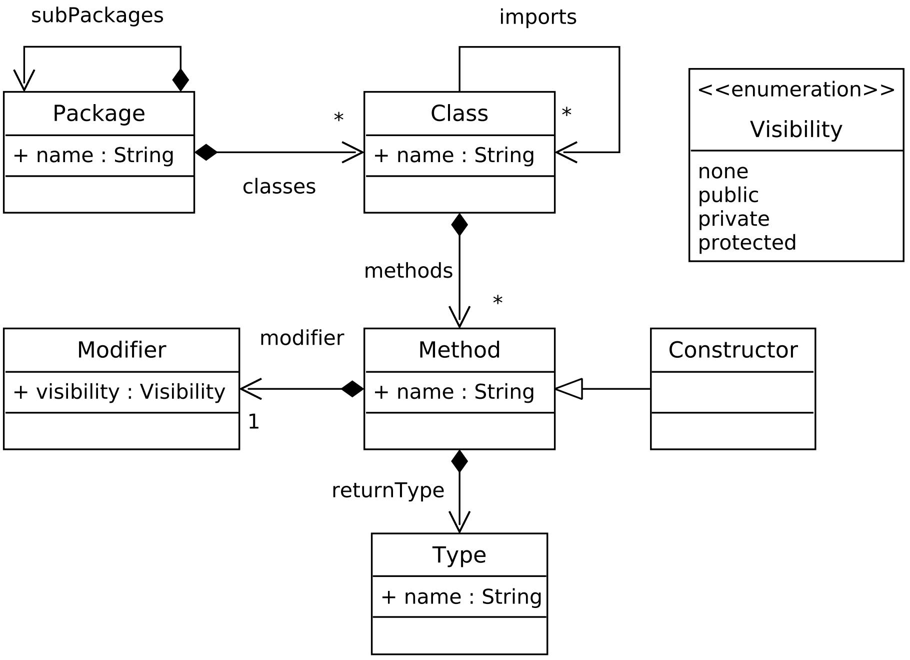

subPackages
imports
Package
+ name : String
*
Class
+ name : String
*
classes
<<enumeration>>
Visibility
none
public
private
protected
methods
*
Modifier
+ visibility : Visibility
1
modifier
Method
+ name : String
Constructor
returnType
Type
+ name : String

Figure 6.2 – A Simple Java Metamodel

the first one (line 18) contains an OCL expression that checks if the source element is a *Method* or a *Constructor* and sets the KDM *kind* attribute accordingly. The second binding (line 20) sets the value of the *export* attribute by calling the OCL helper *getVisibility* on the source element. Finally, the *type* reference is set with the *Type* element contained in the *returnType* reference of the source *Method*. Note that an additional rule has to be specified to map *Type* instances to their corresponding output elements, that will be resolved by the ATL engine using its *trace links* mechanism.

```
 1  module example;
 2
 3  create OUT: KDM from IN: Java;
 4
 5  -- returns a String representing the visibility of a Method
 6  helper context Java!Method def : getVisibility() : String =
 7    let result : VisibilityKind = self.visibility in
 8      if result.oclIsUndefined() then
 9        "unknown"
10      else
11        result.toString()
12      endif;
13
14  -- Transforms a Java Method into a KDM Method unit
15  rule MethodToMethodUnit {
16    from src : Java!Method(not src.name.contains('Test'))
17      to tgt : KDM!MethodUnit(
18      kind ← if (src.oclIsKindOf(java!Constructor))
19                then 'constructor' else 'method' endif,
20      export ← src.getVisibility(),
21      type ← src.returnType
22  }
```

Listing 12 – Simplified Java2KDM Rule Example

6.2.2 Framework Overview

Figure 6.3 presents an overview of the GREMLIN-ATL framework that creates and computes *Gremlin Traversals* from *ATL Transformations*. An input transformation is parsed into an *ATL Transformation Model* conforming to the *ATL metamodel*. This model constitutes the input of our *ATLtoGremlin* high-order transformation that creates an output *Gremlin Traversal* representing the query to compute (details on the traversal construction are provided in the following sections) and sends it to the database for execution. Note that the generated Gremlin traversal conforms to the Gremlin metamodel defined in Section 5.2.

The *ATLtoGremlin* transformation uses two generic libraries to produce the output query: (i) a *Model Mapping Definition* providing an abstraction layer that decouples the transformation computation from the low-level database access, allowing to access several data-sources as a model by mapping its implicit schema to modeling primitives, and (ii) a *Transformation Helper Definition* defining an API to redefine transformation-specific operations, enabling to tune the transformation algorithm according to memory and execution time requirements [6]. Note that this generic approach is an extension of the one presented in Chapter 5 that only focuses on generating Blueprints-compatible queries.

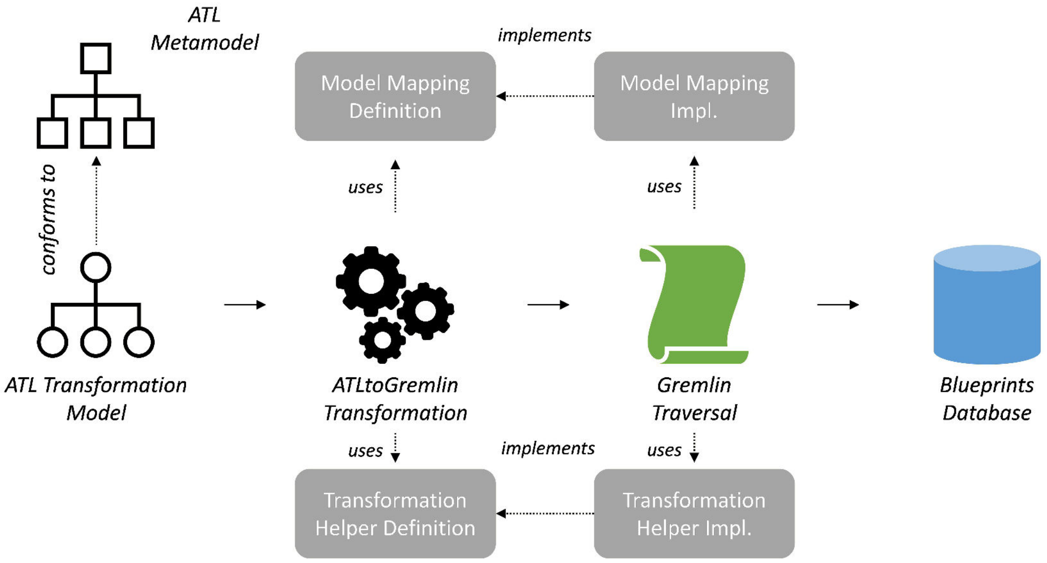

ATL Metamodel
implements
Model Mapping Definition
Model Mapping Impl.
conforms to
uses
uses
ATL Transformation Model
ATLtoGremlin Transformation
Gremlin Traversal
Blueprints Database
uses
implements
uses
Transformation Helper Definition
Transformation Helper Impl.

Figure 6.3 – Overview of the GREMLIN-ATL Framework

The generated traversal can be returned to the modeler and used as stored procedures to execute in the future, or directly computed in a Gremlin engine that uses specific implementations of the *Model Mapping* and *Transformation Helpers* libraries to access the underlying data-store and operates the transformation. In addition, GREMLIN-ATL also provides support to compute directly the generated query in a preset NEOEMF database and generates EMF compatible target models. This advanced integration is detailed in Section 6.5.

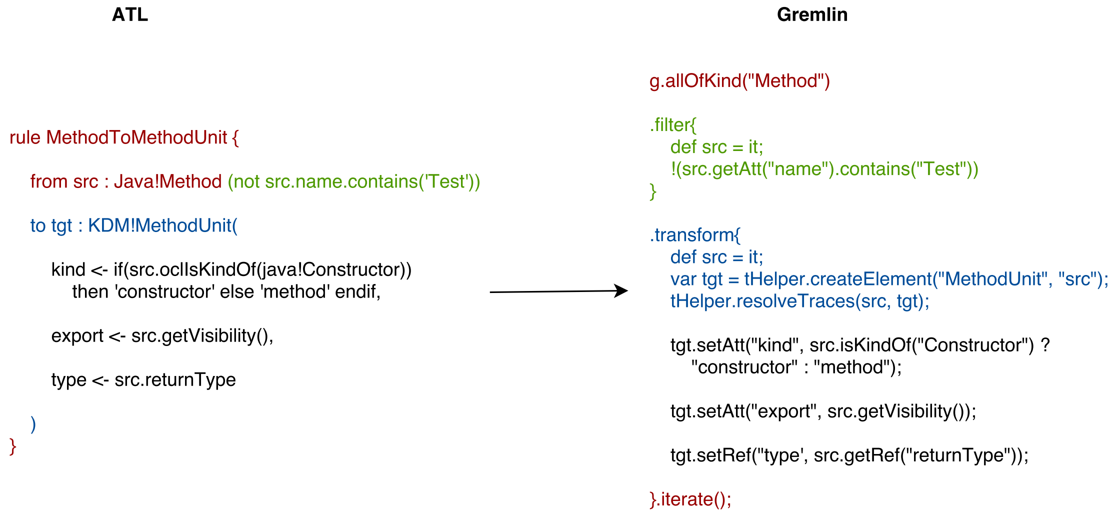
In the following we detail the *ATLtoGremlin* transformation, and we show how the generated Gremlin traversal interacts with the *Model Mapping* and *Transformation Helper Definitions* to support multiple model storage solutions and allow fine-grained customization of the transformation process. A complementary user point of view is provided in Section 6.4. Note that details on the Gremlin language, as well as our motivations to select it as our target database language has been provided in Section 5.2.1.

---

[6] Details on these libraries are provided in Section 6.4

6.3 ATLtoGremlin Transformation

Figure 6.4 presents an overview of our *ATLtoGremlin* transformation process that generates Gremlin traversals from ATL transformations. An ATL transformation rule is composed of four elements: (i) a *rule definition* containing the name of the rule and the matched type in the metamodel (red), (ii) a *guard* representing the condition to check against matched elements (green), (iii) an *out pattern* representing the element to create (blue), and (iv) a set of *binding* operations that set the attributes and references of the created element. The Gremlin script generated by our approach is constructed by mapping these individual constructs into their Gremlin equivalent, and assembles them to respect the transformation semantic. In the following we present the details of our mapping and we show how the generated query fragments are combined to create the final Gremlin traversal to execute.



```
rule MethodToMethodUnit {

   from src : Java!Method (not src.name.contains('Test'))

   to tgt : KDM!MethodUnit(

      kind <- if(src.oclIsKindOf(java!Constructor))
         then 'constructor' else 'method' endif,

      export <- src.getVisibility(),

      type <- src.returnType

   )
}
```

```
g.allOfKind("Method")

.filter{
   def src = it;
   !(src.getAtt("name").contains("Test"))
}

.transform{
   def src = it;
   var tgt = tHelper.createElement("MethodUnit", "src");
   tHelper.resolveTraces(src, tgt);

   tgt.setAtt("kind", src.isKindOf("Constructor") ?
      "constructor" : "method");

   tgt.setAtt("export", src.getVisibility());

   tgt.setRef("type', src.getRef("returnType"));

}.iterate();
```

Figure 6.4 – ATLtoGremlin Transformation Overview

6.3.1 ATL Operations Mapping

Table 6.2 shows the mapping used by GREMLIN-ATL to translate individual ATL constructs into Gremlin steps. An ATL *module*, that represents the top-level container holding the transformation rules, is translated into an equivalent Gremlin script that contains all the instructions of the traversal to execute.

Matched Rules Definitions contained in the ATL module are mapped to a sequence of steps that access all the elements of the type matched by the rule. For example, the matched rule definition *MethodToMethodUnit* in Listing 12 is translated into the Gremlin expression `g.allOfKind("Method")` that searches in the input data-store all the elements representing *Method* instances. *Abstract Rule Definitions* are not translated, because they are called only when a specialized rule is matched. *Lazy rule definitions* are translated into function definitions named with the rule's identifier and contains their translated body.

Matched rule bodies are mapped to a *transform* step containing the translated expressions representing rule's out pattern and bindings. This transform step is followed by an *iterate* step that tells the Gremlin engine to execute the query for each source elements. *Abstract rule bodies* are directly mapped without creating a transform step, and generated Gremlin steps are added to the ones of the translated bodies of the corresponding *specialized rules*. This approach flattens the inheritance hierarchy of a transformation by duplicating parent code in each concrete sub-rule.

Rule Guards defining the set of elements matched by a rule are translated into a Gremlin filter step containing the translated condition to verify. For example, the guard of the rule *MethodToMethodUnit* is translated into the following Gremlin expression that first navigates the *name* attribute and checks if it contains the *Test* string: `filter {!(src.getAtt("name").contains("Test")}`.

Rules' body expressions contain two types of ATL constructs: *out patterns* representing the target element to create, and *attribute/reference bindings* describing the attribute and references to set on the created element. *Out patterns* are mapped to a variable definition storing the result of the *createElement* function which creates the new target instance and the associated source to target trace links. This instruction is followed by a *resolveTraces* call that tells the engine to resolve the potential proxies associated to the created element using the trace links mechanism. In our example, this step generates the sequence `tHelper.createElement("MethodUnit", src)` that creates a new *MethodUnit* instance in the target model and associates it with the *src* element from the source model. *Attribute* and *Reference Bindings* are respectively translated into the mapping operation *setAtt* and a transformation helper's *link* call[7].

Our mapping translates *helper definitions* into global methods, which define a *self* parameter representing the context of the helper and a list of optional parameters. This global function is dynamically added to the *Object* metaclass to allow method-like invocation, improving query readability. *Global helper definitions* are also mapped to global methods, but do not define a *self* parameter. Finally, *Global Variables* are translated into unscoped Gremlin variables that can be accessed in every instruction.

ATL embeds its own implementation of OCL, which is used to navigate the source elements to find the objects to transform, express the guard condition of the transformation rules, and define helpers' body. We have adapted the mapping defined in the MOGWAÏ framework (see Chapter 5) to fit the OCL metamodel embedded in ATL. In addition, we integrated our *Model Mapping Definition* component in the translation in order to provide a generic translation based on an explicit mapping. The current version of GREMLIN-ATL supports an important part of the OCL constructs, allowing to express complex navigation queries over models. As an example, the inline *if* construct used in *MethodToMethodUnit* to check whether the source element represents a constructor can be translated into the equivalent Gremlin ternary operator: `src.isKindOf("Constructor") ? "constructor" : "method"` that uses the *isKindOf Model Mapping* operation to perform the type conformance checking.

---

[7] Reference bindings are handled by the Transformation Helper component to enable proxy element creation and trace links management

Table 6.2 – ATL to Gremlin mapping

ATL expression	Gremlin step
module	Gremlin Script
matched_rule definition	g.allOfType(type)
abstract_rule definition	not mapped
lazy_rule definition	def name(type) { body }
matched_rule body	transform{ body }.iterate()
abstract_rule body	body [8]
specialized_rule body	transform{ $body \cup parent.body$ }.iterate()
rule_guard(condition)	filter{condition}
out_pattern(srcEl, tgtEl)	var out = thelper .createElement(tgtEl.type, srcEl) tHelper.resolveTraces(srcEl,tgtEl);
attribute_binding	e1.setAtt(exp)
reference_binding	e1.link(exp)
helper_definition	def name(var self, params){ expression } Object.metaClass.name = { (params) $\rightarrow$ name(delegate, params)}
obj.helper(params)	obj.helper(params)
global_helper_definition	def name(params) { expression }
global_helper_computation	name(params);
global_variable	def name = expression
OCL_Expression	Mogwaï [9]

6.3.2 Operation Composition

Once the individual ATL constructs have been mapped to their corresponding Gremlin steps using the presented mapping, our framework composes the generated query fragments to create the complete traversal to execute. This section details the GREMLIN-ATL composition process by reusing the running example's transformation presented in Listing 12 and shows how the generated steps are assembled to create the final Gremlin output presented in Listing 13.

In order to generate a complete Gremlin query, our transformation has to navigate the input ATL model and link the generated elements together. First, the *ATLtoGremlin* transformation searches all the *helper* definitions (including global ones) and translates them according to the mapping shown in Table 6.2. The generated functions are added to the Gremlin script container, making them visible for the translated ATL rules. This first step generates the function definition and its body expression shown in lines 1 to 8 corresponding to the *getVisibility* helper presented in Listing 12. Note that this initial phase also generates the function that registers contextual helpers to the *Object* metaclass (lines 10-13), allowing method-like invocation of the function in generated expressions.

---

[8] The body of abstract rules is duplicated in the transform step of all its sub-rules

[9] OCL expression are translated by an improved version of the Mogwaï framework presented in Chapter 5

Lazy_rule definitions are then translated into global functions, and added to the Gremlin script container. Their *out pattern* and *binding* expressions are translated into their Gremlin equivalent following the mapping presented in the previous section and appended in the generated function body. This pre-processing generates the Gremlin constructs that corresponds to the ATL expressions that are explicitly called during the transformation, and ensures that all these auxiliary functions are defined when computing the traversal parts corresponding to the *matched rules*.

```
1  // getVisibility() helper
2  def getVisibility(var self) {
3    var result = self.getAtt("visibility");
4    if(result == null)
5      return "unknown";
6    else
7      return result;
8  }
9
10 // Add getVisibility to Vertex method list
11 Vertex.metaClass.getVisibility =
12   { → getVisibility(delegate) }
13
14 // MethodToMethodUnit
15 g.allOfKind("Method").filter{
16   def src = it;
17   !(src.getAtt("name").contains("Test"))
18 }.transform{
19   def src = it;
20   var tgt = tHelper.createElement("MethodUnit", src);
21   tHelper.resolveTraces(src, tgt);
22   tgt.setAtt("kind", src.isKindOf("Constructor") ? "constructor" : "
        method");
23   tgt.setAtt("export", src.getVisibility());
24   tgt.setRef("type", src.getRef("returnType"));
25 }.iterate();
```

Listing 13 – Generated Gremlin Traversal

Once this initial step has been performed the transformation searches all the *matched rules definitions* and creates the associated Gremlin instructions. If the rule defines a guard the generated *filter* step is directly linked after the *allOfKind* operation in order to select the elements that satisfy the guard's condition. Then, the *transform step* and the associated *iterate* call corresponding to the *matched rule body* are created and added at the end of the traversal. In our example this operation generates lines 15 to 18.

The *Out pattern* elements contained in the *matched rule body* are retrieved and the corresponding Gremlin instructions (lines 20 and 21) are added to the *transform* step closure. Finally, the rule's *bindings* expressions are transformed following the same mapping and appended at the end of the closure's instruction, ensuring that all the manipulated elements have been created before they are accessed (lines 22 to 24). Note that helper calls inside binding's expressions are also translated into the corresponding method calls (that have been added to the *Object* metaclass) during this operation.

6.4 Execution Environment

Figure 6.5 shows the execution environment provided by our approach: a modeler provides an ATL transformation to the *Query Translation* component (1), which compiles the query into a generic Gremlin script according to the mapping presented above, and returns it back to the modeler (2). The generated script is then sent to the *Query Execution* API with a concrete implementation of the *Model Mapping* and *Transformation Helper* libraries (3) that specify the transformation behavior and the source and target data-stores. The internal Gremlin engine finally computes the transformation using these implementations (4) and returns the result (such as execution traces, or transformation errors if any) to the modeler (5).

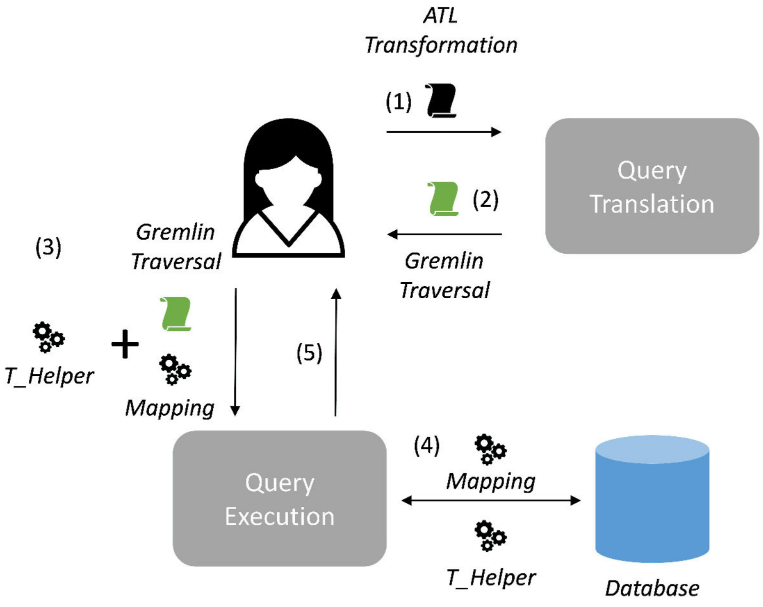


Figure 6.5 – GREMLIN-ATL Execution Environment

This architecture allows to pre-compile transformations into Gremlin scripts that can be executed multiple times without recompilation. In addition, the same generated script can be computed on top of several data-stores and using different transformation strategies by providing different implementations of the generic libraries to the *Query Execution* component, avoiding complex and costly transformation re-computations. Note that the framework can also process and compile new input transformations on the fly, generating the corresponding Gremlin scripts dynamically.

In the following, we introduce the auxiliary libraries used in the *Query Execution* component to evaluate the generated script and perform the transformation. We first introduce the low-level model mapping that provides an API to access a model through high-level modeling operations, then we present our transformation helper component that provides an API to tune the transformation computation. Note that GREMLIN-ATL embeds a set of preset model mappings and transformation helpers that can be used to access and transform models stored in the NEOEMF framework.

6.4.1 Model Mapping

The *Model Mapping* library defines the high-level modeling operations that can be computed by a given data-source storing a model. It provides a simple API allowing designers to express the implicit schema of their data-store with modeling primitives, and enables operation specific optimizations on the database side. *Model Mapping* can be manually implemented for a given database, or automatically extracted using schema inference techniques such as the NoSQLDataEngineering framework [93]. This mapping is used within the generated Gremlin query to access all the elements of a given type, retrieve element's attribute values, or navigate references.

Compared to existing modeling framework APIs like EMF that focuses on providing low-level, atomic operations (such as retrieving a specific attribute or reference of a model element), our *Model Mapping* defines high-level operations extracted from the OCL and ATL syntax. These complex operations can be translated into optimized database queries, reducing low-level query fragmentation and improving computation performances compared to the standard modeling solutions.

Table 6.3 summarizes the mapping operations used in GREMLIN-ATL and groups them into two categories: **metamodel-level** and **model-level operations**. The first group provides high-level operations that operate at the metamodel level, such as retrieving the type of an element, type conformance checks, new instances creation, and retrieving all the elements conforming to a type. The second group provides methods that compute element-based navigation, such as retrieving a referenced element, computing the parent/children of an element, and access attributes. Finally, these model-level methods allow to update and delete existing references and attributes.

Note that the *ATLtoGremlin* transformation only relies on the definition of these operations to generate a Gremlin query, and is not tailored to a specific *Model Mapping* implementation, making our approach independent of the low-level persistence solution.

6.4.2 Transformation Helper

The second component used by our *ATLtoGremlin* transformation is a *Transformation Helper* library that provides transformation-related operations called within the generated Gremlin traversal. Compared to the *Model Mapping* API presented above that focuses on accessing a data-source through modeling primitives, this library is based on the ATL syntax and execution engine, and provides a set of methods wrapping a *Model Mapping* with transformation specific operations. Note that this library aims to be generic and can be used directly in ad-hoc Gremlin scripts to manually express transformation rules.

Using an external library to compute transformation-specific operations allows to define alternative transformation algorithms that are optimized for a specific execution scenario. For example, an in-memory implementation of the trace links mechanism can be a good solution to improve the execution time of a transformation, while storing them in an on-disk persistence solution can be selected to reduce the memory consumption. To provide these specific implementations of the transformation algorithm, our *Transformation Helper* defines the following interface:

— **createElement(type, source)**: creates a new instance of the given *type* mapped to the provided *source* element.

Table 6.3 – Model Mapping API

Operation	**Description**
allOfType(type)	Returns all the strict instances of the given *type*
allOfKind(type)	Returns all the instances of the given type or one of its subtypes
getType(el)	Returns the type of the element *el*
isTypeOf(el, type)	Computes whether *el* is a strict instance of *type*
isKindOf(el, type)	Computes whether *el* is an instance of *type* or one of its subtypes
newInstance(type)	Creates a new instance of the given *type*
getParent(el)	Returns the element corresponding to the parent of *el*
getChildren(el)	Returns the elements corresponding to the children of *el*
getRef(from, ref)	Returns the elements connected to *from* with the reference *ref*
setRef(from, ref, to)	Creates a new reference *ref* between *from* and *to*
delRef(from, ref, to)	Deletes the reference *ref* between *from* and *to*
getAtt(from, att)	Returns the value of the attribute *att* contained in the element *from*
setAtt(from, att, v)	Set the value of the attribute *att* of the element *from* to the given value *v*
delAtt(from, att)	Deletes the attribute *att* contained in the element *from*

- **link(from, ref, to)**: creates a link between *from* and *to*.
- **resolveTraces(source, target)**: resolves the trace links connected to *source* with the *target*.
- **getTarget(source, bindingName)**: retrieves the target element mapped to *source*. If multiple target elements are created from a single source one an optional *bindingName* can be specified to tell the engine which one to choose.
- **isResolvable(el)**: computes whether an element can be resolved.
- **isTarget(el)**: computes whether an element is in the target model.
- **isSource(el)**: computes whether an element is in the source model.

The two first operations are wrappers around *Model Mapping* operations that add model transformation specific behavior to the mapping. The `createElement(type, sourceElement)` operation delegates to the mapping operation `newInstance(type)`, and adds a trace link between the created element and the source one. The `link(from, ref, to)` method delegates to the mapping operation *setRef* to create a regular reference between *from* and *to* or a proxy link if *to* is not yet part of the target model (i. e. if it has not been transformed so far).

The five last operations define transformation specific behaviors, such as retrieving the target element from a source one, resolve trace links, or compute whether an element is part of the source or the target model. Note that GREMLIN-ATL embeds two implementations of the *Transformation Helper* library: the first one uses an in-memory representation of the trace links to speed-up transformation computations, and the second one is optimized for in-place transformations (where the source database is used to store the result of the transformation) that can be directly plugged to customize the transformation execution. Alternative versions of this library can be defined by implementing the *Transformation Helper* API presented above.

6.5 Tool Support

As the other solutions presented in this manuscript, GREMLIN-ATL is released as a set of open source Eclipse plugins publicly available on GitHub. [10]. The framework is provided as a component of the MOGWAÏ query solution that aims to be merged in the future into a global model query and transformation solution.

The current implementation provides a simple transformation API to load an input ATL transformation, translate it into the equivalent generic Gremlin script, and execute it on a given data-source. The framework embeds a set of *Model Mappings* that can be plugged to access Neo4j and relational databases, and provides two *Transformation Helpers* for convenience purposes that can be extended to specify how transformation-related operations are computed.

ATL models are obtained from the input transformation by using the standard ATL parser, which creates a model conforming to the ATL grammar representing the abstract syntax of the transformation to compute. The resulting model is sent to the *ATLtoGremlin* transformation, which is itself defined using the ATL language (with the help of the higher-order transformation feature introduced in Section 6.2.1) and contains around 120 rules and helpers implementing the mapping and operation composition presented in Section 6.2.

As an example, Listing 14 shows the rule that creates the Gremlin instructions corresponding to an ATL attribute binding. An attribute binding b is translated into a *variableAccess* followed by a *setAtt* call (represented as a *CustomStep* in our Gremlin metamodel) containing the name of the attribute and the associated value to set. The value expression is generated using the `getFirstInstruction` helper, that returns the root element in the expression tree, and delegates the translation to the corresponding ATL rule.

```
rule attributeBinding2step {
  from
    b : ATL!Binding (binding.isAttributeBinding())
  to
    variableAccess : Gremlin!VariableAccess (
      name ← b.getBindedElementName(),
      nextElement ← setAttStep
    ),
    setAttStep : Gremlin!CustomStep(
      name ← 'setAtt',
      params ← Sequence{attNameLiteral, b.value.getFirstInstruction()
          }
    ),
    attNameLiteral : Gremlin!StringLiteral(
      value ← b.propertyName
    )
}
```

Listing 14 – AttributeBinding2Step ATL Rule

Once the generic Gremlin model has been generated, it is transformed into a textual Gremlin query using a model-to-text transformation. A final step binds the *Model Map-*

---

[10] https://github.com/atlanmod/Mogwai

ping and the *Transformation Helper* definitions to the concrete implementations provided by the modeler, and the resulting script is sent to a Gremlin script engine, which is responsible of computing the query on top of the data-store. The resulting database (or the updated one in case of in-place transformations) is finally saved with the transformed content.

Additionally, a pre-configured implementation of GREMLIN-ATL is bundled with the NEOEMF model persistence framework that extends the standard *Resource* API with transformation operations. ATL transformations are computed using the GREMLIN-ATL engine transparently, on top of a preset NeoEMF *Model Mapping* implementation. The resulting model is compatible with the NEOEMF framework and its content can be reified and manipulated using the standard EMF modeling API if needed. This transparent integration allows the use of GREMLIN-ATL for critical model transformations, while keeping the rest of the application code unchanged and compatible with existing tools based on the EMF API. We believe that this approach can ease the integration of GREMLIN-ATL into existing modeling applications that have to compute complex transformations on the top of large models.

6.6 Evaluation

In this section we evaluate the performance of the GREMLIN-ATL framework by comparing the execution performance of the same set of ATL transformations using the standard ATL engine and the GREMLIN-ATL framework. Note that we do not consider alternative transformation frameworks in this preliminary evaluation, because they are either based on the same low-level modeling API as ATL (such as QVT), or are not designed to compute transformation on top of large models (such as Viatra), and we could not envision a fair comparison scenario with our solution.

We run two transformations, a toy one created on purpose for this analysis and a more complex one taken from an industrial project involving a reverse engineering scenario. Transformation computations are evaluated in terms of execution time and memory consumption on a set of models of increasing size, stored in Neo4j using the NEOEMF mapping. Resulting models are also stored in NEOEMF using the same mapping.

6.6.1 Benchmark Presentation

Experiments are run over the three models presented in Section 3.5 plus an additional larger one that allows to test our approach on top of very-large models. As a remainder, Table 6.4 presents the size of the input models in terms of number of elements they contain and XMI file size. These models are migrated to NEOEMF/GRAPH before executing the benchmark using the NEOEMF *io* module to enable scalable access of their contents.

In order to evaluate our approach, we perform two transformations that take as their input models conforming to the MoDisco Java metamodel, and translate them into a KDM model [81]. The *AbstractTypeDeclaration2DataType* transformation matches all the *AbstractTypeDeclaration* elements (declared classes, interfaces, enumerations, etc.) of the input model and create the corresponding KDM *DataType*. It is composed of a single rule that matches all the subtypes of *AbstractTypeDeclaration*, and is used as the basis to

Table 6.4 – Benchmarked Models

Model	**# Elements**	**XMI Size** (MB)
set1	6756	1.7
set2	80 665	20.2
set3	1 557 007	420.6
set4	3 609 354	983.7

evaluate if the transformation complexity (i.e. the number of rules) has an impact on the overall execution time and memory consumption.

The second benchmarked transformation is a subset of the *Java2KDM* transformation embedded in the MoDisco platform itself. It is extracted from an existing industrial transformation that takes a low-level Java model and creates the corresponding abstract KDM model. This transformation is typically the first step of MoDisco-based reverse engineering processes, and produces a generic model that can be manipulated by the framework regardless of its concrete implementation (Java, JavaScript, etc), and used to compute metrics, perform refactoring operations, and generate updated software artifacts.

The two transformations are run in a 512 MB Java virtual machine with the arguments `-server` and `-XX:+UseConcMarkSweepGC` that are recommended by the Neo4j documentation.

6.6.2 Results

Tables 6.5 and 6.6 present the results of executing the transformations on top of the input model sets. First columns contain the name of the input model of the transformation, second and third columns present the execution time and the memory consumption of the standard ATL engine and GREMLIN-ATL , respectively. Execution times are expressed in milliseconds and memory consumption in megabytes.

The correctness of the output models are checked by comparing the results of our approach with the ones generated by running the ATL transformation on the original input XMI files using a large Java virtual machine able to handle it. The comparison is performed using EMF Compare [18], an open-source Eclipse plugin that aims to provide efficient comparison of EMF models. Note that the presented results have been obtained by computing the average execution time and memory consumption values after 30 executions of the transformations.

6.6.3 Discussion

The results presented in Tables 6.5 and 6.6 show that GREMLIN-ATL is a good candidate to compute model transformations on top of large models stored in NEOEMF/-GRAPH . Our framework is faster than the standard ATL engine for the two benchmarked transformations, outperforming it both in terms of execution time and memory consumption for the larger models.

The results also show that the complexity of the transformation has a significant im-

Table 6.5 – AbstractTypeDeclaration2DataType Results

Model	Execution Time (ms)		Memory Consumption (MB)	
	ATL	Gremlin-ATL	ATL	Gremlin-ATL
set1	3505	1139	3.2	10
set2	11 480	1649	17.6	11.7
set3	67 204	3427	99.3	23
set4	*OOM*[1]	11 843	*OOM*	100

[1] The application threw an OutOfMemory error after two hours

Table 6.6 – Java2KDM Results

Model	Execution Time (ms)		Memory Consumption (MB)	
	ATL	Gremlin-ATL	ATL	Gremlin-ATL
set1	4874	2469	11	12.8
set2	33 407	4321	45.2	23.2
set3	5 156 798	38 402	504.5	52
set4	*OOM*	129 908	*OOM*	96

pact on ATL's performances. This is particularly visible when the transformations are evaluated on the larger models: *Java2KDM* is 2.9 times slower than *AbstractType2DataType* on *set2*, and up to 76 times on *set3*. This difference is explained by the large number of low-level modeling API calls that are generated by the *Java2KDM* transformation in order to retrieve the matching elements, compute rules' conditions, and helpers' body.

GREMLIN-ATL 's execution time is less impacted by the transformation complexity, because the generated Gremlin query is entirely computed at the database level, bypassing the modeling API and reducing the number of intermediate object reification. The database engine optimizes the query to detect access patterns and cache elements efficiently, and allows to benefit from the built-in indexes to retrieve elements efficiently. Results on *set3* show that GREMLIN-ATL 's approach is faster than ATL by a factor of 19 and 134 for *AbstractType2DataType* and *Java2KDM*, respectively.

The presented tables also emphasize that ATL's performance on *set3* and *set4* is tightly coupled to the memory consumption. Indeed, in our experiments we measured that most of the execution time was spent in garbage collection operations. This high memory consumption is caused by the in-memory nature of the engine, that keeps in memory all the matched elements, as well as the trace links between source and target models. When the input model grows, this in-memory information grows accordingly, triggering the garbage collector, which blocks the application until enough memory has been freed. In addition, the intensive usage of the low-level model handling API increases the memory overhead, by reifying intermediate modeling elements that also stay in memory.

Conversely, GREMLIN-ATL does not require these complex structures, because trace links are stored within the database itself, and can be removed from the memory if needed. This implementation avoids garbage collection pauses, and allows to scale to very large models. Our approach operates on the optimized database structures, and thus does not have to reify modeling elements, improving the memory consumption. Looking at the *Java2KDM* transformation, this strategy reduces the memory consumption by a factor of 10 for *set3*.

Finally, ATL requires less memory than GREMLIN-ATL to compute the transformations on top of smaller models (*set1*). This difference is caused by the initialization of the Gremlin engine (and its underlying Groovy interpreter) used to evaluate the generated scripts. However, this extra memory consumption has a fixed size and does not depend on the evaluated transformation nor on the transformed model.

Note that the presented results only show execution time and memory consumption related to the computation of the transformations. We did not take into account the time and memory required to generate an executable file that can be interpreted by the ATL engine nor the time needed to produce the Gremlin script to compute, because this extra costs is fixed for a given transformation and does not depend on the model size. In addition, GREMLIN-ATL allows to pre-compile and to cache existing Gremlin queries in order to limit script generation.

6.7 Conclusion

In this chapter we presented GREMLIN-ATL , a framework that computes rule-based model transformations by reexpressing them as database queries written in the Gremlin traversal language. Our approach is based on a modular architecture which allows to compute transformations on top of several data stores, and we presented a simple use case where GREMLIN-ATL is used to migrate data from an existing Neo4j data store into a relational database, thanks to the modularity provided by the *Model Mapping* component.

GREMLIN-ATL also embeds a *Transformation Helper* that allows to define finely how to compute transformation operations. We use this component to further improve the scalability of our approach by storing the information related to the transformation itself within the database, limiting the memory consumption. Our evaluation shows that using GREMLIN-ATL to transform large models significantly improves the performance both in terms of execution time and memory consumption.

Finally, our frameworks fulfill all the requirements listed in Section 6.5: GREMLIN-ATL bypasses the existing modeling APIs to generate Gremlin traversals that are computed on the database side, benefiting of its low-level features such as index lookups and query optimizations (**Rq1**). Our *ATLtoGremlin* component takes as its input an ATL transformation that is parsed and translated into an equivalent Gremlin traversal, allowing to use the high-level constructs provided by the language to express the transformation (**Rq2**). Our preliminary experiments report that using GREMLIN-ATL to compute a well-known MDRE transformation outperforms the standard ATL engine when applied on top of large models stored in current model persistence frameworks. However, additional experiments are required to assess the benefits of GREMLIN-ATL when compared to alternative transformation solutions such as QVT or ETL. Finally, our *Model Mapping* and *Transformation Helper* components allow to dynamically adapt the transformation computation to fit a specific data-store and use a dedicated implementation of the transformation algorithm that fits the modeler needs (**Rq 4** and **5**).

In the following we show how NEOEMF , MOGWAÏ , and GREMLIN-ATL can be reused and combined in a complete application example that aims to generate graph database access code and invariant verifications from high-level conceptual schemas.

7

A NeoEMF/Mogwaï Infrastructure to Generate Database Software

In the previous chapters we have introduced solutions that enable to efficiently persist, query, and transform large models stored in NoSQL databases. The presented approaches are integrated into the modeling ecosystem (especially the EMF environment), and our experiments have shown that relying on the data-store facilities to store and access large models can significantly improve applications' execution time and memory footprint.

While these NoSQL-based techniques are promising in the context of MDE, we also believe that bringing modeling techniques at the data-store level can ease the definition, development, and maintenance of NoSQL applications. Indeed, NoSQL data-stores have become a promising solution to enhance scalability, availability, and query performance of data intensive applications. Their schemaless infrastructure offers great flexibility since it is possible to use different representations of a same concept (non-uniform data), but client applications still need to know (at least partially) how conceptual elements are stored in the database in order to access and manipulate them. Acquiring this implicit knowledge of the underlying schema can be an important issue, for example in data integration processes, where each data source has to be inspected to find its underlying structure [56].

In order to take full benefit of NoSQL solutions, designers must be able to integrate them into current code-generation architectures and use them as target persistence backend for their conceptual schemas. Unfortunately, while several solutions provide transformations from Entity Relationship (ER) and UML models to relational database schemas, the same is not true for NoSQL databases as discussed in detail in the related work. Moreover, NoSQL databases present an additional challenge: data consistency is a big problem since the vast majority of NoSQL approaches lack any advanced mechanism for integrity constraint checking [79].

In this chapter we emphasize how the contributions of this thesis that are designed to solve core MDE issues can be combined into a solution dedicated to bridging the gap between conceptual modeling and NoSQL (especially graph database) infrastructures. UMLTOGRAPHDB is a model-driven approach that translates conceptual schemas expressed using UML [86] into a graph representation, and generates database-level queries from business rules and invariants defined using OCL [83]. Our approach reuses the implicit model to database mapping embedded in NEOEMF (chapter 3), and integrates the MOGWAÏ framework (chapter 5) to generate database queries ensuring data integrity. The framework relies on a new GraphDB metamodel, as an intermediate representation to facilitate the integration of several kinds of graph databases. Enforcement of (both OCL and structural) constraints is delegated to an intermediate software component (middleware) in charge of maintaining the underlying database consistent with the conceptual schema. External applications can then use this middleware to safely access the database. This is illustrated in Figure 7.1.

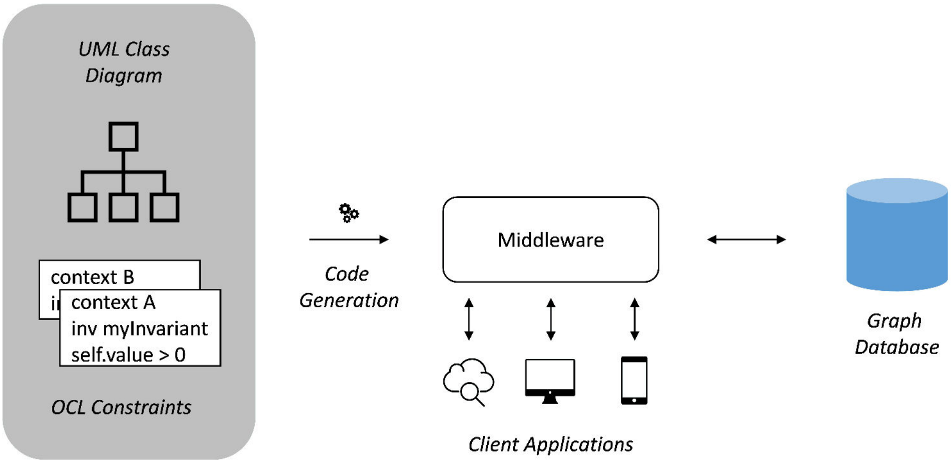


Figure 7.1 – Conceptual Model to Graph database

The rest of the chapter is structured as follows: Section 7.1 presents the UMLTOGRAPHDB framework and its core components, Section 7.2 introduces the GraphDB metamodel and details the model-to-model transformation which creates an instance of GraphDB from a UML model. Section 7.3 presents the transformation that creates graph database queries from OCL expressions, and Section 7.4 introduces the code generator. Finally, Section 7.5 describes our tool support, Section 7.6 presents the related work and Section 7.7 summarizes the key points and draws conclusions.

7.1 UMLtoGraphDB Approach

UMLTOGRAPHDB is aligned with the OMG's MDA standard [80], proposing a structured methodology to systems development that promotes the separation between a specification defined in a platform independent way (PIM), and the refinement of that specification adapted to the technical constraints of the implementation platform (PSM). In our scenario, the initial UML and OCL models would conform to the PIM level. UMLTOGRAPHDB takes care of generating the PSM and the middleware code from them. Figure 7.2 presents the different components of the UMLTOGRAPHDB framework (light-grey box).

In particular, **Class2GraphDB** (1) is the first model transformation of the UMLTOGRAPHDB framework. It is in charge of the creation of a low-level graph representation (PSM) from the input UML class diagram (PIM). The output of the Class2GraphDB transformation is a **GraphDB Model** (2), conforming to the GraphDB metamodel (Section 7.2). This metamodel is defined at the PSM level, and describes data structures in terms of graph primitives, such as *vertices* or *edges*. The **OCL2Gremlin** transformation (3) is the second transformation applied on the input models. It is in charge of the translation of the OCL constraints, queries, and business rules defined at the PIM level into graph-level queries. The transformation produces a **Gremlin Model**, conforming to the Gremlin language metamodel defined in Section 5.2, that complements the previous GraphDB one with low-level queries representing constraints and invariants to check.

The last step in MDA process is a PSM-to-code transformation, which generates the software artifacts (database schema, code, configuration files . . .) in the target platform. In our approach, this final step is handled by the **Graph2Code** (5) transformation (Section 7.4) that processes the generated GraphDB and Gremlin models to create a set of Java Classes wrapping the structure of the database, the associated constraints, and the business rules. These Java classes compose the **Middleware** layer (6) presented in Figure 7.1, and contain the generated code to access the physical **Graph Database**[1] (7).

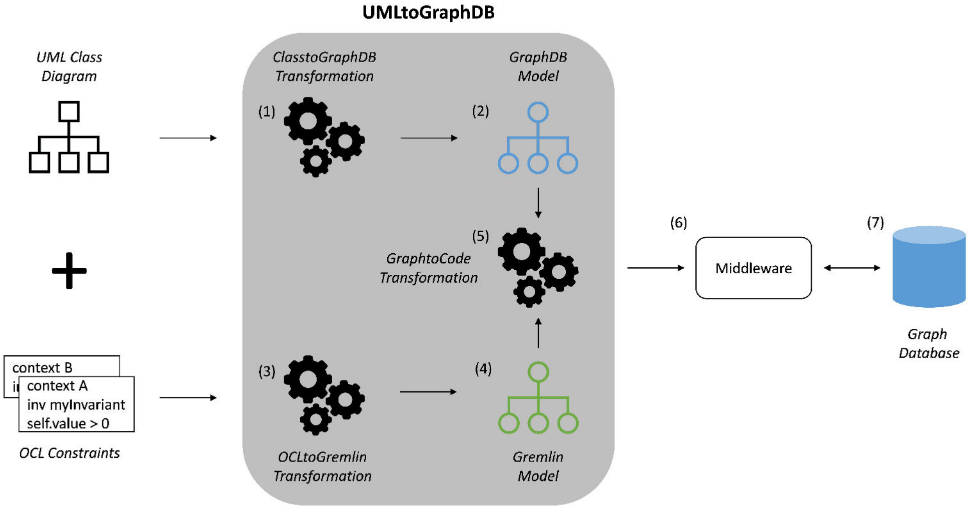


Figure 7.2 – Overview of the UMLTOGRAPHDB Infrastructure

To better illustrate the different transformation steps of our framework, we introduce as a running example the conceptual schema presented in Figure 7.3 representing a simple excerpt of an e-commerce application. This schema is specified using the UML notation, and describes *Client*, *Orders*, and *Products* concepts. A *Client* is an abstract class defined by a name and an address. *PrivateCustomers* and *CorporateCustomers* are subclasses of *Client*. They contain respectively a *cardNumber* and a *contractRef* attribute. *Clients* own *Orders*, that are defined by a *reference*, a *shipmentDate*, and a *deliveryDate*. In addition, an *Order* maintains a *paid* attribute, that is set to true if the *Order* has been paid. *Products* are defined by their *name*, *price*, a textual *description*, and are linked to *Orders* through the *OrderLine* association class, which records the *quantity* and the *price* of each *Product* in a given *Order*.

---

[1] While this work is focused on graph database access code generation, the same architecture could be applied to document, key-values, or wide-column data-stores.

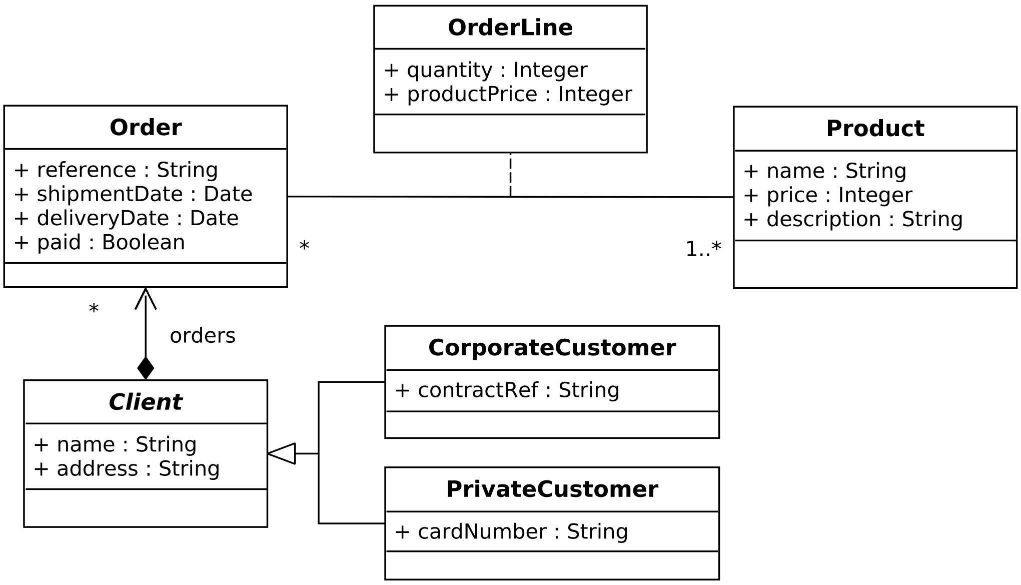

OrderLine
+ quantity : Integer
+ productPrice : Integer
Order
+ reference : String
+ shipmentDate : Date
+ deliveryDate : Date
+ paid : Boolean
Product
+ name : String
+ price : Integer
+ description : String
*
1..*
*
orders
Client
+ name : String
+ address : String
CorporateCustomer
+ contractRef : String
PrivateCustomer
+ cardNumber : String

Figure 7.3 – Class Diagram of a Simple e-commerce Application

In addition, our conceptual data model defines three OCL constraints (presented in Listing 15), which represent basic business rules. The first one checks that the *price* of a *Product* is always positive, the second one verifies that the *shipmentDate* of an *Order* precedes its *deliveryDate*, and the last one ensures a *Client* has less than three unpaid *Orders*.

```
context Product inv validPrice: self.price > 0
context Order inv validOrder: self.shipmentDate < self.deliveryDate
context Client inv maxUnpaidOrders:
    self.orders→select(o | not o.paid)→size() < 3
```

Listing 15 – Textual Constraints

In the following we introduce the internal components in our approach and we detail how the example conceptual schema presented in Figure 7.3 is mapped into a PSM dedicated to graph databases. Then we show how the business rules and invariants expressed in Listing 15 are mapped to Gremlin steps and assembled into efficient database queries. We finally show how these low-level models are weaved by our model-to-text transformation to generate the application code wrapping the database.

7.2 Mapping UML Class Diagram to GraphDB

We now present the *Class2Graph* transformation, which is the initial step in the approach presented in Figure 7.2. We first introduce the GraphDB metamodel that aims to represent graph database's implicit schema, then we focus on the transformation that generates a GraphDB instance from a conceptual model defined in UML.

7.2.1 GraphDB Metamodel

The GraphDB metamodel presented in Figure 7.4 is our proposal to represent the possible internal structures of a graph database. It is based on Blueprints, the generic NoSQL API presented in Section 5.2.1 that constitutes the basis of the Tinkerpop stack [110], including the Gremlin traversal language. Note that the version of Blueprints used in this chapter slightly differs from the one used in Chapter 3 and 6 for compatibility reasons: UMLTOGRAPHDB relies on the latest Blueprints version that supports Neo4j 2/3, while NEOEMF -compatible tools are based on a previous version in order to provide historical support for Neo4j 1 [2].

Our metamodel defines a *GraphSpecification* element that represents the top-level container that owns all the database objects. It contains a *baseDB* attribute, that defines the concrete database to instantiate under the Blueprints API. In our prototype, the baseDB can be either Neo4j or OrientDB, two well known graph database implementations. A *GraphSpecification* also contains all the *VertexDefinitions* and *EdgeDefinitions* —representing the possible vertex and edge constructs in the graph— through the associations *vertices* and *edges*.

A *VertexDefinition* can be *unique*, meaning that there is only one vertex in the database that conforms to it. *VertexDefinitions* and *EdgeDefinitions* can be linked together using *outEdges* and *inEdges* associations, meaning respectively that a *VertexDefinition* has outgoing edges and incoming edges. In addition, *VertexDefinition* and *EdgeDefinition* are both subtypes of *GraphElement*, which can define a set of *labels* [3] that describe the type of the element, and a set of *PropertiesDefinition* through its *properties* reference. In graph databases, properties are represented by a *key* (the name of the property) and a *Type*. In the first version of this metamodel we define four primitive types: *Object*, *Integer*, *String*, and *Boolean*.

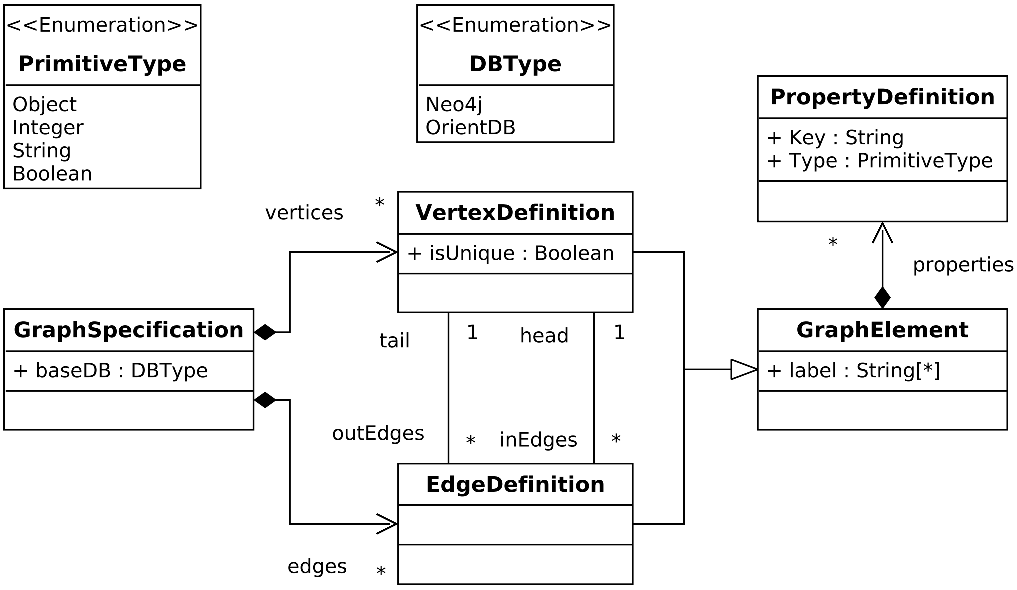

<<Enumeration>>
PrimitiveType
Object
Integer
String
Boolean
<<Enumeration>>
DBType
Neo4j
OrientDB
PropertyDefinition
+ Key : String
+ Type : PrimitiveType
vertices
*
VertexDefinition
+ isUnique : Boolean
*
properties
GraphSpecification
+ baseDB : DBType
tail
1
head
1
GraphElement
+ label : String[*]
outEdges
*
inEdges
*
EdgeDefinition
edges
*

Figure 7.4 – GraphDB Metamodel

---

[2] Migrating all the presented solutions to Blueprints 3 is planned in a future release

[3] Labels are specific to Blueprints 3

7.2.2 Class2GraphDB Transformation

The Class2GraphDB transformation is responsible for creating instances of the GraphDB metamodel presented above from conceptual schemas expressed in UML. It is adapted from the implicit mapping between Ecore operations and Blueprints constructs embeded in NEOEMF/GRAPH , as well as the NEOEMF *Model Mapping* introduced in Section 6.5.

Intuitively, the transformation consists of mapping UML *Classes* to *VertexDefinitions*, *Associations* to *EdgeDefinitions*, and *AssociationClasses* to new *VertexDefinitions* connected to the ones representing the involved classes. The mapping also creates *PropertyDefinitions* for each *Attribute* in the input model, and adds them to the corresponding mapped element.

Note that our GraphDB metamodel has no construct to represent explicitly inheritance, and thus, the mapping has to deal with inherited attributes and associations. To handle them, the translation flattens the inheritance hierarchy by finding all the attributes and associations in the parent hierarchy of each class, and adding them to the mapped *VertexDefinition*. While this creates duplicated elements in the GraphDB model, it is the more direct representation to facilitate queries on the resulting database.

Specifically, a class diagram *CD* is defined as a tuple $CD = (Cl, As, Ac, I)$, where *Cl* is the set of classes, *As* is the set of associations, *Ac* is the set of association classes, and *I* the set of pairs of classes such as $(c1, c2)$ represents the fact that $c1$ is a direct or indirect subclass of $c2$. Support for more advanced concepts such as enumarations and interfaces is planned for future work.

A GraphDB diagram *GD* is defined as a tuple $GD = (V, E, P)$, where *V* is set of vertex definitions, and *E* the set of edge definitions, and *P* the set of property definitions that compose the graph. Based on these defninitions, we can define our transformation through the following rules:

— **R1:** each class $c \in Cl, not\ c.isAbstract$ is mapped to a vertex definition $v \in V$, where $v.label = c.name \cup c_{parents}.name$, with $c_{parents} \subset Cl$ and $\forall p \in c_{parents}, (c, p) \in I$.

— **R2:** each attribute $a \in (c \cup c_{parents}).attributes$ is mapped to a property definition p, where $p.key = a.name$, $p.type = a.type$, and added to the property list of its mapped container v such as $p \in v.properties$.

— **R3:** each association $as \in As$ between two classes $c_1, c_2 \in Cl$ is mapped to an edge definition $e \in E$, where $e.label = as.name$, $e.tail = v_1$, and $e.head = v_2$, where v_1 and v_2 are the *VertexDefinitions* representing c_1 and c_2. Note that $e.tail$ and $e.head$ values are set according to the direction of the association. If the association is not directed, a second edge definitions $e_{opposite}$ is created, where $e_{opposite}.label = as.name$, $e_{opposite}.tail = v_2$, and $e_{opposite}.head = v_1$, representing the second possible direction of the association. Aggregation associations are mapped the same way, but their semantic is handled differently in the generated code. In order to support inherited associations, *EdgeDefinitions* are also created to represent associations involving the parents of c.

— **R4:** each association $as \in As$ between multiple classes $c_1 ... c_n \in Cl$ is mapped to a vertex definition v_{asso} such as $v_{asso}.label = as.name$ and a set of *EdgeDefinitions* $e_i.tail = v_i$ and $e_i.head = v_{asso}$, associating the created vertex definition to the ones representing $c_1 ... c_n$.

— **R5:** each association class $ac \in Ac$ between classes $c_1 ... c_n$ is mapped like an association between multiple classes using a vertex definition v_{ac} such as $v_{ac}.label = ac.name$. As for a regular class, v_{ac} contains the properties corresponding to the attributes $ac.attributes$, and a set of *EdgeDefinitions* $e_i \in E$ where $e_i.tail = v_i$ and $e_i.head = v_{ac}$.

To better illustrate this mapping, we now describe how the GraphDB model shown in Figure 7.5 is created from the example presented in Figure 7.3. Note that for the sake of readability we only show an excerpt of the created GraphDB model. To begin with, all the classes are translated into *VertexDefinition* instances following *R1*. This process generates the elements `v1`, `v2`, `v3`, and `v4`, with the labels *(Client, PrivateCustomer)*, *(Client,CorporateCustomer)*, *Order*, and *Product*. Then, *R2* is applied to transform attributes into *PropertyDefinitions*. For example, the attribute *name* of the class *Client* is mapped to the *PropertyDefinition* `p1`, which defines a key *name* and a type *String*. These *PropertyDefinition* elements are linked to their containing *VertexDefinition* using the *properties* association. Once this first step has been done, *R3* is applied on the association *orders*, mapping it to the *EdgeDefinitions* `e1` and `e2`, containing the name of the association. *VertexDefinitions* representing *PrivateCustomer* and *CorporateCustomer* classes are then linked to the one representing *Order*, respectively with `e1` and `e2`. Since the association *orders* is directed, the transformation puts `v1` and `v2` as the tail of the edge, and `v3` as its head. Then, the association class *OrderLine* is transformed by *R5* to the *VertexDefinition* `v5`, and its attributes *productPrice* and *quantity* are transformed into the *PropertyDefinitions* `p6` and `p7`. Finally, two *EdgeDefinitions* (`e3` and `e4`) are also created to link the *VertexDefinition* `v3` and `v4` to it.

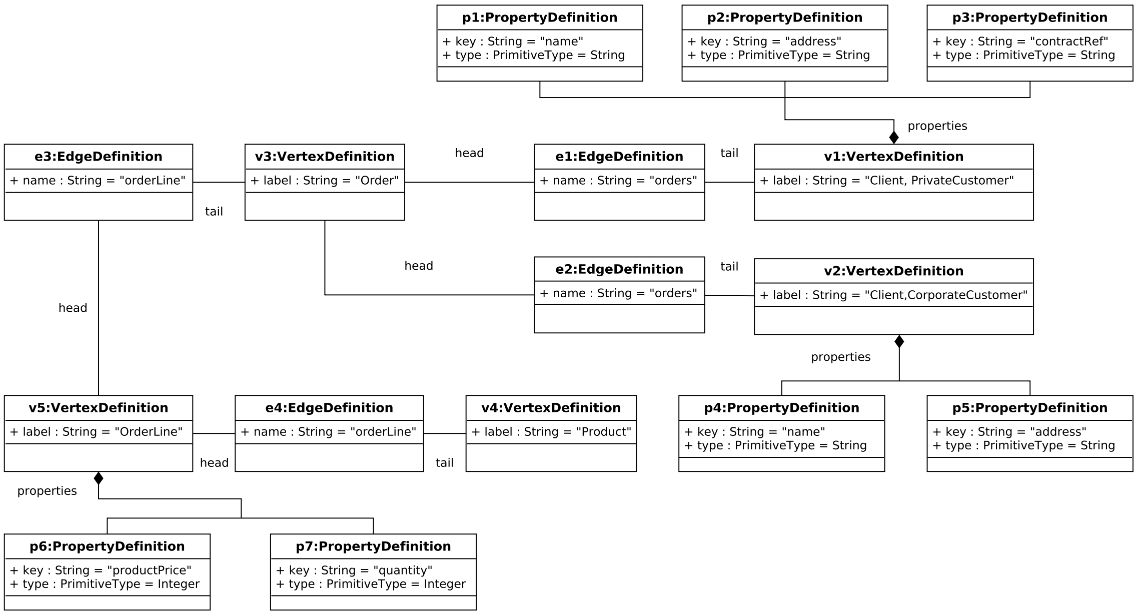

p1:PropertyDefinition
+ key : String = "name"
+ type : PrimitiveType = String
p2:PropertyDefinition
+ key : String = "address"
+ type : PrimitiveType = String
p3:PropertyDefinition
+ key : String = "contractRef"
+ type : PrimitiveType = String
properties
e3:EdgeDefinition
+ name : String = "orderLine"
v3:VertexDefinition
+ label : String = "Order"
head
e1:EdgeDefinition
+ name : String = "orders"
tail
v1:VertexDefinition
+ label : String = "Client, PrivateCustomer"
tail
head
e2:EdgeDefinition
+ name : String = "orders"
tail
v2:VertexDefinition
+ label : String = "Client,CorporateCustomer"
head
properties
v5:VertexDefinition
+ label : String = "OrderLine"
e4:EdgeDefinition
+ name : String = "orderLine"
v4:VertexDefinition
+ label : String = "Product"
p4:PropertyDefinition
+ key : String = "name"
+ type : PrimitiveType = String
p5:PropertyDefinition
+ key : String = "address"
+ type : PrimitiveType = String
head
tail
properties
p6:PropertyDefinition
+ key : String = "productPrice"
+ type : PrimitiveType = Integer
p7:PropertyDefinition
+ key : String = "quantity"
+ type : PrimitiveType = Integer

Figure 7.5 – Excerpt of the Mapped GraphDB Model

These mapping rules have been specified as an ATL transformation [58] containing 45 helpers and rules that match the supported UML constructs and create their GraphDB equivalent. As an example, Listing 16 shows the ATL transformation rule that maps a UML *Class* to a *VertexDefinition*. It is applied for each non-abstract *Class* element, excepted *AssociationClasses*, which are mapped according to *R5*. The rule creates a *Ver-*

VertexDefinition element, and sets its label attribute with the *name* of each *Class* in its parent hierarchy. The set of parent *Classes* is computed by the helper `getParentClassHierarchy`, which returns a sequence containing all the parents of the current *Class*. Finally, *VertexDefinition properties* are set, by getting all the attributes from the parent hierarchy, and are transformed by the abstract *lazy rule* `GenericAttribute2Property`. The complete ATL transformation is available in the project repository[4].

```
rule Class2VertexDefinition {
  from
    class : UML!Class (not(class.oclIsTypeOf(UML!AssociationClass))
        and not(class.abstract))
  to
      vertex : Graph!VertexDefinition (
      labels ← class.getParentClassHierarchy()→collect(cc | cc.name)
      -- Generate a property for each Attribute in the class hierarchy
      properties ← class.getParentClassHierarchy()
        →collect(cc | cc.attribute)
        →collect(att | thisModule.GenericAttribute2Property(att))
      )
}
```

Listing 16 – Class2VertexDefinition ATL Transformation Rule

7.3 Translating OCL Expressions to Gremlin

Once the GraphDB model has been created, another transformation is performed to translate the OCL expressions defined in the conceptual schema into a Gremlin query model. The mapping presented in this Section is adapted from the MOGWAÏ translation presented in Chapter 5 dedicated to OCL query evaluation on top of models stored in NEOEMF/GRAPH . Note that this translation is updated to fit the UMLTOGRAPHDB architecture that relies on Blueprints 3.

Table 7.1 presents an excerpt of the mapping between OCL expressions and Gremlin concepts. Compared to the mapping presented in Section 5.2, this implementation handles type conformance and `allInstances` operations using the label mechanism provided by Blueprints 3 databases. Other operations are simply mapped the same way they are in the MOGWAÏ framework, and are simply presented here as a remainder.

These mappings are systematically applied on the input OCL expressions, following a postorder traversal of the OCL Abstract Syntax Tree. Note that the transformation reuses the same composition process to assemble the final query as the one presented in Section 5.2. As an example, Listing 17 shows the Gremlin queries generated from the OCL constraints of the running example (Section 7.1). The `v` variable represents the vertex that is being currently checked (i. e. the `self` context variable in OCL constraints), and the following steps are created using the mapping. Note that generated expressions are queries that return a boolean value. These queries are embedded in checking methods during the final code generation phase (Section 7.4).

---

[4] https://github.com/atlanmod/UML2NoSQL

Table 7.1 – OCL to Gremlin mapping

OCL expression	**Gremlin step**
Type	"Type.name"
C.allInstances()	g.V().hasLabel("C.name")
collect(attribute)	property(attribute)
collect(reference)	outE('reference').inV
oclIsTypeOf(C)	o.hasLabel("C.name")
col$_1$ $\rightarrow$union(col$_2$)	col$_1$.fill(var$_1$); col$_2$.fill(var$_2$); union(var$_1$, var$_2$);
including(object)	gather{it << object;}.scatter;
excluding(object)	except([object]);
size()	count()
isEmpty()	toList().isEmpty()
select(condition)	c.filter{condition}
reject(condition)	c.filter{!(condition)}
exists(expression)	filter{condition}.hasNext()
$=, >, >=, <, <=, <>$	$==, >, >=, <, <=, ! =$
$+, -, /, \%, *$	$+, -, /, \%, *$
and,or,not	&&,||,!
variable	variable
literals	*literals*

```
v.property("price") > 0; // validPrice
v.property("shipmentDate") < self.property("deliveryDate"); //
    validOrder
v.outE("orders").inV.filter{it.property("paid")==false}
    .count() < 3; // maxUnpaidOrders
```

Listing 17 – Generated Gremlin Queries

7.4 Code Generation

The last step in our UMLTOGRAPHDB infrastructure is a model-to-text transformation that processes the generated GraphDB and Gremlin models and creates the middleware component that wraps database accesses. Our code-generator relies on the Blueprints API for interacting with the graph database in a vendor neutral way. We first briefly review the Blueprints concepts used in our middleware, then we show how we leverage them to enforce that any application aiming to query/store data through the created middleware does it so according to the its initial UML/OCL conceptual schema.

7.4.1 Blueprints API

The Blueprints API is composed of a Java layer that allows to manipulate graph databases in a generic way, and is the basis of the Gremlin language. It is composed of a set of classes that wraps database-level elements, such as *vertices* and *edges*, providing methods to access, update, and delete them. A Blueprints database is instantiated using a `GraphFactory` instance, that takes a configuration file containing the properties of the databases (the type of the underlying graph engine, the allocated memory for vertices and edges caches, etc) and creates the corresponding graph store.

The Blueprints `Vertex` class provides the methods `addEdge(String label, Vertex otherEnd)` and `removeEdge(otherEnd)` that allow to connect/disconnect two vertices by creating/deleting an edge between the current vertex and *otherEnd* with the given *label*. Blueprints also defines the vertex method `property(String key)`, that retrieve the value of the vertex property defined by the given property key. In addition, the Blueprints API provides the `traversal()` method, that allows to compute Gremlin traversals —expressed using an internal Java DSL— and returns the resulting record wrapped in Blueprints constructs[5].

7.4.2 Graph2Code Transformation

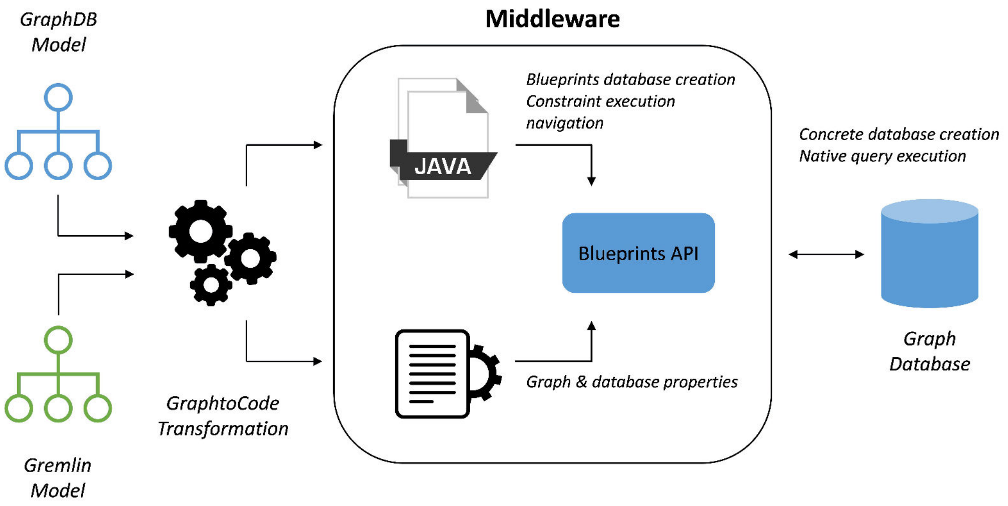

GraphDB
Model
Middleware
Blueprints database creation
Constraint execution
navigation
JAVA
Concrete database creation
Native query execution
Blueprints API
Graph
Database
Graph & database properties
GraphtoCode
Transformation
Gremlin
Model

Figure 7.6 – Generated Infrastructure

Figure 7.6 presents the infrastructure generated by the Graph2Code transformation. In short, the generator processes the GraphDB model to retrieve all the *VertexDefinition* elements and, for each one, it creates a corresponding Java class with the relevant *getters* and *setters* for its attributes (derived from the properties definitions linked to the vertex) and associations (derived from the input/output edges of the vertex).

Listing 18 presents an excerpt of the Java class generated from the *Client* element. Note that this class extends *BlueprintsBean*, which is a generic class that we provide as part of the UMLTOGRAPHDB infrastructure. *BlueprintsBean* provides auxiliary methods to connect the class with the Graph database via the Blueprints API and facilitates the creation and management of graph elements.

Once this basic Java class structure have been generated, the transformation starts processing the *Gremlin Model* to create additional methods. Each method is in charge of checking one of the OCL constraints (or queries) in the conceptual schema. As usual, checking methods return a boolean value (*false* if the constraint is violated). As an example, Listing 18 includes the method `checkMaxUnpaidOrder` executing the Gremlin traversal mapped from the OCL expression `self.orders→select(o | not o.paid)→size() < 3`. The generated expression follows the Java internal DSL of the Gremlin language and not the Groovy-based syntax presented in Section 5.2.1, yet both versions can be generated by our infrastructure. Note that the task of calling the generated constraint-checking method is responsibility of the client application. Automatic and incremental checking of these constraints is left for future work.

---

[5] A complete reference of the Blueprints API is available in the Tinkerpop documentation [110]

Finally, the Graph2Code generator creates a *Configuration File* that contains the graph and database properties, and is used by the Blueprints API to instantiate the concrete graph engine.

```java
public class Client extends BlueprintsBean {
  public String getName() {
    return (String)this.vertex.property("name").value();
  }
  public String getAddress() {
    return (String)this.vertex.property("address").value();
  }
  public void setName(String newName) {
    this.vertex.property("name",newName);
  }
  public void setAddress(String newAddress) {
    this.vertex.property("address",newAddress);
  }
  public void addOrder(Order order) {
    this.vertex.addEdge("orders", order.getVertex());
  }
  public void removeOrder(Order order) {
    this.vertex.removeEdge(order.getVertex());
  }
  public boolean checkMaxUnpaidOrders() {
    return this.graph.traversal().V(this.vertex).outE("orders")
      .inV().filter(v → v.get().<Boolean>property("paid").value())
        .count().is(P.lt(3)).hasNext();
  }
}
```

Listing 18 – Generated Client Java Class

7.5 Tool Support

A prototype of the UMLtoGraphDB framework has been implemented as a collection of open-source Eclipse plugins, available on Github[6]. UMLtoGraphDB takes as input the UML and OCL files (defined, for instance, using Eclipse-based UML editors such as Papyrus[7]), that are then translated, respectively, by the *Class2GraphDB* and *OCL2Gremlin* ATL transformations seen before. These transformations add up to a total of 110 rules and helper functions.

Our code-generator is implemented using the XTend platform [12], that provides a template-based language for model-based code generation. The language itself is expressed as a superset of Java that provides syntactic sugar, lambda expressions, and other useful extensions to process input models and generate the final software code efficiently. The generator takes the GraphDB and Gremlin models and processes them as described in Section 7.4, and produces the middleware containing our Blueprints-based classes that wraps the database and ensures data consistency.

---

[6] `https://github.com/atlanmod/UML2NoSQL`

[7] `https://eclipse.org/papyrus/`

The time needed by the entire transformation chain to produce the Java code from the input UML and OCL specifications is in the order of a few seconds for the several examples we have tested. A precise analysis of the scalability of the transformation performance according to the size of the input for very large conceptual model is left for future work.

7.6 Related Work

Mapping conceptual schemas to relational databases is a well-studied field of research[74]. A few research efforts also cover schemas that include (OCL) constraints. For example, Demuth and Hussman [39] propose a mapping from UML (augmented with OCL constraints) to SQL that covers most of the OCL specification and implements it via a code generator [40] that automates the process. Brambilla and Cabot [17] propose a methodology to implement integrity constraints into relational databases recommending alternative implementations based on performance parameters. While these approaches are well-suited for relational databases, they all rely on the generation of database constraints to ensure data consistency. However, in the NoSQL ecosystem —and especially for graph databases—, there is a lack of support for built-in constraint constructs, and data validation must be delegated to the application layer, such as the middleware component generated by our approach.

Li et al. propose an approach to transform UML class diagrams into a HBase data models [72], by mapping classes to tables, attributes to columns, and providing transformation rules for associations, compositions, and generalizations. Still, it is only applicable to column-based datastores, and does not support the definition of custom OCL constraints and business rules.

More specific to NoSQL databases, the NoSQL Schema Evaluator [75] generates query implementation plans from a conceptual schema and workload definition. For now, the approach is limited to Cassandra, but authors intend to adapt it to different data models, such as key-values and document stores. However, this solution does not take into account constraints specified in the conceptual model. Sevilla et al. [93] presented a tool to infer versioned schemas from NoSQL databases. The resulting model is then used to automatically generate a viewer and validator for the schema but they do not aim to provide support for a full-fledged application nor consider the addition of constraints on the reversed schema. Bugiotti et al. [22] propose a database design methodology for NoSQL databases. It relies on NoAM, an abstract data model that aims to represent NoSQL data-stores in a system-independent way. NoAM models can be implemented in several NoSQL databases, including key-value stores, document databases, and extensible record stores. Instead, we focus on generating NoSQL databases from higher-level UML models, and thus, designers do not need to learn a new language/platform. Nevertheless, NoAM could be integrated in our approach if we manage to extend it with constraint support. In that case, NoAM could be seen as a PSM derived from UML models and OCL constraints, and can be used to implement non-graph databases, which are not supported by our approach for now.

7.7 Conclusion

In this chapter we have presented the UMLTOGRAPHDB framework, a MDA-based approach to implement (UML) conceptual schemas in graph databases, including the generation of the code required to check the OCL constraints defined in the schema. Our approach is specified as a chain of model transformations that use a new intermediate GraphDB metamodel to specify graph database schemas[8], and reuses the Gremlin metamodel embedded in the MOGWAÏ framework to express database queries generated from OCL invariants.

We showed how our existing work on aligning modeling level constructs and graph database primitives can be reused to provide a functional solution enabling to design schema-less graph databases using high-level conceptual languages. UMLTOGRAPHDB is based on the implicit model-to-graph mapping embedded in NEOEMF/GRAPH to create the GraphDB instance representing the structure of the database to create, and reuses the MOGWAÏ translation approach to generate efficient Gremlin traversals enabling efficient computation of buisiness rules and data integrity constraints. A prototype implementation of UMLTOGRAPHDB supporting Neo4j and OrientDB is available on Github.

---

[8] Note that this metamodel can also be regarded as a kind of UML profile (and could be easily reexpressed as such) for graph databases

8

Conclusion

8.1 Summary

In this manuscript, we have drawn the basis of a new modeling infrastructure that aims to address the scalability issues experienced in the application of MDE techniques in industrial processes. Specifically, we have described a family of approaches that provide efficient persistence, query, and transformation of large models typically manipulated in generative MDE processes such as MDRE techniques.

We have presented NEOEMF , a multi-database persistence backend that provides a set of persistence solutions adapted to specific modeling scenarios. Our approach is based on a generic architecture that allows to easily plug a new data-store, and provides extension points that can be used to further improve performances and memory consumption of existing applications. NEOEMF is built on top of the EMF platform, and embeds three preset novel model to database mappings (graph, map, and column) that enable to benefit from the advanced capabilities of the backends.

In order to further improve the performances of *lazy-loading* model persistence solutions, we have proposed an approach aiming at integrating prefetching and caching techniques. We introduced PREFETCHML , a DSL and execution engine that allows to define prefetching and caching rules over models using a high-level declarative language. Our experiments have shown that applying prefetching and caching techniques on top of existing model persistence framework can significantly improve execution time while finely controlling the memory consumption by tuning the cache policy. A dedicated implementation of PREFETCHML is embedded in NEOEMF to further improve performances by computing prefetching instruction at the database level, bypassing the modeling API.

We have shown how the scalability issues encountered in existing model query frameworks can be addressed with a translational approach. We proposed MOGWAÏ , a novel model query solution that generates Gremlin traversals from OCL queries in order to maximize the benefits of using a NoSQL backend to store large models. OCL queries are translated using model-to-model transformation into Gremlin traversals that are then computed on the database side, reducing the overhead implied by the modeling API and the reification of intermediate objects.

Finally, we coped with the similar issues faced by the current model transformation frameworks by proposing GREMLIN-ATL , an extension of our model query solution dedicated to model transformations. We designed a new transformation engine that provides an abstraction layer to access heterogeneous data-sources and tune the transformation algorithm to fit memory and execution time requirements.

All the presented solutions have been implemented as a set of open source Eclipse plugins released under the EPL [1] license, and available online on the NEOEMF website [2].

8.2 Impact of the Results

In this thesis we have reported several improvements both in terms of execution time and memory consumption when using the presented approaches to store, access, query, and transform large models. We have shown through our experiments that these improvements are conceptually valuable to any MDE process aiming at manipulating large models. In this section we explore the specific fields where execution time and memory consumption are critical aspects, and thus could be significantly improved by our techniques.

The model@run.time approach [78] is a development method that relies on models to describe an application and its behavior. Models are then used as primary artifacts during the application execution, and are interpreted to perform specific operations. In this context, model queries and transformation are heavily used to dynamically retrieve execution information, check invariants and constraints on the fly, or represent the control flow of the application. Thus, our work presented in Chapters 5 and 6 could be an interesting solution to enhance the execution time and memory consumption of these specific operations, improving the performances of the entire application.

Interactive model manipulations are specific modeling scenarios that are typically performed by a modeler to manually check a generated model, define and validate constraints, or inspect an existing model to better understand it. In these applications, the user's experience is tightly coupled to the reactivity of the entire system, that should be able to load, navigate, and query large models efficiently. In this sense, our solutions constitute an improvement compared to the state of the art tools, and constitute a first step towards a fully interactive model edition platform managing large models.

Finally, the reported results based on existing MDRE applications show that our approaches are interesting solutions to improve the support of large models in such processes. As introduced in Chapter 2, these processes are typically facing execution time and memory consumption issues when dealing with large applications to analyze and refactor. In addition, MDRE processes are typically iterative approaches where a modeler defines and refines a set of modeling steps to perform (including model store, queries, and transformations), and performs some interactive model manipulation to check the intermediate results. In this context, enhancing the support of large models could have a

---

1. `https://www.eclipse.org/legal/epl-v10.html`
2. `www.neoemf.com`

significant impact on the process computation as well as its maintainance when dealing with large models.

8.3 Perspectives and Future Work

Modeling issues currently encountered in industrial scenarios are one of the consequences of the increasing amount and diversity of data to model. Indeed, we believe that the new generation of connected devices (such as the well known Internet of Things (IoT) [119]) as well as the development of open data programs [53] and cloud-based computing will further emphasize the need to provide efficient modeling techniques to represent, store, and query these complex systems.

In this section we present the perspectives and future work of our solutions that could be explored to try to deal with this new generation of modeling techniques and their inherent scalability requirements. Then, we enlarge the discussion to the possible applications of our work outside the MDE ecosystem, and we conclude with some thoughts on the available channels to disseminate our contributions and improve their visibility.

8.3.1 Model Storage Scalability

In the context of NEOEMF , we plan to study the integration of new data-stores and evaluate how specific data representation techniques and database implementations (such as in-memory key-value stores or temporal databases [100]) behave in specific modeling scenarios. Such a study would complement the one proposed by Shah et al. [98] on benchmarking NoSQL data-stores for large models.

We also want to improve our multi-database architecture by allowing designers to use and combine multiple databases at the same time to store a model. Indeed, typical modeling scenarios are usually composed of different processes (such as interactive model edition, model queries, transformations) that can be efficiently handled by different data-store. Integrating them would allow to dynamically select the one that fit a given modeling task, however, this integration raises performance and consistency issues that need to be addressed.

We are also exploring how collaborative modeling techniques can be integrated in our approach. Indeed, while automatic model generation techniques are usually single-user tasks performed in an homogeneous context (same tools / family of tools), they usually constitute the first step of complex processes such as software modernization scenarios, that require multiple modelers and developers manipulating, querying, and updating a shared set of models concurrently. The data distribution provided by the NEOEMF/GRAPH and NEOEMF/COLUMN connectors could be used as a baseline to enable collaborative modeling, but additional work is required to support multiple modeling solutions concurrently, in particular in heterogeneous modeling contexts. We plan to study how existing collaborative modeling approaches such as ModelBus[15, 101] and the CDO collaboration component[3] could be adapted to NEOEMF .

---

[3] `https://help.eclipse.org/mars/topic/org.eclipse.emf.cdo.doc/html/users/Doc08_Collaborating.html?cp=13_1_7`

Collaborative modeling usually requires additional security and access control layers to ensure that the manipulated model stay consistent and that each modeler can only access the parts of the model corresponding to its access rights. In this sense, we plan to study how we can integrate access control rule definition and computation in NEOEMF . Model view extraction frameworks such as EMF Views [20] can be an interesting solution to filter an existing metamodel according to a given access policy, but the view extraction mechanism has to be tuned to support large models and *lazy-loading* persistence solutio efficiently.

On the PREFETCHML side, we plan to work on the automatic generation of prefetching and caching scripts based on static analysis of available queries and transformations for the metamodel we are trying to optimize. Indeed, while our DSL is a first step to ease the specification of prefetching and caching plans, the creation of *good* PREFETCHML plans still implies low-level knowledge on the PREFETCHML engine as well as the expected query execution.

We also plan to perform additional experiments evaluating the user's experience when using PREFETCHML in existing modeling scenarios. Indeed, our experience on defining PREFETCHML plans to use in our experiments has shown that creating an optimized plan is not a trivial task, and implies to have a good understanding of the prefetching and caching algorithms as well as the model structure. Thus, an evaluation focused on the cost of defining a good PREFETCHML plan is needed to fully assess the benefits of the approach.

Finally, we plan to improve our monitoring component to support the creation of new prefetching rules on the fly based on model access traces. This could be done, for example, by integrating existing work aiming at improving application performances by detecting frequent access patterns in server logs [16] and machine learning techniques to automatically derive high-level prefetching and caching rule.

8.3.2 Model Query and Transformations Scalability

First, we plan to improve our translation techniques to fully support the set of constructs of their input languages. As discussed along this thesis, some constructs are not targeted by our solutions, such as the OCL tuple data type and ATL imperative code blocks. Integrating them would align our solutions with state of the art tools, and enable advanced comparisons. Furthermore, we plan to study the transformation of alternative input languages, such as EOL and QVT, and study how their common features can be generalized into a common infrastructure.

Looking at the performance of our solutions, we plan to study the impact of semantically-equivalent OCL expressions [25] on generated traversals. With this information, it could be possible to improve the quality of the generated database queries by first applying an automatic refactoring on the OCL side. Additional experiments on semantically-equivalent ATL transformations (for example using imperative blocks instead of matched rules) also has to be performed.

Finally, we plan to extend our transformation and query approaches into a family of mappings adapted to different NoSQL backends in order to provide a set of modeling frameworks able to both store and manipulate models "natively". This mapping definition could be partially generated by schema inspection techniques such as JSonDiscoverer [56] or the NoSQLDataEngineering framework [93], and would leverage the capabilities of model query and transformation languages on top of various data storage solutions.

8.3.3 On the application of our work outside the MDE

In chapter 7 we have shown how our contributions can be combined to develop UMLTOGRAPHDB , a solution aiming at bridging the gap between conceptual modeling and graph databases. This first step was primarily designed to emphasize the versatility of our tools and their usefulness outside the core MDE ecosystem, however, additional work is required to extend our solution and provide a sound infrastructure to generate data access applications from conceptual schemas.

First, we plan to align our framework with nowadays industry requirements by covering the combination of multiple database types. Indeed, existing software environments usually provide multiple data-source aiming at storing part of the application data (such as relational databases storing client records, raw text files containing application logs, or document database representing the business logic). To handle these heterogeneous infrastructures, we plan to support conceptual schema fragmentation between several databases. This requires a mechanism to evaluate constraints over several persistence solutions and query languages. Apache Drill [50] or Hibernate OGM [71] could be reused for this.

We also study the integration of refactoring operations on top of the PSM model generated by our approach to allow designers to tune the data representation according to specific needs, such as query execution performance or memory consumption. While providing these operations would allow fine grain tuning of the generated applications, we need to define a set of refactorings that does not break the alignment between the GraphDB models and the conceptual schema. In addition, the transformation generating the data consistency checks should be aware of these refactorings.

Our solutions could be also used to improve data migration processes. Indeed, the model mapping developed along this manuscript can be seen as modeling APIs used to describe the implicit schema of an existing database. With this information, data migration operations can be expressed using high-level model transformation languages such as ATL, hiding low-level details to the modeler and reducing the technical cost. In this sense, we performed preliminary experiments in our existing work [] that shows that the impact in terms of performance for a simple case study is acceptable compared to the gains in terms of readability. Still, additional experiments are required to assess the benefits and drawbacks of the approach.

Finally, these model to database mappings could be reused in the context of the PREFETCHML framework, in order to define prefetching and caching instruction on top of multiple data-sources, even if they do not explicitly contain a model. This could be interesting in the context of NoSQL databases, which typically lack advanced prefetching components [79].

8.3.4 Dissemination

The NeoEMF framework is one of the model persistence solutions embedded in the MONDO platform [69], an European project aiming at improving scalability of MDE solutions. In the future, we plan to continue to push our solutions in projects that could benefit from our advanced scalability features, such as the MegaMart2 European project[4], that aims to provide a scalable model-based framework for continuous development and runtime validation of complex systems.

On the technical side, we plan to continue our work on presenting NeoEMF related technologies in demonstration and tutorial sessions during the modeling conferences. In addition, our website is regularly updated with the latest releases and changelog, and several discussion channels are available to let users and designers interact with us.

Finally, we believe that integrating gamification techniques [41] in our approaches could be an interesting solution to improve their adoption by end users. Recent work on gamifying the learning of modeling techniques through the Papyrus platform [30] are a promising start in this direction, that could be extended to the learning of advanced concepts such as PrefetchML plan definition and NeoEMF configuration.

---

4. `https://megamart2-ecsel.eu/`

List of Terms

ATL AtlanMod Transformation Language. 17, 23, 102–108, 110–115, 117–121, 129, 130, 133, 140, 141

CDO Connected Data Objects model repository. 33, 46–48, 53, 79, 83, 84, 89, 96, 139

DSL Domain Specific Language. 17, 18, 23, 26, 52, 54, 64, 66, 67, 79, 84, 103, 105, 132, 133, 137, 140

EMF Eclipse Modeling Framework. 14, 17, 26, 29, 32–34, 36, 37, 40, 46, 48, 49, 52, 53, 78, 79, 82–84, 89, 96–99, 103–106, 109, 115, 118, 119, 123, 137
EMOF Essential MOF. 26
EOL Epsilon Object Language. 34, 81–84, 86, 104, 140
ER Entity Relationship. 123
ETL Epsilon Transformation Language. 103–105, 121

HQL Hibernate Query Language. 33

IoT Internet of Things. 139

JPA Java Persistence API. 86, 87
JSON JavaScript Object Notation. 26, 27
JVM Java Virtual Machine. 46, 47, 97

KDM Knowledge Discovery Model. 107, 108, 118, 119
KMF Kevoree Modeling Framework. 26, 36

LRU Least Recently Used. 51

MDA Model Driven Architecture. 21, 101, 124, 125, 135
MDE Model-Driven Engineering. 13–17, 21–23, 29, 31, 43, 54, 81, 101, 104, 123, 137–139, 141, 142
MDRE Model Driven Reverse Engineering. 13, 14, 18, 32, 101, 121, 137, 138
MDT Model Development Tools. 25, 83, 84, 89, 95, 96, 98
MOF MetaObject Facility. 23, 24, 26
MQT Model Query Translator. 82, 83, 96
MRU Most Recently Used. 51, 74

OCL Object Constraint Language. 15, 17, 18, 23–25, 54, 55, 58, 59, 62, 64, 66, 68, 72, 73, 75, 79, 81–86, 89, 91–99, 107, 108, 111, 115, 124–126, 130–135, 137, 140

OGM Object/Grid Mapper. 86

OMG Object Management Group. 13, 15, 21, 23, 24, 26, 54, 82, 86, 99, 101, 103, 107, 124

ORM Object-Relational Mapping. 33

PIM Platform Independent Model. 101, 124, 125

PSM Platform Specific Model. 101, 124–126, 134, 141

QVT Query/View/Transformation. 23, 102–106, 118, 121, 140

RDBMS Relational Database Management System. 16, 26, 27, 31–33, 83, 86

RDF Resource Description Framework. 26

SQL Structured Query Language. 28, 32, 33, 83, 84, 86, 96, 134

UML Unified Modeling Language. 13, 22–26, 83, 96, 105, 123–126, 128, 129, 131, 133–135

URI Unified Resource Identifier. 55

XMI XML Metadata Interchange. 23, 32–34, 36, 46, 47, 75, 118, 119

XML EXtensible Markup Language. 15, 23, 31, 36

Contents

1 Context 13
- 1.1 Introduction . . . 13
- 1.2 Problem Statement . . . 14
- 1.3 Approach . . . 16
- 1.4 Contributions . . . 16
- 1.5 Outline of thesis . . . 17
- 1.6 Scientific Production . . . 19
- 1.7 Awards . . . 20

2 Background 21
- 2.1 Model-Driven Engineering . . . 21
  - 2.1.1 Models, Metamodels, Model Transformations . . . 22
- 2.2 MDE Standards and Technologies . . . 23
  - 2.2.1 UML/OCL . . . 23
  - 2.2.2 Modeling Frameworks . . . 25
- 2.3 NoSQL Databases . . . 26
  - 2.3.1 Key-Value Stores . . . 27
  - 2.3.2 Document Databases . . . 27
  - 2.3.3 Column Databases . . . 28
  - 2.3.4 Graph Databases . . . 28
- 2.4 Conclusion . . . 29

3 Scalable Model Persistence 31
- 3.1 State of the Art . . . 32
  - 3.1.1 Relational Persistence Layers . . . 32
  - 3.1.2 NoSQL Persistence Layers . . . 33
  - 3.1.3 Problematic & Requirements . . . 34
- 3.2 NeoEMF: a Multi-Database Persistence Framework . . . 35

3.2.1 Architectural Overview 36
3.2.2 Integration in the Modeling Ecosystem 36
3.2.3 Advanced Capabilities 38
3.3 Model-to-Database Mappings 39
3.3.1 NeoEMF/Graph 39
3.3.2 NeoEMF/Map 40
3.3.3 NeoEMF/Column 43
3.4 Tooling 44
3.5 Empirical Evaluation 46
3.5.1 Benchmark Presentations 46
3.5.2 Results 47
3.5.3 Discussion 47
3.6 Conclusion 49

4 Model Prefetching and Caching 51
4.1 State of the Art 52
4.1.1 Prefetching and Caching in Current Modeling Frameworks 52
4.1.2 Problematic and Requirements 53
4.2 The PrefetchML DSL 54
4.2.1 Abstract Syntax 55
4.2.2 Concrete Syntax 56
4.2.3 Running Example 58
4.3 PrefetchML Framework Infrastructure 59
4.3.1 Architecture 59
4.3.2 Rule Processing 62
4.3.3 Cache Consistency 64
4.3.4 Global shared cache 65
4.4 Plan Monitoring 66
4.4.1 Language Extensions for Plan Monitoring 66
4.4.2 Simple Monitoring 67
4.4.3 Adaptative Monitoring 69
4.5 Tool Support 70
4.5.1 Language Editor 70
4.5.2 EMF Integration 71
4.5.3 NeoEMF/Graph Integration 72

4.6 Evaluation 72
  4.6.1 Benchmark Presentation 73
  4.6.2 Results 75
  4.6.3 Discussion 78
4.7 Conclusions 79

5 Efficient Queries 81
5.1 State of the Art 82
  5.1.1 Model Query Solutions 82
  5.1.2 Summary and Research Problem 85
5.2 The Mogwaï Framework 86
  5.2.1 The Gremlin Query Language 86
  5.2.2 The Mogwaï Query Approach 89
  5.2.3 Gremlin Metamodel 90
  5.2.4 Mapping of OCL expressions 91
  5.2.5 Transformation process 92
5.3 Tooling 95
5.4 Evaluation 96
  5.4.1 Benchmark presentation 97
  5.4.2 Results 97
  5.4.3 Discussion 97
5.5 Conclusion 99

6 Efficient Transformations 101
6.1 State of the Art 103
  6.1.1 Model Transformation frameworks 103
  6.1.2 Towards Scalable Model Transformations 104
  6.1.3 Summary and Research Problem 106
6.2 Gremlin-ATL Framework 106
  6.2.1 The ATL Transformation Language 106
  6.2.2 Framework Overview 109
6.3 ATLtoGremlin Transformation 110
  6.3.1 ATL Operations Mapping 110
  6.3.2 Operation Composition 112
6.4 Execution Environment 114

6.4.1 Model Mapping . . . 115
6.4.2 Transformation Helper . . . 115
6.5 Tool Support . . . 117
6.6 Evaluation . . . 118
6.6.1 Benchmark Presentation . . . 118
6.6.2 Results . . . 119
6.6.3 Discussion . . . 119
6.7 Conclusion . . . 121

7 A NeoEMF/Mogwaï Infrastructure to Generate Database Software 123
7.1 UMLtoGraphDB Approach . . . 124
7.2 Mapping UML Class Diagram to GraphDB . . . 126
7.2.1 GraphDB Metamodel . . . 127
7.2.2 Class2GraphDB Transformation . . . 128
7.3 Translating OCL Expressions to Gremlin . . . 130
7.4 Code Generation . . . 131
7.4.1 Blueprints API . . . 131
7.4.2 Graph2Code Transformation . . . 132
7.5 Tool Support . . . 133
7.6 Related Work . . . 134
7.7 Conclusion . . . 135

8 Conclusion 137
8.1 Summary . . . 137
8.2 Impact of the Results . . . 138
8.3 Perspectives and Future Work . . . 139
8.3.1 Model Storage Scalability . . . 139
8.3.2 Model Query and Transformations Scalability . . . 140
8.3.3 On the application of our work outside the MDE . . . 141
8.3.4 Dissemination . . . 142

Appendices 163

A NeoEMF Tutorial 167

List of Tables

3.1 Property Map 42
3.2 Type Map 42
3.3 Containment Map 42
3.4 Examples instance stored as a sparse table in NeoEMF/Column 44
3.5 Software metadata 45
3.6 Benchmarked Models 47
3.7 ClassAttributes Results in milliseconds (Large VM / Small VM) 47
3.8 SingletonMethods Results in milliseconds (Large VM / Small VM) 47
3.9 InvisibleMethods Results in milliseconds (Large VM / Small VM) 48
3.10 UnusedMethods Results in milliseconds (Large VM / Small VM) 48

4.1 Experimental Set Details 75
4.2 NeoEMF/Graph Query Execution Time in milliseconds 76
4.3 NeoEMF/Map Query Execution Time in milliseconds 77
4.4 Multiple Query Execution Time in milliseconds 77

5.1 Features of existing query solutions 85
5.2 OCL to Gremlin mapping 93
5.3 Query Intermediate Loaded Objects and Result Size per Model 97
5.4 Query Results on Set 2 98
5.5 Query Results on Set 3 98

6.1 Features of existing transformation solutions 105
6.2 ATL to Gremlin mapping 112
6.3 Model Mapping API 116
6.4 Benchmarked Models 119
6.5 AbstractTypeDeclaration2DataType Results 120
6.6 Java2KDM Results 120

7.1 OCL to Gremlin mapping 131

List of Figures

1 Legacy System Modernization using MDRE Techniques 7
2 NeoEMF Modeling Ecosystem 9

1.1 Legacy System Modernization using MDRE Techniques 15
1.2 NeoEMF Modeling Ecosystem 18

2.1 Modernization process of legacy applications 22
2.2 A Simple Java Metamodel 24
2.3 A Simple Instance 25

3.1 NEOEMF Integration in the Modeling Ecosystem 37
3.2 NEOEMF Integration in EMF Infrastructure 38
3.3 Running example persisted in NEOEMF/GRAPH 41

4.1 Prefetch Abstract Syntax Metamodel 55
4.2 PREFETCHML Integration in MDE Ecosystem 60
4.3 PREFETCHML Framework Infrastructure 61
4.4 PREFETCHML Initialization Sequence Diagram 63
4.5 PREFETCHML Access Event Handling Sequence Diagram 63
4.6 PrefetchML Update Event Handling Sequence Diagram 65
4.7 PREFETCHML Abstract Syntax Metamodel with Monitoring Extensions . 67
4.8 PrefetchML Rule Editor 71
4.9 Overview of EMF-Based Prefetcher 72
4.10 Overview of NeoEMF-Based Prefetcher 73

5.1 Comparison of OCL execution 90
5.2 Extract of Gremlin Metamodel 91
5.3 OCL Syntax Tree 94
5.4 Generated Gremlin Syntax Tree 95

6.1 Model Transformation Engine and Modeling Framework Integration . . . 102

6.2 A Simple Java Metamodel 108
6.3 Overview of the GREMLIN-ATL Framework 109
6.4 ATLtoGremlin Transformation Overview 110
6.5 GREMLIN-ATL Execution Environment 114

7.1 Conceptual Model to Graph database 124
7.2 Overview of the UMLTOGRAPHDB Infrastructure 125
7.3 Class Diagram of a Simple e-commerce Application 126
7.4 GraphDB Metamodel 127
7.5 Excerpt of the Mapped GraphDB Model 129
7.6 Generated Infrastructure 132

Bibliography

[1] R. Ansorg and L. Schwabe. Domain-Specific Modeling as a Pragmatic Approach to Neuronal Model Descriptions. In *Brain Informatics*, pages 168–179. Springer, 2010. 5, 13, 22

[2] S. Azhar. Building Information Modeling (BIM): Trends, Benefits, Risks, and Challenges for the AEC Industry. *Leadership and Management in Engineering*, 11(3):241–252, 2011. 5, 6, 13, 14, 22

[3] K. Barmpis and D. Kolovos. Hawk: Towards a Scalable Model Indexing Architecture. In *Proceedings of the 1st BigMDE Workshop*, pages 6–9. ACM, 2013. 32, 34, 53, 84

[4] K. Barmpis and D. S. Kolovos. Comparative Analysis of Data Persistence Technologies for Large-scale Models. In *Proceedings of the 1st XM Workshop*, pages 33–38. ACM, 2012. 33

[5] C. Bauer and G. King. *Hibernate in Action*. Manning Greenwich CT, 2005. 33, 86

[6] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4EMF, a Scalable Persistence Layer for EMF Models. In *Proceedings of the 10th ECMFA Conference*, pages 230–241. Springer, 2014. 39, 52

[7] A. Benelallam, A. Gómez, M. Tisi, and J. Cabot. Distributed Model-to-Model Transformation with ATL on MapReduce. In *Proceedings of the 8th SLE Conference*, pages 37–48. ACM, 2015. 43, 104, 106

[8] A. Benelallam, M. Tisi, J. S. Cuadrado, J. De Lara, and J. Cabot. Efficient Model Partitioning for Distributed Model Transformations. In *Proceedings of the 9th SLE Conference*, pages 226–238. ACM, 2016. 104

[9] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. Viatra 3: A Reactive Model Transformation Platform. In *Proceedings of the 8th ICMT Conference*, pages 101–110. Springer, 2015. 84

[10] G. Bergmann, A. Horváth, I. Ráth, and D. Varró. Incremental Evaluation of Model Queries over EMF Models: A Tutorial on EMF-IncQuery. In *Proceedings of the 7th ECMFA Conference*, pages 389–390. Springer, 2011. 81, 84

[11] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental Evaluation of Model Queries Over EMF Models. In *Proceedings of the 13th MoDELS Conference*, pages 76–90. Springer, 2010. 22, 53, 73, 84, 104

[12] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. 103, 133

[13] J. Bézivin. On the Unification Power of Models. *Software and Systems Modeling*, 4(2):171–188, 2005. 22

[14] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective Model Driven Engineering. In *Proceedings of the 6th «UML» Conference*, pages 175–189. Springer, 2003. 21

[15] X. Blanc, M.-P. Gervais, and P. Sriplakich. Model bus: Towards the interoperability of modelling tools. In *Model Driven Architecture: MDAFA 2003 and MDAFA 2004 Selected Papers*, pages 17–32. Springer, 2005. 139

[16] C. Bouras, A. Konidaris, and D. Kostoulas. Predictive Prefetching on the Web and its Potential Impact in the Wide Area. *World Wide Web*, 7(2):143–179, 2004. 140

[17] M. Brambilla and J. Cabot. Constraint Tuning and Management for Web Applications. In *Proceedings of the 6th ICWE Conference*, pages 345–352, 2006. 83, 85, 134

[18] C. Brun and A. Pierantonio. Model Differences in the Eclipse Modeling Framework. *The European Journal for the Informatics Professional*, 9(2):29–34, 2008. 47, 119

[19] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014. 5, 7, 9, 13, 15, 32, 82, 101

[20] H. Bruneliere, J. G. Perez, M. Wimmer, and J. Cabot. Emf Views: A view Mechanism for Integrating Heterogeneous Models. In *Proceedings of the 34th ER Conference*, pages 317–325. Springer, 2015. 140

[21] Bryan Hunt. MongoEMF, 2017. URL: `https://github.com/BryanHunt/mongo-emf/`. 33, 87

[22] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone. Database Design for NoSQL Systems. In *Proceedings of the 33rd ER Conference*, pages 223–231. Springer, 2014. 134

[23] L. Burgueño, M. Wimmer, and A. Vallecillo. A Linda-Based Platform for the Parallel Execution of Out-Place Model Transformations. *Information and Software Technology*, 79:17–35, 2016. 105

[24] J. Cabot and E. Teniente. Constraint Support in MDA Tools: A Survey. In *Proceedings of the 2nd ECMDA-FA Conference*, pages 256–267. 2006. 83

[25] J. Cabot and E. Teniente. Transformation Techniques for OCL Constraints. *Science of Computer Programming*, 68(3):179 – 195, 2007. 91, 140

[26] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. *ACM SIGMETRICS Performance Evaluation Review*, pages 188–197, 1995. 51

[27] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4, 2008. 28

[28] Z. Cheng, R. Monahan, and J. F. Power. A Sound Execution Semantics for ATL via Translation Validation. In *Proceedings of the 8th ICMT Conference*, pages 133–148. Springer, 2015. 107

[29] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica*, pages 311–336, 1986. 74

[30] V. Cosentino, S. Gérard, and J. Cabot. A Model-based Approach to Gamify the Learning of Modeling. In *Proceedings of the 5th SCME Symposium*. CEUR-WS, 2017. 142

[31] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA-Visual Automated Transformations for Formal Verification and Validation of UML models. In *Proceedings of the 17th ASE Conference*, pages 267–270. IEEE, 2002. 104

[32] J. S. Cuadrado, E. Guerra, and J. de Lara. Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In *Proceedings of the 25th ISSRE Symposium*, pages 34–44. IEEE, 2014. 107

[33] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. In *ACM SIGMOD Record*, pages 257–266. ACM, 1993. 51

[34] G. Daniel, G. Sunyé, A. Benelallam, and M. Tisi. Improving Memory Efficiency for Processing Large-Scale Models. In *Proceedings of the 2nd BigMDE Workshop*, pages 31–39. CEUR-WS, 2014. 52

[35] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot. NeoEMF: a Multi-Database Model Persistence Framework for Very Large Models. *Science of Computer Programming*, 2017. 35, 51, 52, 53, 62, 81, 83, 95

[36] G. Daniel, G. Sunyé, and J. Cabot. PrefetchML: a Framework for Prefetching and Caching Models. In *Proceedings of the 19th MoDELS Conference*, pages 318–328. ACM/IEEE, 2016. 66, 74

[37] X. De Carlos, G. Sagardui, A. Murguzur, S. Trujillo, and X. Mendialdua. Model Query Translator: A Model-Level Query Approach for Large-Scale Models. In *Proceedings of the 3rd MODELSWARD Conference*, pages 62–73. IEEE, 2015. 82

[38] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. 28

[39] B. Demuth and H. Hussmann. Using UML/OCL Constraints for Relational Database Design. In *Proceedings of the 2nd UML Conference*, pages 598–613. Springer, 1999. 134

[40] B. Demuth, H. Hußmann, and S. Loecher. OCL as a Specification Language for Business Rules in Database Applications. In *Proceedings of the 4th UML Conference*, pages 104–117. Springer, 2001. 83, 134

[41] S. Deterding, M. Sicart, L. Nacke, K. O'Hara, and D. Dixon. Gamification. Using Game-Design Elements in Non-Gaming Contexts. In *Proceedings of the 29th CHI Conference*, pages 2425–2428. ACM, 2011. 142

[42] Eclipse Foundation. MDT OCL. URL: `www.eclipse.org/modeling/mdt/?project=ocl`. 25

[43] Eclipse Foundation. The CDO Model Repository (CDO), 2017. URL: `http://www.eclipse.org/cdo/`. 7, 15, 31, 33, 51, 52, 53, 62, 81, 83

[44] M. Eysholdt and H. Behrens. Xtext: Implement your Language Faster than the Quick and Dirty Way. In *Proceedings of the 25th OOPSLA Conference*, pages 307–309. ACM, 2010. 56

[45] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial intelligence*, 19(1):17–37, 1982. 84

[46] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *Proceedings of the 15th MoDELS Conference*, pages 87–101. Springer, 2012. 26, 36

[47] A. Gómez, A. Benelallam, and M. Tisi. Decentralized Model Persistence for Distributed Computing. In *Proceedings of the 3rd BigMDE Workshop*, pages 42–51. CEUR-WS, 2015. 35, 43

[48] A. Gómez, G. Sunyé, M. Tisi, and J. Cabot. Map-Based Transparent Persistence for Very Large Models. In *Proceedings of the 18th FASE Conference*, pages 19–34. Springer, 2015. 7, 15, 35, 40

[49] T. Hartmann, A. Moawad, F. Fouquet, G. Nain, J. Klein, and Y. Le Traon. Stream my Models: Reactive Peer-to-Peer Distributed models@run.time. In *Proceedings of the 18th MoDELS Conference*, pages 80–89. IEEE, 2015. 53

[50] M. Hausenblas and J. Nadeau. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data*, 1(2):100–104, 2013. 141

[51] F. Heidenreich, C. Wende, and B. Demuth. A Framework for Generating Query Language Code from OCL Invariants. *Electronic Communications of the EASST*, 9, 2007. 83, 85

[52] F. Holzschuher and R. Peinl. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J. In *Proceedings of the Joint EDBT/ICDT Workshops*, pages 195–204, 2013. 87

[53] N. Huijboom and T. Van den Broek. Open Data: an International Comparison of Strategies. *European journal of ePractice*, 12(1):4–16, 2011. 101, 139

[54] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-Driven Engineering Practices in Industry. In *Proceedings of the 33rd ICSE Conference*, pages 633–642. IEEE, 2011. 5, 14, 51, 104

[55] J. Hutchinson, J. Whittle, and M. Rouncefield. Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors that Lead to Success or Failure. *Science of Computer Programming*, 89:144–161, 2014. 6, 14, 21, 31, 101, 104

[56] J. L. C. Izquierdo and J. Cabot. JSONDiscoverer: Visualizing the Schema Lurking Behind JSON Documents. *Knowledge-Based Systems*, 103:52–55, 2016. 5, 13, 123, 141

[57] I. Jacobson. *Object-Oriented Software Engineering: a Use Case Driven Approach*. Pearson Education India, 1993. 23

[58] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1):31 – 39, 2008. 23, 81, 95, 101, 103, 106, 129

[59] F. Jouault and I. Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of the 21st SAC Conference*, pages 1188–1195. ACM, 2006. 106

[60] K. Kaur and R. Rani. Modeling and Querying Data in NoSQL Databases. In *Proceedings of the 2nd Big Data Congress*, pages 1–7. IEEE, 2013. 27

[61] S. Kent. Model Driven Engineering. In *Proceedings of the 3rd IFM Conference*, pages 286–298. Springer, 2002. 21

[62] A. C. Klaiber and H. M. Levy. An Architecture for Software-Controlled Data Prefetching. In *ACM SIGARCH Computer Architecture News*, pages 43–53. ACM, 1991. 51

[63] M. Koegel and J. Helming. EMFStore: a Model Repository for EMF Models. In *Proceedings of the 32nd ICSE Conference*, pages 307–308. ACM, 2010. 34, 51

[64] D. S. Kolovos, R. F. Paige, and F. Polack. Merging Models with the Epsilon Merging Language (EML). In *Proceedings of the 9th MoDELS Conference*, volume 6, pages 215–229. Springer, 2006. 84, 104

[65] D. S. Kolovos, R. F. Paige, and F. A. Polack. Eclipse Development Tools for Epsilon. In *Proceedings of the Eclipse Summit Europe, Eclipse Modeling Symposium*, page 200, 2006. 26, 104

[66] D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Object Language (EOL). In *Proceedings of the 2nd ECMDA-FA Conference*, pages 128–142. Springer, 2006. 34

[67] D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Transformation Language. *Proceedings of the 1st ICMT Conference*, 8:46–60, 2008. 23, 84, 103

[68] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, et al. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proceedings of the 1st BigMDE Workshop*, pages 1–10. ACM, 2013. 6, 14, 32, 51, 101

[69] D. S. Kolovos, L. M. Rose, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, G. Sunyé, and M. Tisi. MONDO: Scalable Modelling and Model Management on the Cloud. In *Proceedings of the Projects Showcase (STAF 2015)*, pages 44–53, 2015. 44, 142

[70] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier. Papyrus UML: an Open Source Toolset for MDA. In *Proceedings of the 5th ECMDA-FA Conference*, pages 1–4, 2009. 6, 14

[71] A. Leonard. *Pro Hibernate and MongoDB*. Apress, 2013. 141

[72] Y. Li, P. Gu, and C. Zhang. Transforming UML Class Diagrams into HBase Based on Meta-Model. In *Proceedings of the 4th ISEEE Conference*, volume 2, pages 720–724. IEEE, 2014. 134

[73] D. Lucrédio, R. P. d. M. Fortes, and J. Whittle. MOOGLE: A Model Search Engine. In *Proceedings of the 11th MoDELS Conference*, pages 296–310. Springer, 2008. 73

[74] E. Marcos, B. Vela, and J. M. Cavero. A Methodological Approach for Object-Relational Database Design Using UML. *Software & System Modeling*, 2(1):59–72, 2003. 134

[75] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema Design for NoSQL Applications. In *Proceedings of the 32nd ICDE Conference*. IEEE, 2016. 134

[76] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani. MDE Adoption in Industry: Challenges and Success Criteria. In *Proceedings of Workshops at MoDELS 2008*, pages 54–59. Springer, 2009. 5, 14, 51

[77] A. Moniruzzaman and S. A. Hossain. NoSQL Database: New Era of Databases for Big Data Analytics - Classification, Characteristics and Comparison. *International Journal of Database Theory and Application*, 6(4):1–14, 2013. 27

[78] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg. Models@ run. time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009. 138

[79] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov. Security Issues in NoSQL Databases. In *Proceedings of the 10th TrustCom Conference*, pages 541–547. IEEE, 2011. 123, 141

[80] OMG. MDA Specifications, 2016. URL: `http://www.omg.org/mda/specs.htm`. 21, 101, 124

[81] OMG. Knowledge Discovery Metamodel (KDM), 2017. URL: `http://www.omg.org/technology/kdm/`. 107, 118

[82] OMG. Meta Object Facility (MOF) Specifications, 2017. URL: `http://www.omg.org/spec/MOF/2.4.1/`. 23

[83] OMG. OCL Specification, 2017. URL: `www.omg.org/spec/OCL`. 15, 23, 24, 81, 107, 124

[84] OMG. OMG MOF 2 XMI Mapping Specification version 2.5.1, 2017. URL: `http://www.omg.org/spec/XMI/2.5.1/`. 23, 32

[85] OMG. QVT Specification, 2017. URL: `http://www.omg.org/spec/QVT`. 23, 103, 106

[86] OMG. UML Specification, 2017. URL: `www.omg.org/spec/UML`. 22, 23, 124

[87] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Proceedings of the 14th MoDELS Conference*, pages 77–92. Springer, 2011. 7, 15

[88] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. A Repository for Scalable Model Management. *Software & Systems Modeling*, 14(1):219–239, 2015. 31, 33, 51, 52, 53, 62, 81, 87

[89] J. E. Pagán and J. G. Molina. Querying Large Models Efficiently. *Information and Software Technology*, 56(6):586–622, 2014. 33

[90] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. *Informed Prefetching and Caching*. ACM, 1995. 51

[91] R. Pohjonen and J.-P. Tolvanen. Automated Production of Family Members: Lessons Learned. In *Proceedings of the PLEES Conference*, pages 49–57. IESE, 2002. 5, 13

[92] V. Reniers, A. Rafique, D. Van Landuyt, and W. Joosen. Object-NoSQL Database Mappers: a benchmark study on the performance overhead. *Journal of Internet Services and Applications*, 8(1):1, 2017. 87

[93] D. S. Ruiz, S. F. Morales, and J. G. Molina. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Proceedings of the 34th ER Conference*, pages 467–480. Springer, 2015. 115, 134, 141

[94] J. Rumbaugh. OMT: The Object Model. *Journal of Object-Oriented Programming*, pages 21–27, 1995. 23

[95] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004. 13, 23

[96] M. Scheidgen, A. Zubow, J. Fischer, and T. Kolbe. Automated and Transparent Model Fragmentation for Persisting Large Models. In *Proceedings of the 15th MoDELS Conference*, pages 102–118. Springer, 2012. 34

[97] S. Sendall and W. Kozaczynski. Model transformation: The Heart and Soul of Model-Driven Software Development. *IEEE software*, 20(5):42–45, 2003. 101

[98] S. M. Shah, R. Wei, D. S. Kolovos, L. M. Rose, R. F. Paige, and K. Barmpis. A Framework to Benchmark NoSQL Data Stores for Large-Scale Model Persistence. In *Proceedings of the 17th MoDELS Conference*, pages 586–601. Springer, 2014. 35, 139

[99] A. J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, pages 223–247, 1978. 51, 52

[100] R. T. Snodgrass. Temporal Databases. In *Theories and methods of spatio-temporal reasoning in geographic space*, pages 22–64. Springer, 1992. 139

[101] P. Sriplakich. *ModelBus: un environnement réparti et ouvert pour l'ingénierie de modèles*. PhD thesis, Paris 6, 2007. 139

[102] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008. 6, 14, 26

[103] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró. The Train Benchmark: Cross-Technology Performance Evaluation of Continuous Model Queries. *Software & Systems Modeling*, pages 1–29, 2017. 9, 73, 74

[104] The Eclipse Foundation. Eclipse Marketplace, 2017. URL: `https://marketplace.eclipse.org/`. 26

[105] The Eclipse Foundation. EMF Query, 2017. URL: `https://projects.eclipse.org/projects/modeling.emf.query`. 84

[106] The Eclipse Foundation. Teneo, 2017. URL: `https://wiki.eclipse.org/Teneo`. 33

[107] Tinkerpop. Blueprints API, 2017. URL: `www.blueprints.tinkerpop.com`. 87

[108] Tinkerpop. Pipes Framework, 2017. URL: `www.pipes.tinkerpop.com`. 87

[109] Tinkerpop. The Gremlin Language, 2017. URL: `www.gremlin.tinkerpop.com`. 82, 87

[110] Tinkerpop. Tinkerpop Website, 2017. URL: `www.tinkerpop.com`. 87, 127, 132

[111] M. Tisi, F. Jouault, J. Delatour, Z. Saidi, and H. Choura. FUML as an Assembly Language for Model Transformation. In *Proceedings of the 7th SLE Conference*, pages 171–190. Springer, 2014. 107

[112] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. volume 9, pages 18–33. Springer, 2009. 107

[113] M. Tisi, S. Martinez, and H. Choura. Parallel Execution of ATL Transformation Rules. In *Proceedings of the 16th MoDELS Conference*, pages 656–672. Springer, 2013. 104, 106, 107

[114] P. Warden. *Big Data Glossary*. O'Reilly Media, Inc., 2011. 26

[115] J. Warmer and A. Kleppe. Building a Flexible Software Factory Using Partial Domain Specific Models. In *Proceedings of the 6th DSM Workshop*, pages 15–22. University of Jyvaskyla, 2006. 51

[116] R. Wei and D. S. Kolovos. An Efficient Computation Strategy for allInstances(). *Proceedings of the 3rd BigMDE Workshop*, pages 32–41, 2015. 36, 38, 84, 97, 102

[117] J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE software*, 31(3):79–85, 2014. 6, 14, 21, 31, 32, 101

[118] E. D. Willink. Local Optimizations in Eclipse QVTc and QVTr using the Micro-Mapping Model of Computation. In *Proceedings of the 2nd EXE Workshop*, pages 26–32, 2016. 23, 103

[119] F. Xia, L. T. Yang, L. Wang, and A. Vinel. Internet of Things. *International Journal of Communication Systems*, 25(9):1101, 2012. 139

[120] P. Zikopoulos, C. Eaton, et al. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 2011. 26

Appendices

A

NeoEMF Tutorial

NeoEMF Tutorial

The goal of this tutorial is to present NeoEMF through a simple example. You can download a zipped version of the project here, or import it in your Eclipse workspace using the `File->New->Example->NeoEMF->NeoEMF Tutorial Project` menu.

Introduction

In this tutorial you will create a persistent EMF resource using the Neo4j database as a backend. To do so, you will define a simple Ecore model, create instances of this model and then store these instances in the persistent EMF resource.

Audience

This tutorial is designed for the Eclipse Modeling Framework (EMF) users with a need to handle large-scale models in Java programs.

This tutorial will bring at intermediate level of expertise, where you will be able to use the Neo4j graph database to store EMF models. From this level, you can take yourself at higher level of expertise, understanding how to use NeoEMF with different databases.

Prerequisites

Before proceeding with this tutorial you should have a good understanding of EMF. If you need more information about EMF, please follow the tutorial available here.

A basic understanding of Eclipse IDE is also required because the examples have been compiled using Eclipse IDE.

There is no need to understand Neo4j not graph databases to follow this tutorial. However, a basic understanding of Neo4j may help you to manipulate your models directly from the database.

Installing NeoEMF

NeoEMF is available as an Eclipse plugin. Install it by choosing *Help* → *Install New Software…*. You will need to add the following software site:

https://atlanmod.github.io/NeoEMF/releases/latest/plugin/

Select and install all items.

Installing Emfatic

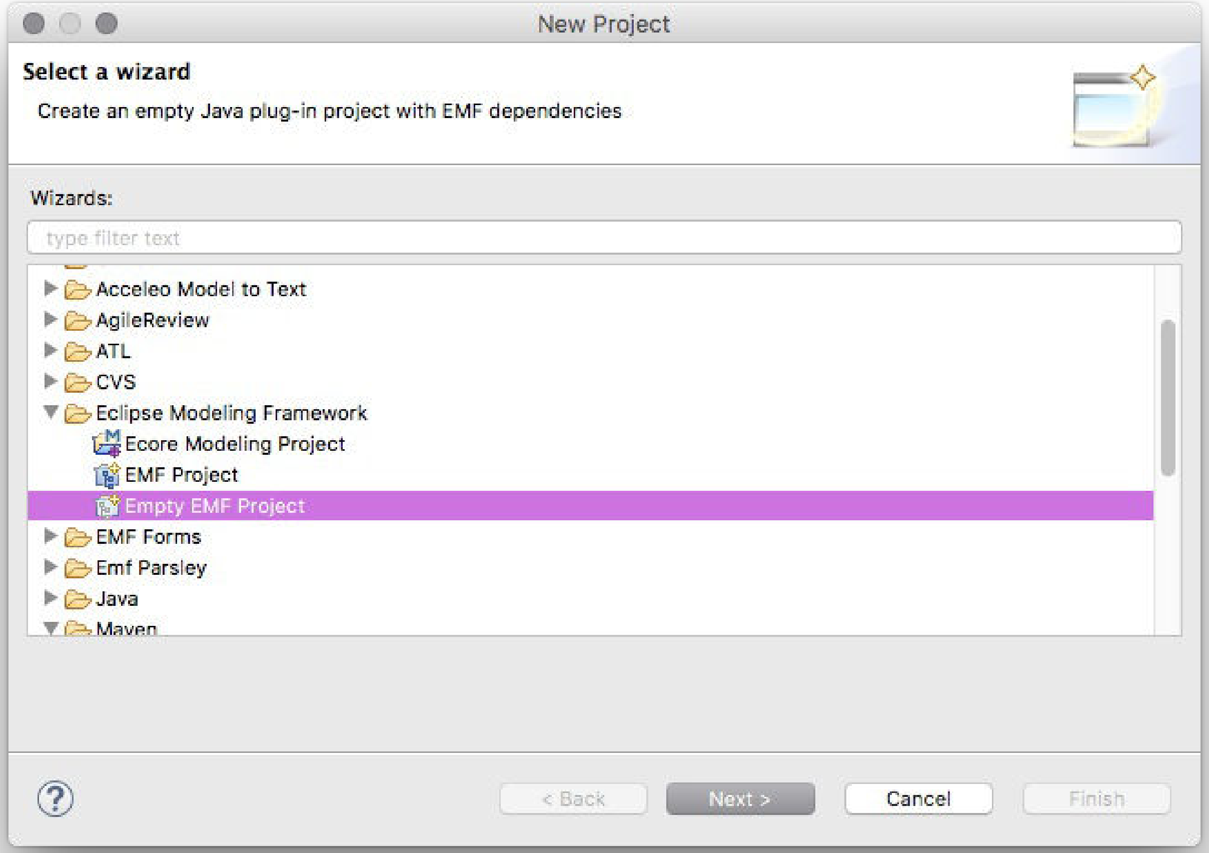
Emfatic will be used as a text editor to create a simple Ecore model. It is also available as an Eclipse plugin. Install it by choosing *Help → Install New Software…*. You will also need to add the following software site:

http://download.eclipse.org/emfatic/update

Select and install all items.

Creating a new EMF Project

Now create a new EMF project by choosing *File → New → Project…* from the main menu. The dialog offers a couple of different project types. Select *Empty EMF Project* from the category *Eclipse Modeling Framework* and continue via *Next*.



Select a wizard

Create an empty Java plug-in project with EMF dependencies

New Project
Wizards:
type filter text
Acceleo Model to Text
AgileReview
ATL
CVS
Eclipse Modeling Framework
Ecore Modeling Project
EMF Project
Empty EMF Project
EMF Forms
Emf Parsley
Java
Maven
< Back
Next >
Cancel
Finish

Feel free to use any name for youy project ("NeoEMF Tutorial" would be great), and finish the wizard.

Creating a Simple Ecore Model

To create and save an EMF resource, you first need an ECore Model. There are several ways to create an Ecore Model, here we use EMFatic, a textual syntax for ECore models.

From the main menu, choose *File→ New → Other…* and select *Emfatic file* from the category *Example EMF*

Creation Wizard. Name your file "graph.emf".

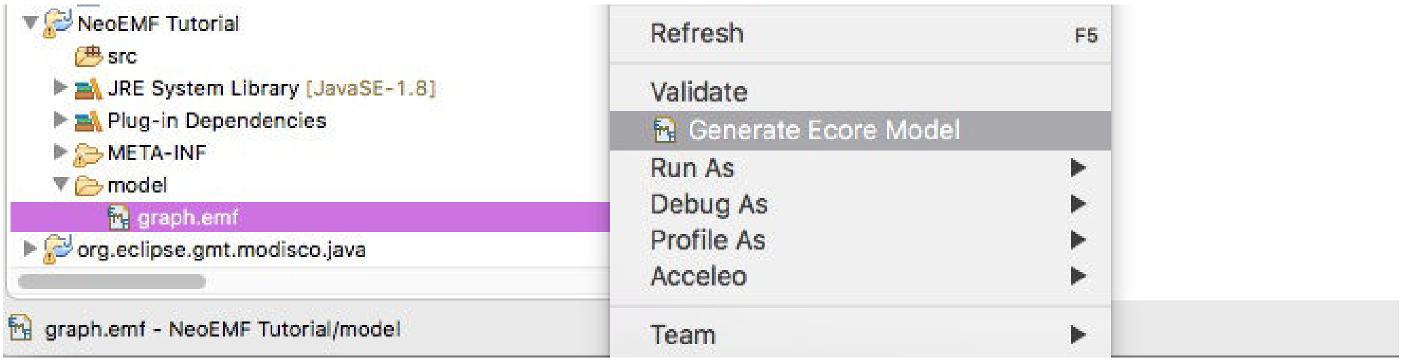
Edit your file to create a simple model specifying a simple graph structure, containing Edges and Vertices, and described below:

```
@namespace(uri="http://atlanmod.neoemf.tutorial", prefix="graph")
package graph;

class Graph {
    val Vertice[*] vertices;
    val Edge[*] edges;
}

class Vertice {
    attr String label;
}

class Edge {
    ref Vertice from;
    ref Vertice to;
}
```

An alternative textual syntax to create an Ecore Model is OclInEcore, which is quite similar to EMFatic.

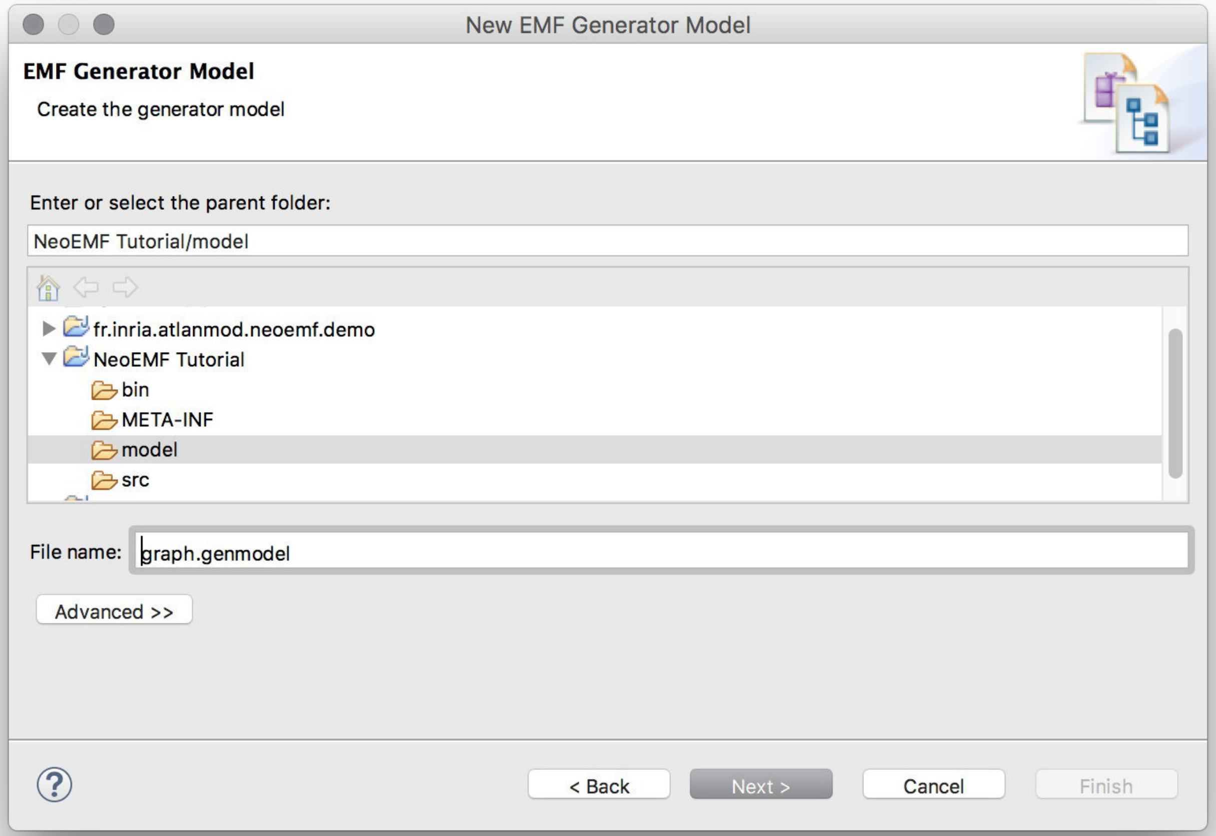
Creating an Ecore File

Once the Emfatic file is ready, you need to generate a *.ecore* file. From the contextual menu (righ-click on the *graph.emf* file), choose *Generate Ecore Model*.


NeoEMF Tutorial
src
JRE System Library [JavaSE-1.8]
Plug-in Dependencies
META-INF
model
graph.emf
org.eclipse.gmt.modisco.java
graph.emf - NeoEMF Tutorial/model
Refresh
F5
Validate
Generate Ecore Model
Run As
Debug As
Profile As
Acceleo
Team

Creating an EMF Generator Model

Now create a new EMF Generator Model by choosing *File → New → Other…* from the main menu. The dialog offers a couple of different wizards. Select *EMF Generator Model* from the category *Eclipse Modeling Framework* and continue via *Next*.

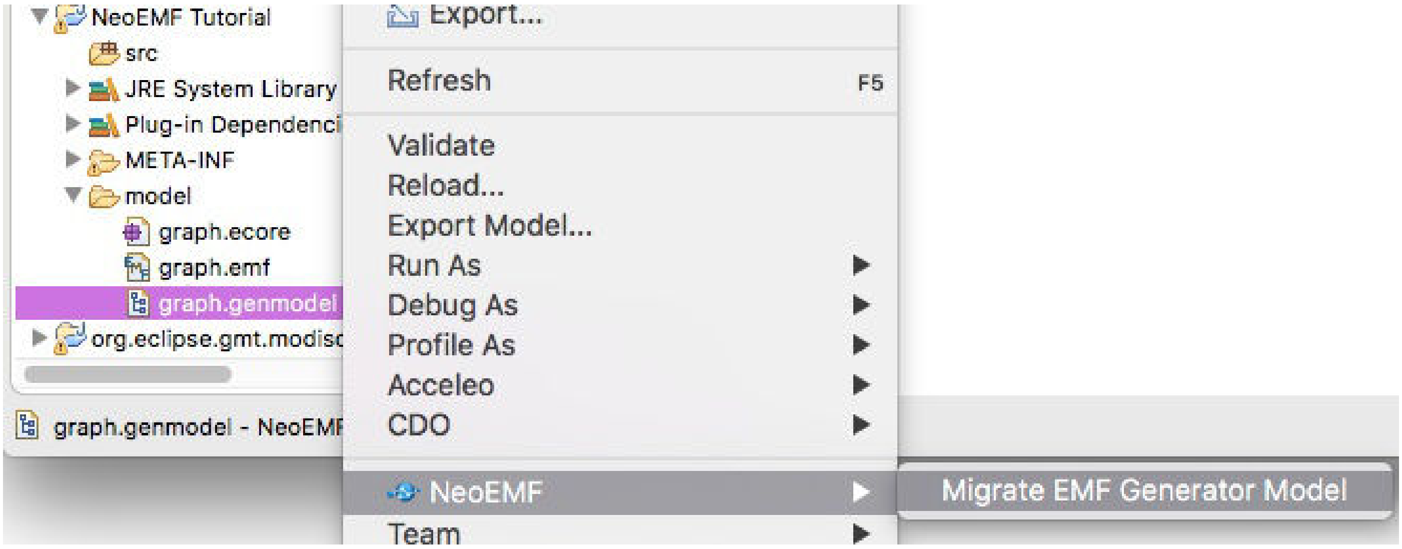

New EMF Generator Model
EMF Generator Model
Create the generator model
fr.inria.atlanmod.neoemf.demo
NeoEMF Tutorial
bin
META-INF
model
src
Advanced >>
< Back
Next >
Cancel
Finish

Enter or select the parent folder: NeoEMF Tutorial/model

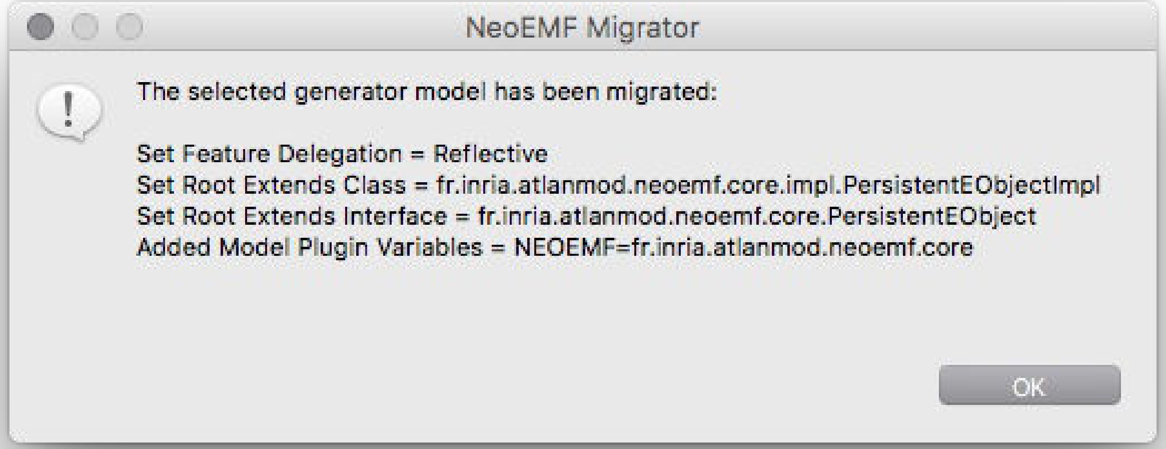
File name: graph.genmodel

Migrating the Generator Model

After generating the `graph.genmodel` file, you need to migrate it to NeoEMF. From the contextual menu, choose *NeoEMF → Migrate EMF Generator Model*.


NeoEMF Tutorial
src
JRE System Library
Plug-in Dependenci
META-INF
model
graph.ecore
graph.emf
graph.genmodel
org.eclipse.gmt.modisc
graph.genmodel - NeoEMF
Export...
Refresh F5
Validate
Reload...
Export Model...
Run As
Debug As
Profile As
Acceleo
CDO
NeoEMF
Team
Migrate EMF Generator Model

The migration will modify several properties in the `graph.genmodel` file. Basically, it will set the root Class and the root Interface of EMF Classes and Interfaces to use the NeoEMF persistent implementations.


NeoEMF Migrator

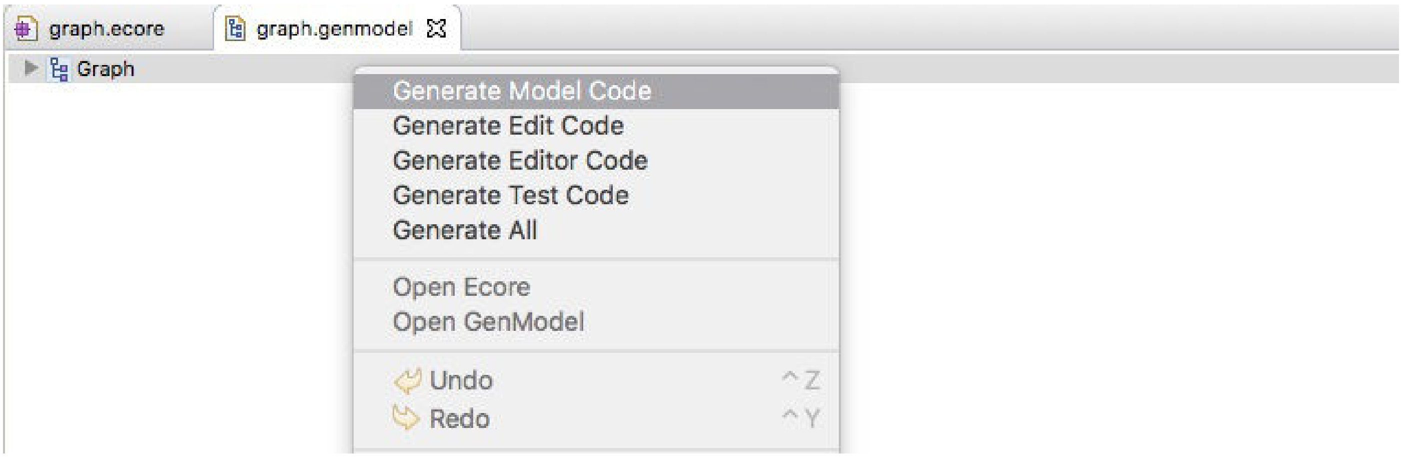
The selected generator model has been migrated:

Set Feature Delegation = Reflective
Set Root Extends Class = fr.inria.atlanmod.neoemf.core.impl.PersistentEObjectImpl
Set Root Extends Interface = fr.inria.atlanmod.neoemf.core.PersistentEObject
Added Model Plugin Variables = NEOEMF=fr.inria.atlanmod.neoemf.core

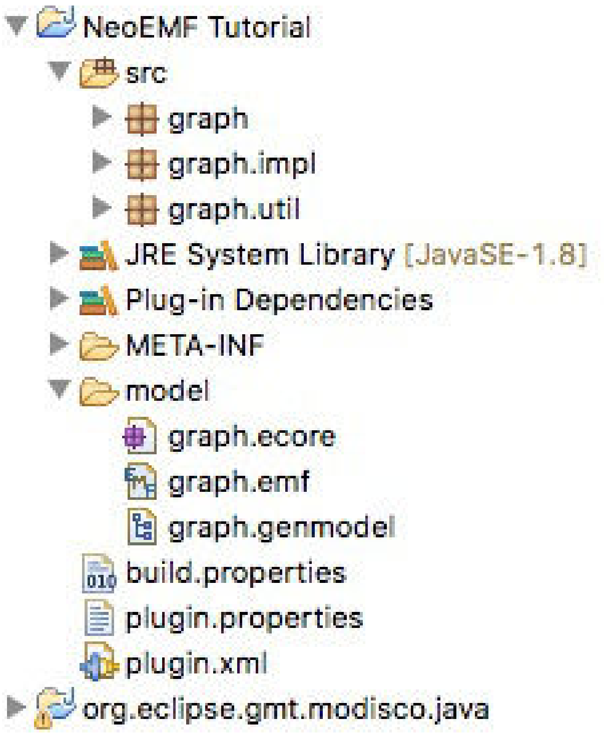
OK

Generating EMF Model Code

After generating the `graph.genmodel` file, you will be able to generate the Java underlying code for this model. Select *Generate Model Code* from the Project Explorer contextual menu (right click the `graph.genmodel` file)


graph.ecore
graph.genmodel
Graph
Generate Model Code
Generate Edit Code
Generate Editor Code
Generate Test Code
Generate All
Open Ecore
Open GenModel
Undo ^Z
Redo ^Y

The generation will add three new packages to your project. If you are familiar to the EMF generated code, you can browse the generated code to observe the differences between the defaut generated code and the NeoEMF one.



NeoEMF Tutorial
src
graph
graph.impl
graph.util
JRE System Library [JavaSE-1.8]
Plug-in Dependencies
META-INF
model
graph.ecore
graph.emf
graph.genmodel
build.properties
plugin.properties
plugin.xml
org.eclipse.gmt.modisco.java

Creating a new Neo4j EMF resource

Once the Ecore model is ready, we can create instances of this model and store them in a NeoEMF persistent resource.

Since NeoEMF can use different backends, you first need to register a persistence backend factory, which is responsible of the creation of the persistence backend that is in charge of the model storage.

Write down the following line to use the Neo4j database under the Blueprints API:

```
PersistenceBackendFactoryRegistry.register(BlueprintsURI.SCHEME,
                    BlueprintsPersistenceBackendFactory.getInstance());
```

Registering the Persistent Resource Factory

As for regular EMF initialization, you need to register a `ResourceFactory` implementation in the resource set specifying the URI protocol. This is done in two steps:

1. Create a new `ResourceSet`.
2. Register a `ResourceFactory` with the corresponding URI protocol.

In NeoEMF, each backend implementation provides a subclass of URI to ease protocol definition. Note that the associated `PersistentResourceFactory` and the created `PersistentResource` do not depend on the selected backend.

```
ResourceSet resSet = new ResourceSetImpl();
```

```
resSet.getResourceFactoryRegistry().getProtocolToFactoryMap().put(BlueprintsURI.SCHEME, PersistentResc
```

Creating a resource

Creating a resource in NeoEMF is similar to standard EMF. Write down the following line to create a resource named "models/myGraph.graphdb" in your current Eclipse project.

```
Resource resource = resSet.createResource(BlueprintsURI.createFileURI(new File("models/myGraph.graphdb
```

Populating the resource

Now, write a simple code to create instances of the Graph model and to save the resource:

```
GraphFactory factory = GraphFactory.eINSTANCE;
Graph graph = factory.createGraph();
for (int i = 0; i < 100; i++) {
    Vertice v1 = factory.createVertice();
    v1.setLabel("Vertice " + i + "a");
    Vertice v2 = factory.createVertice();
    v2.setLabel("Vertice " + i + "b");
    Edge e = factory.createEdge();
    e.setFrom(v1);
    e.setTo(v2);
    graph.getEdges().add(e);
    graph.getVertices().add(v1);
    graph.getVertices().add(v2);
}
resource.getContents().add(graph);
resource.save(BlueprintsNeo4jOptionBuilder.newBuilder().asMap());
```

Reading the resource

```
resource.load(BlueprintsNeo4jOptionBuilder.newBuilder().asMap());
Graph graph = (Graph) resource.getContents().get(0);
for (Edge each : graph.getEdges()) {
    System.out.println(each.getFrom().getLabel() + "--->" + each.getTo().getLabel());
}
```

Creating EMF Resources on a MapDB Database

The process for registering a MapDB persistence backend is similar the one presented above.

Write down the following code to create a persistent EMF resource using MapDB:

```
PersistenceBackendFactoryRegistry.register(MapDbURI.SCHEME,
                                           MapDbPersistenceBackendFactory.getInstance());

resSet.getResourceFactoryRegistry().getProtocolToFactoryMap()
  .put(MapDbURI.SCHEME,PersistentResourceFactory.getInstance());
Resource mapResource = resSet.createResource(MapDbURI
 .createFileURI(new File("models/myGraph.madb")));
```

First, an instance of the MapDB factory is registered in NeoEMF. Then, NeoEMF factory is registered in the EMF Resource Set. Finally, a resource named "myGraph.mapdb" is created in folder "models" of the current Eclipse project.

Conclusion

In this tutorial, you have learned how to create a persistent EMF resource with NeoEMF and how to store this resource in a Neo4j database.

UNIVERSITE
BRETAGNE
LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

Gwendal DANIEL

Persistance, Requêtage, et Transformation Efficaces de Grands modèles

Efficient Persistence, Query, and Transformation of Large Models

Résumé

L'Ingénierie Dirigée par les Modèles (IDM) est une méthode de développement logicielle ayant pour but d'améliorer la productivité et la qualité logicielle en utilisant les modèles comme artefacts de premiers plans durant le processus développement. Dans cette approche, les modèles sont typiquement utilisés pour représenter des vues abstraites d'un système, manipuler des données, valider des propriétés, et sont finalement transformés en ressources applicatives (code, documentation, tests, etc). Bien que les techniques d'IDM aient montré des résultats positifs lors de leurs intégrations dans des processus industriels, les études montrent que la mise à l'échelle des solutions existantes est un des freins majeurs à l'adoption de l'IDM dans l'industrie. Ces problématiques sont particulièrement importantes dans le cadre d'approches génératives, qui nécessitent des techniques efficaces de stockage, requêtage, et transformation de grands modèles typiquement construits dans un contexte mono-utilisateur. Plusieurs solutions de persistance, requêtage, et transformations basées sur des bases de données relationnelles ou NoSQL ont été proposées pour améliorer le passage à l'échelle, mais ces dernières sont souvent basées sur une seule sérialisation model/base de données, adaptée à une activité de modélisation particulière, mais peu efficace pour d'autres cas d'utilisation. Par exemple, une sérialisation en graphe est optimisée pour calculer des chemins de navigations complexes, mais n'est pas adaptée pour accéder à des valeurs atomiques de manière répétée. De plus, les frameworks de modélisations existants ont été initialement développés pour gérer des activités simples, et leurs APIs n'ont pas évolué pour gérer les modèles de grande taille, limitant les performances des outils actuels. Dans cette thèse nous présentons une nouvelle infrastructure de modélisation ayant pour but de résoudre les problèmes de passage à l'échelle en proposant (i) un framework de persistance permettant de choisir la représentation bas niveau la plus adaptée à un cas d'utilisation, (ii) une solution de requêtage efficace qui délègue les navigations complexes à la base de données stockant le modèle, bénéficiant de ses optimisations bas niveau et améliorant significativement les performances en terme de temps d'exécution et consommation mémoire, et (iii) une approche de transformation de modèles qui calcule directement les transformations au niveau de la base de données. Nos solutions sont construites en utilisant des standards OMG tels que UML et OCL, et sont intégrées dans les solutions de modélisations majeures telles que ATL ou EMF.

Mots clés

IDM, Passage à l'Echelle, Requêtage de Modèle, Transformation de Modèle, NoSQL, OCL, Gremlin

Abstract

The Model Driven Engineering (MDE) paradigm is a software development method that aims to improve productivity and software quality by using models as primary artifacts in all the aspects of software engineering processes. In this approach, models are typically used to represent abstract views of a system, manipulate data, validate properties, and are finally transformed to application artifacts (code, documentation, tests, etc).

Among other MDE-based approaches, automatic model generation processes such as Model Driven Reverse Engineering are a family of approaches that rely on existing modeling techniques and languages to automatically create and validate models representing existing artifact. Model extraction tasks are typically performed by a modeler, and produce a set of views that ease the understanding of the system under study.

While MDE techniques have shown positive results when integrated in industrial processes, the existing studies also report that scalability of current solutions is one of the key issues that prevent a wider adoption of MDE techniques in the industry. This is particularly true in the context of generative approaches, that require efficient techniques to store, query, and transform very large models typically built in a single-user context.

Several persistence, query, and transformation solutions based on relational and NoSQL databases have been proposed to achieve scalability, but they often rely on a single model-to-database mapping, which suits a specific modeling activity, but may not be optimized for other use cases. For example a graph-based representation is optimized to compute complex navigation paths, but may not be the best solution for repeated atomic accesses. In addition, low-level modeling framework were originally developed to handle simple modeling activities (such as manual model edition), and their APIs have not evolved to handle large models, limiting the benefits of advance storage mechanisms.

In this thesis we present a novel modeling infrastructure that aims to tackle scalability issues by providing (i) a new persistence framework that allows to choose the appropriate model-to-database mapping according to a given modeling scenario, (ii) an efficient query approach that delegates complex computation to the underlying database, benefiting of its native optimization and reducing drastically memory consumption and execution time, and (iii) a model transformation solution that directly computes transformations in the database. Our solutions are built on top of OMG standards such as UML and OCL, and are integrated with the *de-facto* standard modeling solutions such as EMF and ATL.

Key Words

MDE, Scalability, Model Queries, Model Transformations, NoSQL, OCL, Gremlin