Model transformation on distributed platforms : decentralized persistence and distributed processing

Amine Benelallam

UNIVERSITE BRETAGNE LOIRE

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Amine BENELALLAM

Mémoire présenté en vue de l'obtention du

grade de Docteur de l'École nationale supérieure des mines de Nantes

Label européen

sous le sceau de l'Université Bretagne Loire

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27

Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 07 Décembre 2016

Thèse n° : 2016EMNA0288

Model transformation on distributed platforms

decentralized persistence and distributed processing

JURY

Président :	**M. Jean-Claude ROYER**, Professeur, Mines Nantes
Rapporteurs :	**M. Antonio VALLECILLO**, Professeur, University of Malaga
	M. Antoine BEUGNARD, Professeur, Telecom Bretagne
Examinateurs :	**M. Soichiro HIDAKA**, Professeur, Hosei University,Tokyo
	M. Massimo TISI, Maître Assistant, Mines Nantes
Invité :	**M. Gerson SUNYÉ**, Maître de Conférence HDR, Université de Nantes
Directeur de thèse :	**M. Jordi CABOT**, Professeur, Open University of Catalonia

Acknowledgement

Foremost, I would like to express my sincere gratitude to my advisors Prof. Jordi Cabot, Dr. Massimo Tisi, and Dr. Gerson Sunyé for their support, patience, motivation, but more importantly their enthusiasm and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Antonio Vallecillo, Prof. Antoine Beugnard, Prof. Soichiro Hidaka , and Prof. Jean-Claude Royer for their insightful comments and encouragement, also for the questions which incited me to widen my research from various perspectives.

My sincere gratitude goes to Prof. Juan de Lara and Dr. Jesús Sanchez Cuadraro, who provided me an opportunity to join their team as intern, and gave me access to the laboratory and research facilities.

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. A special thanks goes to Dr. Abel Gómez for his valuable help and technical support. Finally, I would like to thanks Gwendal Daniel together with Dr. Abel Gómez for the joyful moments that we spent working together in the NeoEMF project.

Dedication

To my beloved parents

Ahmed Benelallam and Fatima Hergale whose affection, love, encouragement, and prays make me able to persist

To my beloved brothers and sister

Imane, Imade, and Mohamed for their support throughout the process

To my beloved fiancée

Sara for being there for me throughout the hardest part of the doctorate program

To my friends

for being by my side during these last three years

To W3

For their understanding, and their warm welcome every time I come back home

With love,

Amine,

Résumé Français

Introduction

Dans toutes les disciplines scientifiques, la **modélisation** est le processus de simplification des aspects pertinents d'un système. En informatique, plus particulièrement dans l'ingénierie et le développement logiciel, les modèles sont utilisés sur une grande étendue pour décrire un système. Ces modèles peuvent représenter d'une manière précise sa structure, sa logique ou/et son comportement. C'est grâce aux langages de modélisation que les modèles sont décrits d'une façon structurée et qui respecte un ensemble de règles bien définies. Le langage UML (Unified Modelling Language) [1] en est un exemple répandu.

L'ingénierie Dirigée par les Modèles (IDM) est l'approche qui favorise l'utilisation des modèles pour améliorer l'ingénierie des systèmes et leurs applications. Au cours de la dernière décennie, les technologies IDM ont offert une pratique pour accompagner le développement logiciel tout au long de son cycle de vie, étape par étape. Son but principal est de passer de la spécification d'un système à son exécutabilité. Ceci est concédé par les concepts de base de l'IDM qui sont la centralité des modèles (logiciel, données et système) et leurs traitements automatisés dans toutes les phases du cycle de vie de l'ingénierie logiciel. La transformation et la persistance efficace de ces modèles sont des opérations clés pour permettre cette automatisation.

La transformation de modèles (TM) est le processus de traduction d'un ou de plusieurs modèles source à un ou plusieurs modèles cible. Il existe différents langages de transformation de modèles conçus dans la perspective d'aider les utilisateurs à décrire des opérations de manipulation de modèles d'une manière intuitive. Ils fournissent des concepts adéquats pour définir des transformations modulaires et réutilisables. Les transformations de modèles peuvent être utilisées pour différentes intentions, par exemple, la semi-automatisation des configurations, la vérification et la génération de code. Finalement, les modèles sont stockés et interchangés entre les différents outils de modélisation au travers d'une couche de persistance.

Description de la problématique

Grâce à sa promesse de réduire les efforts de développement et de la maintenance des logiciels, l'IDM attire de plus en plus d'acteurs industriels. En effet, elle a été adoptée avec succès dans plusieurs domaines tels que le génie civil, l'industrie automobile et la modernisation de logiciels. Toutefois, la taille croissante des modèles utilisés nécessite de

1. http://www.omg.org/spec/UML/

concevoir des solutions passant à l'échelle afin de les traiter (transformer) et de les stocker (persister) de manière efficace.

Dans le cadre de cette thèse, nous nous concentrons principalement sur les problèmes de la mise à l'échelle des techniques de transformation des modèles et de leur persistance :

Mise à l'échelle de l'exécution des transformations de modèles — Il est possible d'implémenter des transformations de modèles distribuées à l'aide des langages de programmation à usage général et un des modèles de programmation distribuée, tels que MapReduce ou Pregel. Cependant, un tel processus n'est pas trivial. D'autant plus que, la programmation distribuée nécessite de se familiariser avec la théorie de la concurrence et de la distribution. Ceci n'est pas très commun chez les développeurs d'applications IDM. En outre, ce paradigme introduit toute une nouvelle classe d'erreurs, soit liée à la synchronisation des tâches ou bien à l'accès aux données partagées. Enfin, la programmation distribuée implique une analyse complexe pour l'optimisation des performances comme par exemple l'équilibrage de la charge de calcul et la maximisation de la localité des données.

Mise à l'échelle de la persistance des modèles — Bien qu'il existe plusieurs solutions pour persister les modèles, la plupart ne sont toujours pas à l'écoute des exigences du marché industriel. D'une part, les solutions utilisant la représentation XML pour stocker des modèles ne permettent pas de partiellement charger ou décharger un modèle. Par conséquent, la taille des modèles qu'ils peuvent gérer est limitée par la capacité mémoire de la machine utilisée. D'autre part, les solutions reposant sur des bases de données SQL ou NoSQL ne sont pas appropriées pour supporter des scénarios de modélisation spécifiques, en particulier, la transformation distribuée des modèles. Nous argumentons que ces solutions n'offrent pas une bonne flexibilité permettant l'équilibrage de charge et le partitionnement de données.

Contributions

Une façon de pallier cette problématique est d'utiliser les systèmes et les bases de données répartis. D'une part, les paradigmes de programmation distribuée tels que MapReduce et Pregel peuvent simplifier la distribution de transformations des modèles. Et d'autre part, l'avènement des bases de données NoSQL permet le stockage efficace des modèles d'une manière distribuée.

Dans cette thèse, nous apportons les contributions suivantes :

1. *Un framework réparti pour la transformation des modèles dans le Cloud* — Nous proposons une approche pour la distribution automatique de l'exécution des transformations de modèles écrites dans un langage déclaratif de transformation de modèle, ATL, en dessus d'un modèle de programmation distribuée, MapReduce. Nous montrons que, grâce au haut niveau d'abstraction de ces langages déclaratifs, leur sémantique d'exécution peut être alignée avec le paradigme MapReduce afin de livrer un moteur distribué de transformations de modèles. La distribution est implicite et

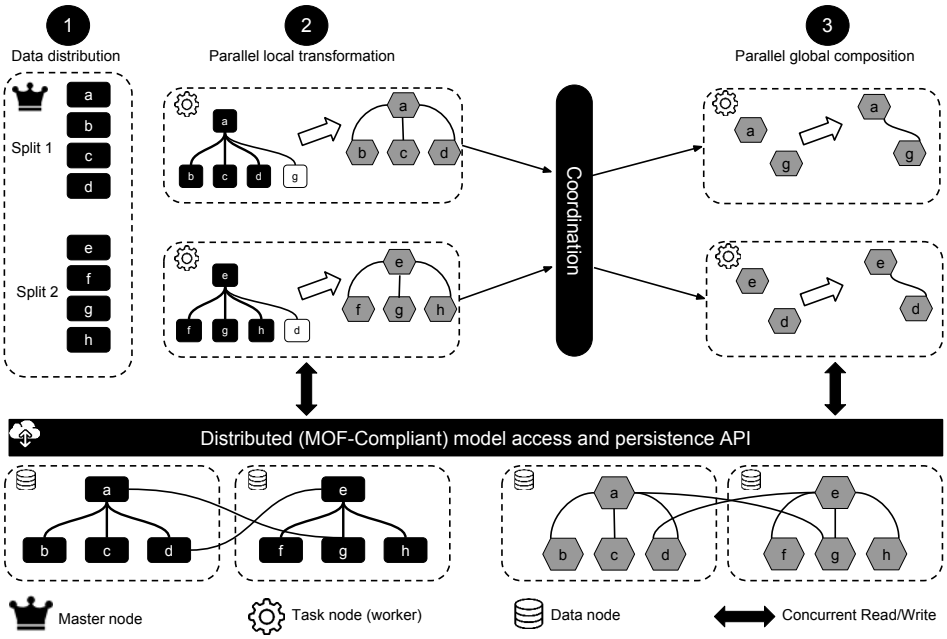


Figure 1 – Aperçu global de notre framework de transformations distribuées

la syntaxe du langage n'est pas modifiée (aucune primitive de parallélisation n'est ajoutée).

2. *Une approche efficace de partitionnement de modèles pour les transformations distribuées* — Nous formalisons le problème de partitionnement des modèles pour les transformations distribuées en programmation linéaire binaire et proposons un algorithme inspiré des résultats récents sur le partitionnement équilibré des graphes-continus. Le processus commence par l'analyse statique des règles de transformation afin d'extraire des informations sur les patterns d'accès aux modèles. Une fois cette information extraite, elle permet à l'algorithme de décider instantanément à quelle partition un élément de modèle doit être assigné. Pour valider notre approche, nous l'appliquons à notre moteur distribué de transformation de modèles.
3. *Un framework décentralisé de persistance de modèles* – Nous étendons une solution existante pour la persistance des modèles, NeoEMF. Nous exploitons les capabilités des bases de données NoSQL, afin d'équiper NEOEMF avec la capacité de manipuler et de stocker les modèles d'une manière décentralisée. NeoEMF est aussi livré avec un ensemble de stratégies pour améliorer les performances, notamment, le chargement paresseux des éléments de modèles. En outre, Nous analysons la sémantique d'ATL, en particulier, les opérations atomiques de manipulation de modèles afin d'exposer l'ensemble minimal de propriétés ACID garantissant la cohérence et la consistance des modèles cible. Finalement, nous montrons comment NEOEMF supporte ces propriétés.

Framework de transformation distribuée dans le Cloud

Nous proposons une approche de parallélisation des transformations de modèles selon un schéma de distribution de données. Ce schéma nous semble le plus approprié comme les calculs de graphes sont souvent pilotés par les données et dictés par la structure du

graphe. La Figure 1 illustre un aperçu global de notre framework de distribution par le moyen d'un simple exemple. La transformation est composée d'une règle unique qui modifie la forme des noeuds (de Carré en Hexagone), mais conserve leur identité ainsi que la topologie du graphe.

Le framework est constitué de deux briques essentielles, un moteur distribué de transformation et un framework de persistance décentralisée. Le cluster est composé d'un noeud maître (master), de nœuds de données (data nodes) et de noeuds de calcul (task nodes). Le master est responsable de partitionner le modèle et de l'envoyer aux noeuds de calcul. Ces derniers effectuent la transformation en deux phases. La première phase correspond à la transformation locale des éléments assignés (la phase map), tandis que la deuxième est responsable de composer les résultats de l'étape précédente (sous la forme de sous-ensemble du modèle cible). C'est grâce aux nœuds de données que les noeuds de calcul sont capables de communiquer et de stocker leur résultat.

Phase de distribution — Au début de cette phase, le noeud master analyse le code de la transformation afin de partitionner le modèle source d'une manière équitable et efficace. Chacun de ces splits contient un sous-ensemble du modèle source pour traitement. Ce processus repose sur un algorithme qui, en se basant sur l'analyse de code de la transformation, décide instantanément à quelle noeud de calcul il est préférable qu'un élément soit assigné.

Phase map — Bien que, chaque noeud de calcul transforme seulement le sous-ensemble assigné, il dispose d'une vue complète sur les modèles en entrée, au cas où il ait besoin d'éléments supplémentaires pour effectuer le calcul. Dans cette phase, chaque noeud charge la transformation, puis reçoit l'ensemble des éléments source lui sont attribués, un par un. Pour chaque élément, il vérifie s'il satisfait la garde de l'une des règles et génère les éléments cible correspondants à cette règle. Par la suite, l'algorithme instancie les liaisons entre les éléments cible qu'il a généré localement. En ce qui concerne les éléments non locaux, l'algorithme génère plutôt des liens vers les éléments source. Ce lien va être substitué par les éléments cible auxquels il correspond à la phase de composition.

Phase reduce — A la fin de la phase précédente, tous les éléments cible sont créés, les liaisons locales sont faites et les liaisons non-locales font référence aux éléments source afin d'être résolues. Le noeud master distribue les liaisons non résolues, et chaque noeud de calcul se charge de résoudre l'ensemble des liaisons lui sont affectées. L'algorithme parcourt toutes les liaisons non-locales, lis les éléments source et les remplace par les éléments cible correspondants. A la fin de cette phase, tous les sous-ensembles du modèle cible sont composés et le modèle global est généré.

Outils et Résultats

Nous validons notre approche en implémentant un moteur distribué des transformations de modèles en ATL sur MapReduce (ATL-MR). Notre moteur est basé d'un côté, sur Hadoop, l'implémentation open source de MapReduce et fourni par Apache, et d'un autre côté, sur EMFTVM [125], une machine virtuelle ATL.

Dans la machine virtuelle standard, le moteur de transformation parcourt l'ensemble des règles et recherche les éléments correspondants à sa condition d'application (garde). Dans notre implémentation, la machine virtuelle itère les différents éléments du modèle d'entrée, et pour chaque élément elle vérifie s'il est apparié par une règle existante.

Nous étendons la machine virtuelle avec un ensemble de méthodes permettant l'exécution en deux modes différents, local ou distribué. Notre implémentation adopte des bonnes pratiques de conception pour le traitement de graphes dans MapReduce, à savoir les patrons `InMapperCombiner` et `Shimmy` proposés par Lin et al. [88].

Nous fournissons notre moteur avec NEOEMF/COLUMN, une couche de persistance distribuée au-dessus de HBase. ATL-MR exploite les fonctionnalités mises en œuvre dans NEOEMF/COLUMN pour distribuer le stockage des modèles et permettre la lecture et l'écriture simultanée des modèles.

Nous démontrons la mise à l'échelle de notre outil en l'évaluant dans différentes études de cas bien connues. Nous montrons que ATL-MR est jusqu'à 6X plus rapide que la VM standard quand exécuté sur un cluster de 8 machines. De plus, nous démontrons qu'en utilisant notre algorithme de distribution de modèles, nous améliorons la localité des données jusqu'à 16%. Nos outils sont open-source et disponibles en ligne [2] [3]. Finalement, un cluster Docker est mis à la disposition des utilisateurs, non seulement pour utiliser, tester et découvrir notre framework, mais aussi pour reproduire nos experiences [4]. Les instructions pour déployer notre cluster sont aussi disponibles en ligne [5].

2. ATL-MR: `https://github.com/atlanmod/ATL_MR/`
3. NEOEMF: `https://www.neoemf.com`
4. Images Docker: `https://hub.docker.com/u/amineben/`
5. Builds:`https://github.com/atlanmod/hadoop-cluster-docker`

1

Context

1.1 Introduction

In all scientific disciplines, modeling is the simplification of relevant aspects of a situation in the real world for its systematic study. It is recognized to fulfill many purposes w.r.t. a particular intention (e.g. descriptivity, simulation, etc.) by means of abstraction. In computer science, especially software engineering and development, models are widely used to describe different aspects of a system under development. Models can represent the structure, logic, and behavior of a system. Thanks to modeling languages, models are consistently expressed in a structure respecting a set of well-defined rules. A typical example is the Unified Modeling Language (UML) [1] suite.

Model-Driven Engineering (MDE) is the approach that promotes the use of models to enhance systems and applications engineering. Over the past decade, MDE technologies have been offering a systematic way to walk software through its whole development cycle, step by step. Its main intent is to cross over from the specification of a system to its executability. This is granted by the core concepts of MDE which are the centrality of models (software, data, and system) and their automated processing in all phases of a software engineering lifecycle. Models represent abstract views of the system. They are designed for a specific purpose (e.g. *interoperability*) in order to answer to a specific class of questions. Figure 1.1 shows some of the activities performed in MDE.

The MDE has been successfully embraced in several domains for automating software development and manufacturing maintainable solutions while decreasing cost and effort. A proof is the plethora of tools and prototypes built around the Eclipse Modeling Framework (EMF) ecosystem, and the significant number of groups that are working on MDE. For instance, recent work have shown benefits of MDE in applications for the construction industry [137] (for communication of building information and interoperation with different tools and actors), modernization of legacy systems [27] (for aiding the migration of large

1. http://www.omg.org/spec/UML/

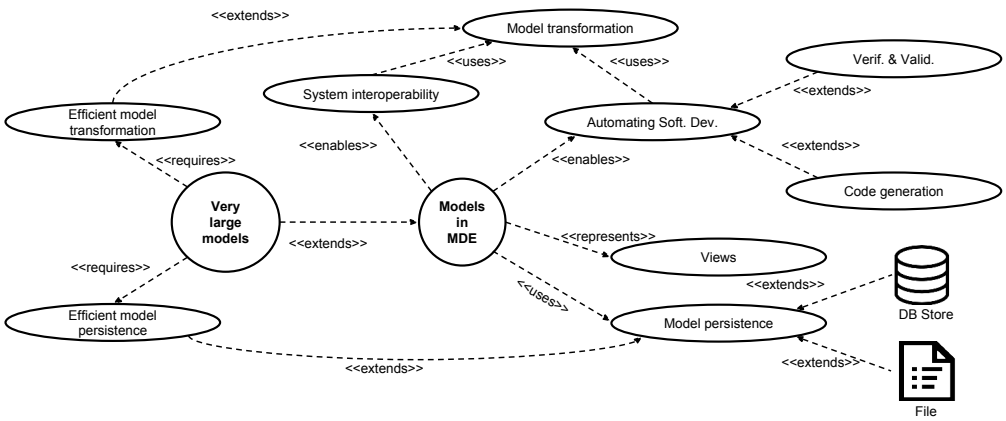


Figure 1.1 – Overview of MDE: activities and requirements

legacy codebases into novel versions meeting particular requirements), learning and big data analytics [46] (for reducing the expertise necessary to implement probabilistic models for machine learning, and speed up development). AUTomotive Open System ARchitecture (AUTOSAR) used in the development of automotive software systems, and Building Information Modeling (BIM) are two successful standards involving MDE development principles in their software life-cycle for more than ten years and twenty years respectively. Model Query and Transformation (MQT) are key operations to guarantee these benefits.

A model transformation operation is the process of translating one model to another. A complex transformation written in one of the General-Purpose Languages (GPLs), such as Java, can be extremely large and unmaintainable. Fortunately, MQT languages come to the rescue, having been designed to help users in specifying and executing model-graph manipulation operations efficiently. They provide adequate constructs and facilities to specify modular and reusable transformations with less effort. Model transformations can be subject to design, implementation, reuse, and evolution. For instance, in AUTOSAR, model transformations are used for different intents: to name a few, semi-automatization of configurations, verification & validation, and code generation. Finally, models in MDE are stored and interchanged between MDE-based modeling tools by means of persistence mediums.

1.2 Problem Statement

Despite all the promises, the MDE approach is not able yet to attract large-scale industrial systems. This is due to the serious scalability issues that the current generation of MDE tools is facing [11, 81, 62]. Indeed, according to existing empirical assessment from industrial companies adopting MDE [11, 51], the lack of tools and technologies supporting collaboration and scalability are the substantial disadvantages in MDE. Such large systems are witnessing a growing complexity in design, as well as a large amount of data. For example, BIM [137] contains a rich set of concepts (more than eight hundred) for modeling different aspects of physical facilities and infrastructures. A building model in BIM is typically made of several gigabytes of densely interconnected graph nodes.

Moreover, Laasko et al. in their review [86] of the history of the IFC standard, have shown how the language was evolving in time when integrating different aspects and stakeholders. Likewise for AUTOSAR [47, 48]. Unluckily, such enrichment comes at a price. Therefore, there is a calling need to develop a new generation of tools capable of coping with these issues.

Modeling tools are built around the so-called modeling frameworks that provide basic model-management functionalities and interoperability to the modeling ecosystem. Among the frameworks currently available, EMF has become the *de facto* standard for building modeling tools, In particular, tools built around EMF count in more than 350 [2], and more than 50 are in effect mature, stable, and official Eclipse projects. In particular, Query/View/Transformation (QVT)-d [3] and AtlanMod Transformation Language (ATL) [4] are the most widespread examples of MT languages and engines in Eclipse. The relational paradigm is the most popular among MT languages, based on the declarative definition of rules relating input and output model elements. Relational model transformations use tracing mechanisms to instate these relations. However, being based on graph matching and traversing techniques, MTs are facing serious scalability issues as graphs grow in scale and complexity. And accordingly, graph processing problems can exceed the resources (memory and CPU) of a single machine.

In this thesis, we mainly focus on enabling scalability in model transformation and persistence:

Model transformation scalability — Distributed model transformations may be implemented by using a GPL and one of the popular distributed programming models such as MapReduce [44] or Pregel [91]. However such development is not trivial, especially since distributed programming requires familiarity with concurrency and distribution theory that is not common among MDE application developers, in particular when the set of possible execution paths can be large, and transformation result can be non-deterministic. Distributed programing also introduces a completely new class of errors w.r.t. sequential programming linked to task synchronization and shared data access. Finally it entails complex analysis for performance optimization, for instance, balancing computation load, and maximizing data locality.

Model persistence scalability — While several solutions to persisting EMF models exist, most of them are still not lined up with nowadays requirements of industrial companies. On the one hand, solutions using XML-like representation to store models do not allow partial model load and unload. Hence, the size of the models they can handle is limited by the memory size. On the other hand, solutions relying on SQL or NoSQL stores are not appropriate for handling particular modeling scenarios. For instance, relational databases perform badly when queries involve multiple joins. In modeling scenarios, this translates to executing a complex traversal. A particular scenario of interest to us is distributed computation, in particular distributed model transformation. We argue that existing solutions do not offer a good flexibility to achieve common practices and requirements in distributed

2. According to the Eclipse market place https://marketplace.eclipse.org
3. https://git.eclipse.org/r/mmt/org.eclipse.qvtd
4. http://eclipse.org/atl/

computing such as load balancing and data partitioning. In this thesis, we emphasize on this particular scenario.

To summarize, in this thesis, we argue that the growth in data and complexity that is being experienced by the industry, is ahead of the current generation of model transformation and persistence tools. This hampers the adoption of MDE in industrial contexts. Therefore a new generation of MQT engines and model persistence frameworks should be provided.

Our overall objectives are:

1. *To develop an MT language and engine enabling MDE developers to specify and execute distributed model transformation in an efficient and reliable manner*

   (a) The syntax of the MT language should not be modified, no primitives for distribution should be added and distribution should be implicit. Moreover, the transformation engine should hide all the complexities related to distributed systems. Finally, the transformation engine should transparently support the integration with existing tools in the EMF ecosystem

   (b) The transformation engine should efficiently distribute the workload across the cluster nodes in a way to improve the performance and maximize data locality

2. *To develop a model persistence framework that enables the efficient persistence of Very Large Models (VLMs).* The framework should support the loading and unloading of very large models. It also should be suitable for different modeling activities, especially, distributed model transformations. Finally, it should transparently support the integration with existing tools in the EMF ecosystem

1.3 Approach

One way to overcome the scalability issues associated with model transformations, is exploiting the recent emergence of systems and programming paradigms for parallelizing model transformation operations and persistence. This is made convenient by the recent wide availability of distributed clusters in the Cloud, as well as the broad variety of distributed programming models. Such frameworks are provided with the complete package for developing distributed applications. MapReduce is one of the well-known programming models. It is designed to simplify data processing on Large Clusters. It comes with a bunch of libraries designated to ensure a good level of reliability, and fault-tolerance.

Relational languages like ATL, thanks to their level of abstraction, can be provided with semantics for implicit distributed execution. By relying on the well-defined computational model of such frameworks, their semantics can be aligned with the to the execution semantics of the transformation language. The task of managing all the complexities related to synchronization and coordination between different workers will then be handed to the underlying programming model.

On the other hand, to cope with the scalability issues related to model persistence, we propose a multi-backend model persistence framework that couples state-of-the-art NoSQL stores. Each store is suitable for a specific modeling scenario. In this thesis we

focus on distributed applications, in particular, distributed relational model transformations. Moreover, we rely on some techniques such as caching and lazy-loading to break down the complexity of VLMs.

1.4 Contributions

This thesis provides the following contributions:

1. *A distributed framework for model transformation in the Cloud* — We propose an approach to automatically distribute the execution of model transformations written in a popular MT language, ATL, on top of a well-known distributed programming model, MapReduce. We show how the execution semantics of ATL can be aligned with the MapReduce computation model. We describe the extensions to the ATL transformation engine to enable distribution, and we experimentally demonstrate the scalability of this solution in a reverse-engineering scenario. We demonstrate the effectiveness of the approach by making an implementation of our solution publicly available and using it to experimentally measure the speed-up of the transformation system while scaling to larger models and clusters. ATL-MR is open source and available online [5].
2. *An efficient data partitioning approach for distributed model transformations* — We propose a data distribution algorithm for declarative model transformation based on static analysis of relational transformation rules. We adapt existing formalization of uniform graph partitioning to the case of distributed MTs by means of binary linear programming. The process starts by extracting footprints from transformation rules. Driven by the extracted footprints, we then propose a fast data distribution algorithm. This algorithm is inspired by recent results on balanced partitioning of streaming graphs. To validate our approach, we apply it to our distributed MT engine. We implement our heuristic as a custom split algorithm for ATL on MapReduce and we evaluate its impact on remote access to the underlying backend.
3. *A persistence framework for storing and accessing VLMs in distributed environments* — We extend an existing multi-persistence backend for manipulating and storing EMF-based models in NoSQL databases, NeoEMF. This solution is implemented in three different backends, each, well-suitable for a specific modeling scenario. NeoEMF comes with a set of caching strategies to enhance its performance. The proposed implementation is highly extensible and adaptable and MDE developers can extend it with additional caching strategies and even backends on a need basis. NeoEMF is open source and available online [6].

1.5 Outline of thesis

The Map of thesis with chapters and relations among them is shown in Figure 1.2.

Chapter 2 gives an overview of the main concepts used in this thesis. It starts by introducing MDE as a software development paradigm, and briefly describes some of its

5. https://github.com/atlanmod/ATL_MR/
6. http://www.neoemf.com/

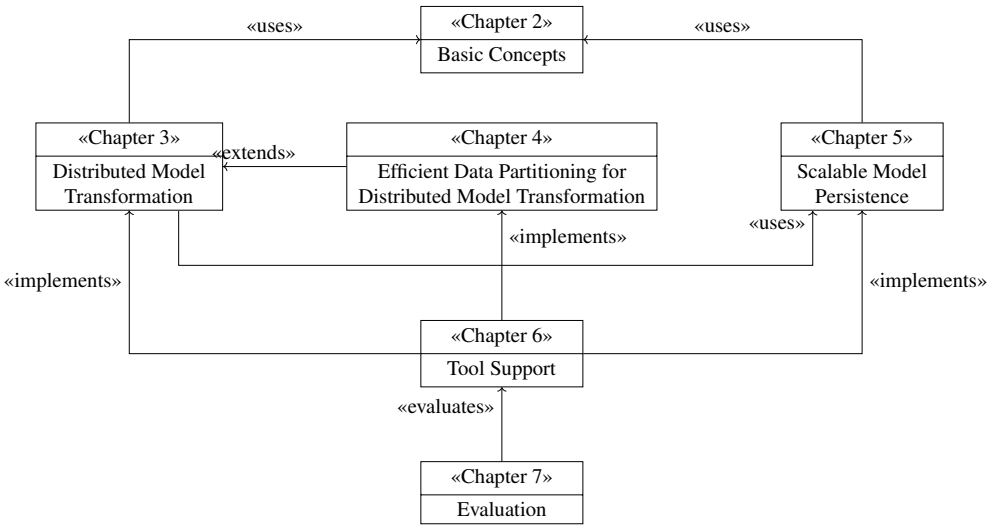


Figure 1.2 – Thesis outline

core concepts. Later, it introduces parallel programming models, more specifically implicit and explicit parallelism. It also presents NoSQL databases as an alternative to store data. It gives an overview of different categories, then details the two categories we use in our thesis.

Chapter 3 presents our conceptual framework and identifies the essential bricks to accomplish distributed model transformations. Afterwards, it introduces our contribution for distributing model transformations on MapReduce. Finally, it examines the execution semantics of ATL against the Reflective MOF specification in order to extract the minimal set of Atomicity, Consistency, Isolation, Durability (ACID) properties guaranteeing a consistent output. This chapter is an enhancement of our work in [17].

Chapter 4 extends our approach with efficient data model partitioning. It first motivates the need for efficient partitioning algorithms. Later, it presents the formalization of our problem using Integer Linear Programming. Then it introduces our two algorithms for footprints extraction and data partitioning algorithm. This chapter is based on our work in [19].

Chapter 5 presents our solutions to store VLMs in EMF. It starts by showing a global overview of our solution, later it describes different data mappings to the underlying backends. Finally, it details the techniques and optimization used in NEOEMF. This chapter is an enhancement of our work in [15, 58].

Chapter 6 validates our contributions by introducing the implementation of our solution as a framework for efficient distribution of model transformation in MapReduce. It also describes the integration process with NEOEMF to enable distributing model transformations in MapReduce. This chapter is an enhancement of our work in [16, 58].

Chapter 7 shows the scalability of our solutions by applying them to various well-known case studies. The results depicted in this chapter come from our work in [15, 17, 19].

Chapter 8 concludes this manuscript, draws the main contributions, and finally describes some future work.

1.6 Scientific Production

— International Conferences

1. **Benelallam, A.**, Gómez, A., Sunye, G., Tisi, M., & Launay, D.(2014,July). a scalable persistence layer for EMF models. In Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA) (pp. 230-241). Springer International Publishing.
2. **Benelallam, A.**, Gómez, A., Tisi, M., & Cabot, J. (2015, October). Distributed model-to-model transformation with ATL on MapReduce. In Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE) (pp. 37-48). ACM Publishing.
3. **Benelallam, A.**, Tisi, M., Cuadrado, J. S., Delara, J., & Cabot, J. (2016, October). Efficient model partitioning of distributed model transformations. In Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE) (pp. 226-238). ACM Publishing.

— International Workshops

1. **Benelallam, A.**, Tisi, M., Ráth, I., Izso, B., & Kolovos, D. S. (2014, July). Towards an Open Set of Real-World Benchmarks for Model Queries and Transformations. In Proceedings of the 2nd International Workshop on BigMDE (pp 14-22), @STAF 2014.
2. **Benelallam, A.**, Gómez, A., & Tisi, M. (2015, October). ATL-MR: model transformation on MapReduce. In Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems (pp. 45-49), @SPLASH 2015. ACM Publishing.
3. Gómez, A., **Benelallam, A.**, & Tisi, M. (2015, July).Decentralized Model Persistence for Distributed Computing. In Proceedings of the 3nd International Workshop on BigMDE (pp. 42-51), @STAF 2015.
4. Daniel, G., Sunyé, G., **Benelallam, A.**, & Tisi, M. (2014, July). Improving memory efficiency for processing large-scale models. In Proceedings of the 2nd International Workshop on BigMDE (pp 31-39), @STAF 2014.
5. Albonico, M., **Benelallam, A.**, Mottu, J.M., & Sunyé, G. (October 2016). A DSL-based Approach for Elasticity Testing of Cloud Systems. In Proceedings of the 3rd Workshop on Domain-Specific Modeling (pp. to appear), DSM@SPLASH 2015. ACM Publishing.
6. Daniel, G., Sunyé, **Benelallam, A.**, Tisi, M., Bernageau, Y., Gómez, A., & Cabot, J. (September 2016). NeoEMF: a Multi-database Model Persistence Framework for Very Large Models. Demo track @MODELS, International Conference on Model Driven Engineering Languages and Systems (pp. to appear).

2

Basic concepts & background

This chapter gives an overview of the main concepts used in this thesis. In a first part, we introduce the notion of Model Driven Engineering, and we briefly describe one application domain among others. Furthermore, we introduce the concepts of models, metamodels, and model transformation languages. We focus on relational model transformation, which we exemplify by the means of the ATL language. Second, we introduce the concepts of Parallel Programming Models. Specifically, we present the two main existing approaches, (i) implicit parallelism, and (ii) explicit parallelism. Finally,we present NoSQL databases as an alternative to store data in other forms than relational databases. We give an overview of different categories of NoSQL databases, then we depict in more details the two categories that will be exploited in course of this thesis, namely, column stores and graph stores.

2.1 Basic concepts in MDE

2.1.1 Model-Driven Engineering

MDE is a software development paradigm favouring the use of models to assist and drive the development and maintenance of software. MDE came to the surface as a generalization of the MDA (Model Driven Architecture) paradigm, an OMG standard. After that, Kent et al. [78] extended it with the notions of software development process and modeling space for organizing models. MDE inherits most of MDA concepts, e.g. metamodels, models, and transformations.

MDE has been successfully applied to several domains. A typical example is the model-based modernization of legacy systems (exemplified in Figure 2.1). A software modernization process follows a systematic approach that starts by building high-level abstraction models from source code (`injection`). Thanks to these models, engineers are able to `understand`, `evaluate`, and `refactor` legacy enterprise architectures. Us-

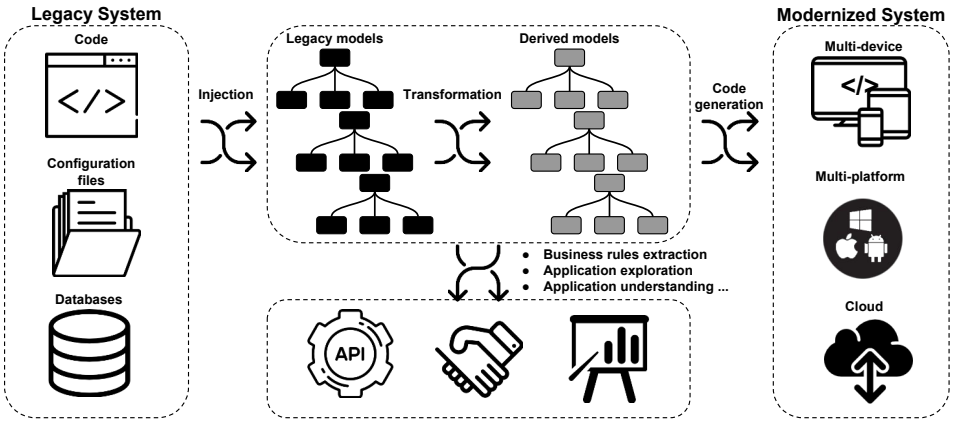


Figure 2.1 – Modernization process of legacy applications

ing these intermediate models, engineers are able to partially generate modernized code for a specific platform (`code generation`). Shifting models from one domain to another, generating code from them, or exploring and understanding them is achieved by means of model-graph queries and transformations (MQTs). Specific languages and transformation engines have been designed to help users define and execute these operations.

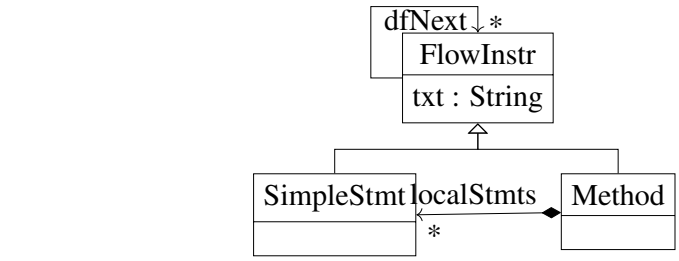
Hereafter, we define the main concepts of MDE, namely, models, metamodels, and model transformations. Later, we introduce the ATL Transformation Language, a relational (rule-based) model-to-model transformation language by means of a running example, the `ControlFlow2Dataflow` transformation.

Models, Metamodels, and Model Transformations

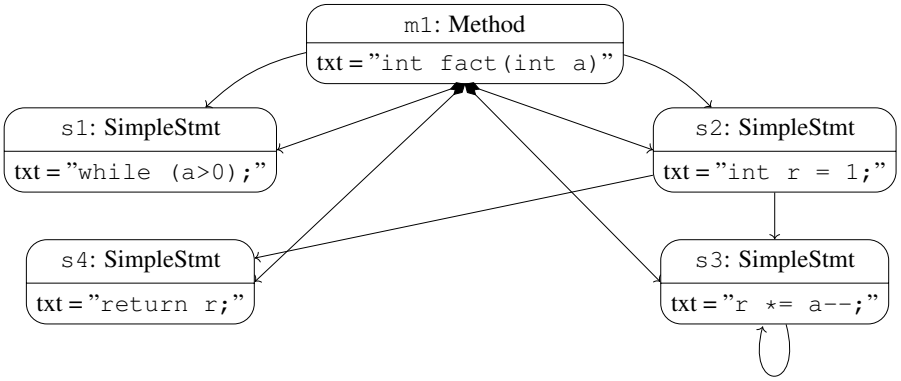
Models. They are an abstracted representation of a system under design. Usually, models are built for a specific goal and intent using a precise denotation defined by a formal or semi-formal language, called modeling language (e.g. UML). Models are used in various disciplines such as civil engineering, mechanical engineering, and computer science. One of the main characteristics of a model is its level of abstraction. Serving a particular goal, a model can represent some characteristics of a system but not others. Nonetheless, this representation should preserve some intended characteristics of the system, in order to allow acquiring knowledge about these characteristics through the model [85].

Metamodels. Metamodeling is the process of studying or providing a well-defined modeling technique (facility). A metamodel define concepts, relationships, and semantics for exchange of user models between different modeling tools. It is a model representing the structural representation of a set of models. It consists of a set of concepts, and rules regulating how models can be denoted in a particular language. A model `conforms to` its metamodel, when it satisfies these rules.

Models are organized in abstraction in spans of multiple levels related by the conformance relationship. This organization is also referred to as the metamodeling stack. In MDE, this stack is limited to three levels, with a self-reflective metamodel level on



(a) DataFlow metamodel excerpt



(b) Sample DataFlow model

Figure 2.2 – DataFlow metamodels excerpt and a sample model

top, called `meta-metamodel` level. The MetaObject Facility [1] (MOF) is the standard meta-modeling architecture proposed by the OMG. In its second version, MOF-2, is built around other OMG standards, namely Object Constraint Language (OCL) [105] for specifying constraints on MOF-based models, and XML Metadata Interchange (XMI) [107] for storing and interchanging MOF-based models in XML.

Figure 2.2 shows an excerpt of a `DataFlow` metamodel together with a sample model that `conformsTo` to it. In a dataflow diagram (Fig. 2.2a), a *FlowInstruction*(FlowInstr) has a field *txt* containing the textual code of the instruction. A method may contain a set of simple statements *localStmts*. A *FlowInstruction* points to the potential set of instructions to whom data may be propagated (*dfNext*). *Method* signatures and *SimpleStatements* (SimpleStmt) are kinds of *FlowInstruction*. Figure 2.2b depicts a dataflow model of a factorial method (Fig. 2.6a). The method `fact` contains a set of local statements (connected to it by diamond arrows) and has the method signature as `txt` attribute value. Local statement on their turn, contain the text of the statement they represent. Finally, statements are connected by simple arrows to show the flow of data between them.

Model Transformations. Model transformation is defined as the operation of automatically generating one or more output models from one or more input models according to a transformation specification. This specification can be implemented using any GPL. However, in the MDE community, it is more common to use Domain Specific Languages (DSLs) to specify these operations. This family of DSLs is also referred to as Model Transformation Languages. The OMG issued a Query/View/Transformation [106] (QVT) request for proposal, which lead to a plethora of model transformation languages [80, 75, 94, 38].

1. `http://www.omg.org/mof/`

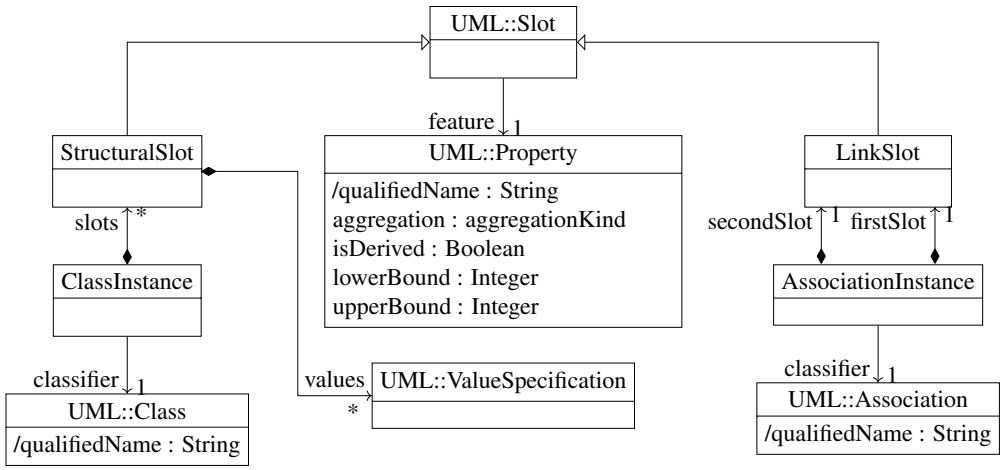


Figure 2.3 – Simplified excerpt of "Instances Model" metamodel in MOF

Model transformation languages can either be `declarative`, `imperative`, or `hybrid`. 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. QVT is defined as an ensemble of three languages `Relations`, `Core`, and `Operational`. They differ in their programming style and their abstraction level. Relations and Core languages are declarative but lay in two different levels of abstraction. Whilst, Operational mappings is an imperative language that extends Relations and Core languages.

Another paradigm of model transformation languages is graph-based [38, 122, 104, 12]. Based on a graph rewriting style, these languages apply transformation rules recursively until no more matches can be found. In a pure graph-rewriting system, the order on which rule applications are executed is not deterministic, and relies heavily upon the causal dependency of rules. Nonetheless, some of these languages are provided with advanced control mechanisms to handle complex transformations. In this thesis we focus on relational model transformation languages.

In what follows, we give a brief description of the MOF metamodel, then we introduce ATL, a QVT/Relations-like language.

2.1.2 The Meta Object Facility (MOF)

We start by giving a brief description of the `instances model` metamodel in MOF, which depicts the relationship between data models and their conforming metamodel. Later, we show the MOF Reflection API. This API enables the discovery and the manipulation of objects and data conform to the MOF specification.

The MOF instances model

A simplified metamodel of "model instances" in MOF is depicted in Figure 2.3. MOF extends the (`UML::`)`InstanceSpecification` class to specialize two types

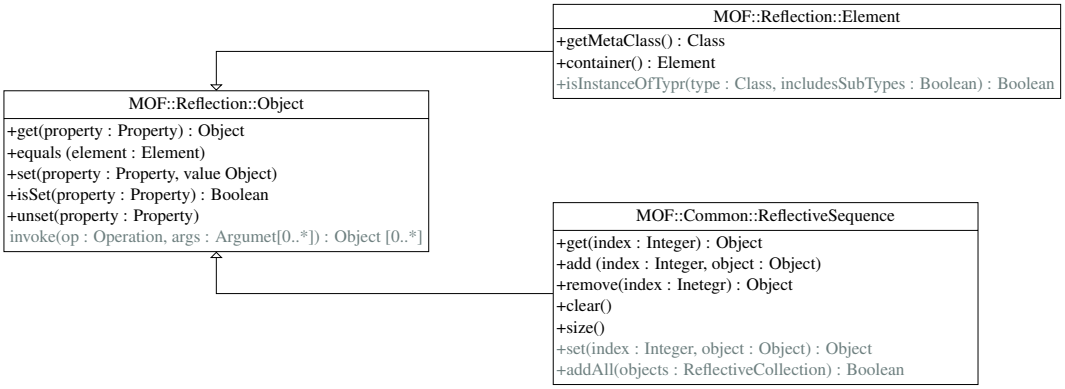


Figure 2.4 – Reflection Element in MOF (extracted from [110])

of instances, associations (`AssociationInstance`) and classes (`ClassInstance`). An `InstanceSpecification` represents data modeled in a system according to its specification. Each instance refers to its corresponding `classifier`, which is either a Class or an Association. An (`UML::`)`Slot` specifies features' values for class (`StructuralSlot`) or association (`LinkSlot`) instances.

An (`UML::`)`Property` is either owned by an Association or a Class. When a property relates to an Association, then it represents a `memberEnd`. Otherwise, it represents a class attribute. A property is derived (`isDerived = true`) when its value can be computed based on existing values. Derived attributes may not be persisted.

Properties of associations can be of two aggregation kinds `composite` or `none`. The composite kind is used to represent a context in which a model element can group other elements of the model (`isComposite`[2]). A model element can only have one container, and does not support cyclic containments. Hence, MOF models have a tree-like structure. Elements with no containers are called `root` elements.

Values are specified using `ValueSpecification`. A value specification can either be of kind `DataValue` (a.k.a. `LiteralSpecification`) in case of attributes, or `InstanceValue` in case of association ends. Six different kinds of literals are supported in MOF, `Strings, Integers, UnlimitedNatural, Real, Boolean`, and finally `Null`. The latter is used to explicitly model the lack of a value.

MOF Reflection API

Figure 2.4 shows the MOF Reflection API. This API enables the discovery and the manipulation of objects and data conform to the MOF specification. `Object` was introduced in order to have an element that represents both model elements and data. An object can get the value of any of its properties (*get(property: Property)*). If the property is multi-valued, it returns a reflective collection containing the values of the property. Otherwise it returns the value of the property.

An object can also modify its properties through the method (*set(property: Property, object : Object)*). The concrete kind of the method's parameter *object* depends on the property's multiplicity. If the property is multi-valued, the object has to be of type `ReflectiveCollection`. If not, it is directly replaced by the property value within

2. aggregation = composite

parameter. The method *unset* sets the value of the single-valued property to its default value for data-types and null for objects. If the property is multi-valued, then the collection is cleared. Finally, the method *isSet* returns if the value of the property is equal to its default value in case the property is single-valued, or if it is empty otherwise.

The class `Element` is the superclass of all classes defined in MOF, and they inherit all of its reflective capabilities. Each element can access its metaclass (*getMetaClass()*) in order to get its reflective description. It can also access its container (*container()*), if any. The class `ReflectiveSequence` is used for manipulating ordered multi-valued properties. It is basically a subtype of `ReflectivCollection` (not shown in Figure 2.4, however all its operations are depicted in the class `ReflectiveSequence`). It has the capacity to *add* or *remove* an element at a given position, clear a collection, or return its size.

Properties can define bidirectional associations. In this case, performing a modification on one side, triggers automatically the appropriate modification at the opposite side. Specifically, modifying the value of a bidirectional association entails the following sequence of operations. (i) The object is first removed from the opposite end of the property. (ii) If the new value's opposite property is of multiplicity upper bound equals to 1, its old value is removed. (iii) Then, this object is added to the new value's opposite property. (iv) Finally, the new value is added to this property. A containment (even if not bidirectional) triggers similar side-effects.

2.1.3 The ATL Transformation Language

Throughout this manuscript we refer to a couple of case studies to elucidate our contributions. However, in order to introduce the ATL transformation language, we choose as a case study, the analysis of data-flows in Java programs. The case study is well-known in the MDE community, being proposed by the Transformation Tool Contest (TTC) 2013 [61] as a benchmark for MT engines. We focus on one phase of the scenario, the transformation of the control-flow diagram (Figure 2.5) of a Java program into a dataflow diagram (Figure 2.2). Such task would be typically found in real-world model-driven applications on program analysis and modernization of legacy code.

In a control-flow diagram, a *FlowInstruction* (FlowInstr) has a field *txt* containing the textual code of the instruction, a set of variables it defines or writes (*def*), and a set of variables it reads (*use*). A method may contain a set of simple statements *localStmts*. A *FlowInstruction* points to the potential set of instructions that may be executed after it (*cfNext*). *Method* signatures and *SimpleStatements* (SimpleStmt) are kinds of *FlowInstruction*. A *Parameter* is a kind of *Variable* that is defined in method signatures.

For every flow instruction n, a *cfNext* link has to be created from all nearest control-flow predecessors m that define a variable which is used by n. Formally [61]:

$$\begin{aligned} m \rightarrow_{dfNext} n \iff\ & def(m) \cap use(n) \neq \emptyset\ \wedge \exists\, Path\ m = \\ & n_0 \rightarrow_{cfNext} \ldots \rightarrow_{cfNext} n_k = n : \\ & (def(m) \cap use(n)) \backslash \Big(\bigcup_{0<i<k} def(n_i) \Big) \neq \emptyset \end{aligned} \tag{2.1}$$

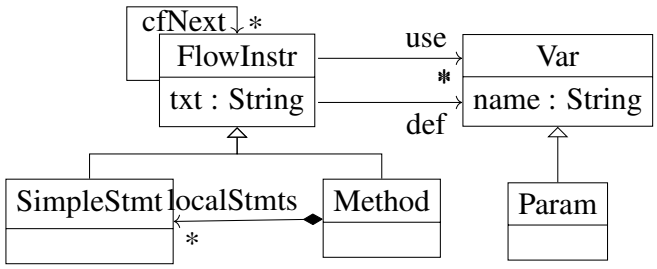


Figure 2.5 – Control-Flow metamodel excerpt

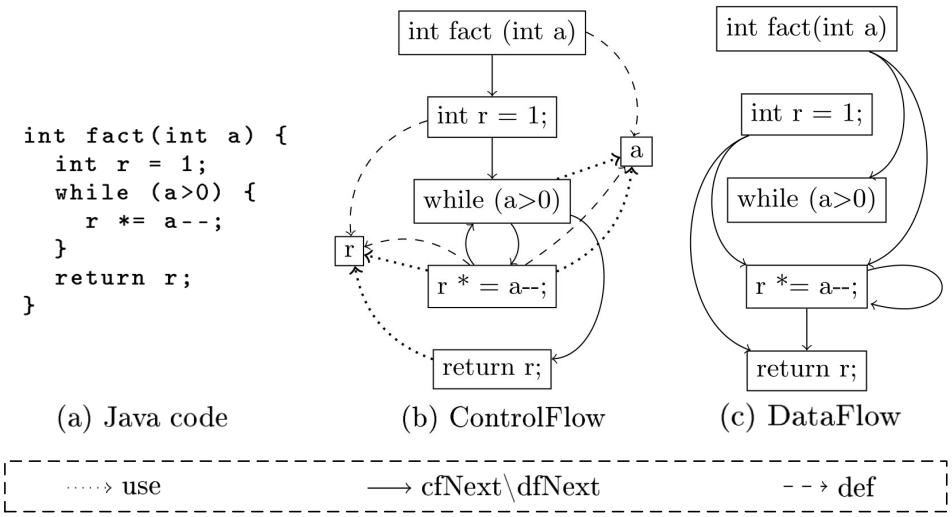


Figure 2.6 – ControlFlow2DataFlow transformation example

Figure 2.6 shows an example of models for each metamodel, derived from a small program calculating a number factorial. For readability reasons, and in order not to congest our graphs, containment references are omitted. As it can be seen in the figure, the transformation changes the topology of the model graph, the number of nodes and their content, and therefore can be regarded as a representative example of general transformations. In this section we refer to an ATL implementation of the transformation named `ControlFlow2DataFlow` that is available online.

Model transformations in ATL are unidirectional. They are applied to read-only source models and produce write-only target models. ATL developers are encouraged to use declarative rules to visualize and implement transformations. Declarative rules abstract the relationship between source and target elements while hiding the semantics dealing with rule triggering, ordering, traceability management and so on. However, rules can be augmented with imperative sections to simplify the expression of complex algorithms. In this thesis, we focus on declarative-only ATL.

Languages like ATL are structured in a set of transformation rules encapsulated in a transformation unit. These transformation units are called `modules` (Listing 2.1, line 1). The query language used in ATL is the OMG's OCL [105]. A significant subset of OCL data types and operations is supported in ATL. Listing 2.1 shows a subset of the rules in the `ControlFlow2DataFlow` transformation and Listing 2.2 an excerpt of its OCL queries (helpers).

Listing 2.1 – ControlFlow2DataFlow - ATL transformation rules (excerpt)

```
1  module ControlFlow2DataFlow;
2  create OUT : DataFlow from IN : ControlFlow;
3  rule Method  {
4    from
5      s : ControlFlow!Method
6    to
7      t : DataFlow!Method (
8        txt <- s.txt,
9        localStmts <- s.localStmts,
10       dfNext <- s.computeNextDataFlows()
11     )
12 }
13
14 rule SimpleStmt {
15   from
16   s : ControlFlow!SimpleStmt (not(s.def-> isEmpty() and s.use->isEmpty()))
17   to
18     t : DataFlow!SimpleStmt (
19       txt <- s.txt,
20       dfNext <- s.computeNextDataFlows()
21     )
22 }
```

ATL *matched rules* are composed of a source pattern and a target pattern. Both of source and target patterns might contain one or many pattern elements. Input patterns are fired automatically when an instance of the source pattern (a match) is identified, and produce an instance of the corresponding target pattern in the output model. Implicitly, transient tracing information is built to associate input elements to their correspondences in the target model.

Source patterns are defined as OCL `guards` over a set of typed elements, i.e. only combinations of input elements satisfying that guard are matched. In ATL, a source pattern lays within the body of the clause '`from`' (Listing 2.1, line 15). For instance, in the rule *SimpleStmt*, the source pattern (Listing 2.1, line 16) matches an element of type *SimpleStmt* in which at least one variable is defined or used. Output patterns, delimited by the clause '`to`' (Listing 2.1, line 18) describe how to compute the model elements to produce when the rule is fired, starting from the values of the matched elements.

E.g., the *SimpleStmt* rule produces a single element of type *SimpleStmt*. A set of OCL `bindings` specify how to fill each of the features (attributes and references) of the produced elements. The binding at line 19 copies the textual representation of the instruction, the binding at line 20 fills the *dfNext* link with values computed by the `computeNextDataFlows` OCL `helper`. The rule for transforming methods is analogous (Listing 2.1, lines 3-12).

OCL helpers enable the definition of reusable OCL expressions. An OCL helper must be attached to a context, that can be a type or global context. Since target models are not navigable, only source types are allowed. Listing 2.2 shows our implementation of the `computeNextDataFlows` helper derived by the direct translation in OCL of the data-flow definition we gave in Equation 2.1. It has as context `FlowInstr` and returns a sequence of same type (Listing 2.2, line 1).

ATL matched rules are executed in two phases, a *match phase* (Algorithm 1 lines 1–6) and an *apply phase* (Algorithm 1 lines 7–14). In the first phase, the rules are applied over source models' elements satisfying their guards. Each single match, corresponds to the creation of an explicit traceability link. This link connects three items: the rule that

Listing 2.2 – ControlFlow2DataFlow - OCL helpers (excerpt)

```
1  helper Context ControlFlow!FlowInstr def :computeNextDataFLows() : Sequence (
       ↪ControlFlow!FlowInstr) =
2      self.def ->collect(d | self.users(d)
3           ->reject(fi | if fi = self then   not fi.isInALoop else false endif )
4           ->select(fi | thisModule.isDefinedBy(fi,Sequence{fi},self, Sequence{}, self.
                ↪definers(d)->excluding( self))))
5           ->flatten();(a)
6
7  helper def : isDefinedBy(start : ControlFlow!FlowInstr, input : Sequence(ControlFlow!
       ↪FlowInstr), end : ControlFlow!FlowInstr, visited :Sequence(ControlFlow!
       ↪FlowInstr), forbidden : Sequence(ControlFlow!FlowInstr)) : Boolean =
8          if input->exists(i | i = end) then true
9          else let newInput : Sequence(ControlFlow!FlowInstr) = input ->collect(i |i.
               ↪cfPrev) ->flatten() ->reject(i | visited ->exists(v | v = i) or
               ↪forbidden ->exists(f| f = i)) in
10           if newInput ->isEmpty() then false
11           else  thisModule.isDefinedBy(start, newInput, end, visited->union(newInput)
                   ↪->asSet() ->asSequence(), forbidden)
12           endif
13         endif;
```

triggered the application, the match, and the newly created output elements (according to the target pattern). At this stage, only output pattern elements type is considered, bindings evaluation is left to the next phase.

The apply phase deals with the initialization of output elements' features. Every feature is associated to a binding in an output pattern element of a given rule application. Indeed, a rule application corresponds to a trace link. Features initialization is performed in two steps, first the corresponding binding expression is computed. Resulting in a collection of elements, it is then passed to a resolution algorithm (called `resolve`) for final update into the output model.

The `resolve` algorithm behaves differently according to the type of each element. If the type is primitive (in the case of attributes) or target, then it is directly assigned to the feature. Otherwise, if it is a source element type, it is first resolved to its respective target element – using the tracing information – before being assigned to the feature. Thanks to this algorithm we are able to initialize the target features without needing to navigate the target models.

2.2 Parallel programming models

Parallel computing refers to the use of a set of commodity resources (processors or machine), jointly, to solve a single computationally expensive problem. Interest on parallel computing kept on growing since the late 50's. This era is acknowledged by the advancements on supercomputers manufacturing for the following decades. In the early 90's, a new topology broke out to replace supercomputers, and dominate in many applications development: computer clusters. This new trend took up interest as an instrument for low-cost and effective parallel processing environments.

Implementing parallel application has always been a challenging procedure. However, parallel computing evolved thanks to the parallel progress in software development [32]. Traditionally, a problem is decomposed into smaller sub-problems (tasks) that are run in parallel, in one or more steps. These sub-problems are then subject to join

Algorithm 1: ATL algorithm

```
input : Rules transfomationRules, Map elementsByType
// STARTS THE MATCH PHASE
1  foreach rule ∈ transfortmationRules do
2      foreach type ∈ getApplicableTypes(rule) do
3          foreach element ∈ getAllInstancesOfType(elementsByType, type)
           do
4              if isMatched(element, rule) then
5                  link ← createLink(element, rule);
6                  addLink(linkSet, link);
// STARTS THE APPLY PHASE
7  foreach link ∈ linkSet do
8      foreach binding ∈ getBindings(link) do
9          if isAttribute(binding) then
10             applyAttribute(binding);
11         else
12             foreach element ∈ computeBindingExp(binding, link) do
13                 resolvedElement ← resolveTarget(element);
14                 applyReference(resolvedElement, binding);
```

up (`synchronization`) at a certain point to maintain consistency and exchange information (`communication` and `coordination`). This programming style requires familiarity and knowledge on parallel software development, which average programmers do not own.

In order to make developing parallel software approachable for regular software developers, a plethora of models, languages, and frameworks for parallel development emerged. Models for parallel software development appear at different levels of abstraction. The highest one is the `programming model`. It expresses, from a programmer's view, how an algorithm can be specified. Each parallel programming model supports a set of operations/patterns to help to implement parallel software, namely parallel loops and independent groups of tasks (pools).

Programming models are distinguished by different criteria. But most importantly, they are identified by their ability to either implicitly or explicitly specify the parallelism. In the next section, we introduce the explicit and implicit representation styles, and we identify their pros and cons.

2.2.1 Explicit vs. implicit parallelism

There exist two main approaches for parallel programming, implicit and explicit representations. Both approaches differ in the parallelization's automation level, as well as the efficiency guaranteed. While implicit approaches rely on parallel languages and frameworks to ease the development by hiding internal complexities related to parallelism, the explicit approaches assume that the programmer is the best judge to decompose tasks

and co-ordinate between them. Although they are hard to use, explicit approaches usually offer more flexibility, which sometimes leads to better performance and efficiency [116].

Explicit parallelism requires the programmer to be responsible for decomposing the problem and designing the communication structure between tasks. This comes in different forms, *explicit communication and synchronization*, *explicit assignment to processors*, and *explicit distribution*.

In the explicit *communication and synchronization* class, the programmer has to imperatively define all the required communication and synchronization between tasks to control the parallel execution of the application. Message-passing models, e.g. MPI, and thread-based models (e.g. Pthread [118]) are two famous example belonging to this class.

The *explicit assignment to processors* class identifies the family of parallel programming models that require to specify task partitioning and assignment to processors but hide the communication details. The best known examples are co-ordination languages, in particular, Linda [142]. It substitutes inter-processor communication by using tuple spaces, a pool of data that can be accessed concurrently.

In contrast to the previous class, the *explicit distribution* class implicitly maps tasks to processors. However, the decomposition into tasks as well as the parallelism should be explicitly specified. A very known example is the Bulk Synchronous Parallel (BSP [134]) programming model.

Implicit parallelism allows the programmer to care only about application development as in sequential execution, without specifying any parallelism primitives. Implicit parallelism is commonly achieved by restricting the semantics and data structures (e.g. graphs, maps, vectors, etc.) of the operations of the language. They can be also provided with built-in communication and coordination constructs at either the language or the framework levels. Such constructs enable automatic data and task management.

In literature, two main classes of implicit parallelism exist: *parallelizing compilers*, parallel languages, in particular *functional programming languages*. However, with the emergence of Cloud computing, various programming models provided with implicit parallelism have come out, namely, *MapReduce* [44] and *Pregel* [91], just to name a few. These models have been implemented in various GPLs (e.g. Java). In this section we focus on the MapReduce model.

A parallelizing compiler aims at automatically generating an efficient parallel program out of a sequential one (automatic parallelization). It starts by performing a dependency analysis of the operations to be executed. Afterwards, it assigns computations to processors in a way that guarantees a good load-balancing and optimal amount of inter-processors communication. However, sometimes, it is impractical to achieve a good dependency analysis. In particular, functions call and loops with unknown bound are difficult to predict at compile time, which sometimes results in an unacceptable runtime behaviour.

Functional languages are an attractive solution to implement parallel software. They are provided with efficient abstraction techniques that can be used for both computation and coordination [89], e.g. *function composition* and *higher-order functions*. Moreover, the absence of side-effects simplifies the process of identifying possible parallelism, and thus

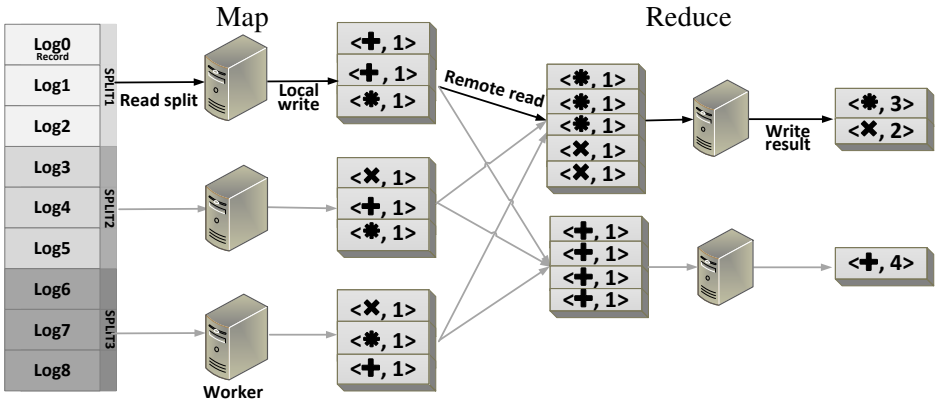


Figure 2.7 – MapReduce programming model overview

eliminate unnecessary dependencies. Indeed, functional languages enabled and inspired the development of various distribution frameworks.

One of the most popular and new emerging distributed programming models is Google's MapReduce. It enables general-purpose programming through user-defined functions, `map` and `reduce`. The semantics of these high-order functions enable implicit data parallelism, which eases parallel software development of data intensive application. However, when dealing with complex programs it becomes necessary to add new primitives. In this sense many projects such as Spark [141] have extended the MapReduce model with rich data-flow APIs and support for non-trivial data and control flow.

2.2.2 MapReduce

MapReduce is a programming model and software framework developed at Google in 2004 [44]. It allows easy and transparent distributed processing of big data sets while concealing the complex distribution details a developer might cross. MapReduce is inspired by the map and reduce primitives of functional languages. Both *Map* and *Reduce* invocations are distributed across cluster nodes, thanks to the *Master* that orchestrates jobs assignment.

Input data is partitioned into a set of chunks called *Splits* as illustrated in Figure 2.7. The partitioning might be controlled by the user throughout a set of parameters. If not, *splits* are automatically and evenly partitioned. Every *split* comprises a set of logical *Records*, each containing a pair of ⟨key, value⟩.

Given the number of `Splits` and idle nodes, the Master node decides the number of workers (slave machines) for the assignment of `Map` jobs. Each `Map worker` reads one or many `Splits`, iterates over the `Records`, processes the ⟨key, value⟩ pairs and stores locally the intermediate ⟨key, value⟩ pairs. In the meanwhile, the `Master` receives periodically the location of these pairs. When `Map workers` finish, the `Master` forwards these locations to the `Reduce workers` that sort them so that all occurrences of the same key are grouped together. The `mapper` then passes the key and list of values to the user-defined reduce function. Following the reduce tasks achievement, an output result is generated per reduce task. Output results do not need to be always combined, especially if they will subsequently be processed by other distributed applications.

Let's take a closer look at the MapReduce programming model by means of a simple example, depicted in Figure 2.7. Assume we have set of log entries coming from a git repository. Each entry contains information about actions performed over a particular file (creation $\rightarrow +$, deletion $\rightarrow X$, or modification $\rightarrow *$). We want to know how many times each action was performed, using MapReduce. The master evenly splits the entries among workers. For every record (log entry), the map worker extracts the action type and creates a $\langle$key,value$\rangle$ pair with a key the action itself and value '1'. In the reduce phase, pairs with the same key are grouped together. In our example, the modification and deletion go to the first reducer, while the creation goes to the second one. For each group, the reducer combines the pairs, and creates a $\langle$key,value$\rangle$ pair, but this time with the total of the values with the same key. This value refers to how many times the action occurred in the logs.

A useful optimization feature shipped with MapReduce is the `Combiner`. It is an optional class taking place right after the map phase, and before the shuffle phase. It operates on pairs originating from the mapper running on the same node. For each set of pairs sharing the same key, values are aggregated in order to combine them into a single pair according to the `Combiner` class definition. As you can notice, the main benefit using combiners is to reduce the volume of data sent between the `Map` and `Reduce` phases, and therefore the time taken to shuffle different pairs across the cluster.

It is thanks to the Distributed File System (DFS) that MapReduce framework owes its ability to scale to hundreds of machines in a cluster. The master node in MapReduce tends to assign workloads to servers where data to be processed is stored to maximize data locality. The interest of MapReduce is due to its fault-tolerant processing. The `Master` keeps track of the evolution of every worker execution. If after a certain amount of time a worker does not react, it is considered as idle and the job is re-assigned to another worker. DFS data is divided into blocks, and copies of these blocks are stored in different nodes in the cluster to achieve a good availability in case nodes fail.

Hadoop [7] and Hadoop Distributed File System (HDFS) [8] are the most well-known and widely used open source implementations of MapReduce and Google-DFS respectively.

2.3 NoSQL databases

Relational schema (SQL) was the most adopted model for storing data in the past decades. With the growing volume of data, and its variating structure, this model is coming up against scalability issues in data representation and query. In particular, the relational model fails to represent some forms of data (e.g. temporal, unstructured, etc.). Also, it lacks an explicit representation of relationships. Moreover, query execution times increase as the size of tables and the number of joins grow. In response to these challenges, many non-relational database models have been proposed, under the umbrella of NoSQL (Not only SQL) [121] databases. Despite they generally do not respect the classical (ACID) properties, these databases are able to manage large-scale data on highly distributed environments.

Although NoSQL databases vary widely in data models, data types, and use cases, they can be grouped into four categories, *Key-Value*, *Document*, *Graph*, and *Column*. Key-value stores are used for storing and managing associative arrays. Document stores group data in

documents that can be thought of as a family of key-value pairs. Graph databases consist of networks of records (vertices) connected by explicit relationships (edges). They are suitable for storing and querying densely inter-connected data. Finally, column databases group data by column in contrast to relational databases that store rows instead. This representation is more suitable for querying large sets of values of a specific attribute. As in this thesis we use only column stores and graph stores, in what follows, we introduce their basic concepts and data models.

2.3.1 Column store

Also known as wide-column store, it is one of the four general types of NoSQL database. It is quite similar to Key-Value store but it mixes some characteristics of relational databases. Column stores are designed for storing data as of columns of data rather than rows of data (as in Key-Value stores). Bigtable [34] is one of the early implementation that has proven a good scalability. Built by Google, Bigtable is used to store data in many MapReduce-based projects at Google. Apache HBase [124] is the Hadoop [7] database, provided as a distributed, scalable, versioned, and non-relational big data store. It is considered the reference open-source implementation of Google's Bigtable.

Bigtable data model. In BigTable, data is stored in tables, which are sparse, distributed, persistent multi-dimensional sorted maps. A map is indexed by a row key, a column key, and a timestamp. Each value in the map is an uninterpreted array of bytes.

BigTable is built on top of the following concepts [79]:

Table — Tables have a name, and are the top-level organization unit for data in BigTable.

Row — Within a table, data is stored in `rows`. Rows are uniquely identified by their `row key`.

Column Family — Data within a row is grouped by `column family`. Column families are defined at table creation and are not easily modified. Every row in a table has the same column families, although a row does not need to store data in all its families.

Column Qualifier — Data within a column family is addressed via its `column qualifier`. Column qualifiers do not need to be specified in advance and do not need to be consistent between rows.

Cell — A combination of `row key`, `column family`, and `column qualifier` uniquely identifies a `cell`. The data stored in a cell is referred to as that cell's value. Values do not have a data type and are always treated as a `byte[]`.

Timestamp — Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp of when the cell was written. If the timestamp is not specified for a read, the latest version is returned. The number of cell value versions retained by BigTable is configured for each column family (the default number of cell versions is three).

Figure 2.8 (extracted from [34]) shows an excerpt of an example table that stores Web pages. The row name is a reversed URL. The `contents` column family holds the page

contents, and the `anchor` column family holds the text of any anchors that reference the page. CNN's home page is referenced by both the `Sports Illustrated` and the `MY-look` home pages, so the row has columns named `anchor:cnnsi.com` and `anchor:my.look.ca`. Each anchor cell has one version; the `contents` column has three versions, at timestamps t_3, t_5, and t_6.

2.3.2 Graph stores

Among the different data models used in NoSQL database, graph databases look particularly suitable to store models. The graph data model uses graph structures with nodes, edges, and properties to store data and provides index-free adjacency. Although this data model is not new—the research on graph databases was popular back in the 1990s—it became again a topic of interest due to the large amounts of graph data introduced by social networks and the web in general.

Graph database management systems expose the graph data model through CRUD operations. It is usually used for transactional systems (OLTP). Several graph database technology vendors exist in the today's market. They differ in two main properties, `the underlying storage` and `the processing engine`. A graph database that depends on a non-native (e.g. using relational or general-purpose stores) graph storage has relationships that will need to be inferred at runtime.

Graph data model. Graph databases consider not only vertices, but also relationships as first class citizen. This representation fits well with the way we visually conceptualize domains using shapes and relationships between them. Hence, a labelled property graph model is a convenient way to represent data.

A `labelled property graph model` consists in *vertices, edges, properties*, and *labels*. Figure 2.9 shows an excerpt of the graph metamodel. Vertices and edges are identified by their `ids`. They contain valued (`value`) and named (`name`) properties. They also can be tagged with labels. Finally, edges connect a `source`. Vertex to a `target` vertex with a named relationship. Edges can also have properties. This feature is useful when implementing graph algorithms. Fig 2.10 shows a sample network of three Twitter users. Each vertex has an id (e.g. user1..3). Users point to each other by directed relationships showing their role in the network (`FOLLOWS`).

Graph traversals. One of the success keys in graph databases is graph `traversals`. A graph traversal is an efficient operation to query data from graph databases. It refers to the operation of visiting graph nodes, and checking, updating, or returning them. A graph traversal is identified by a start node and a traversal specification. A traversal specification

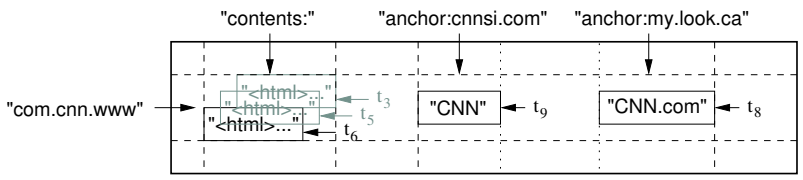


Figure 2.8 – Example of a table in HBase/BigTable (extracted from [34])

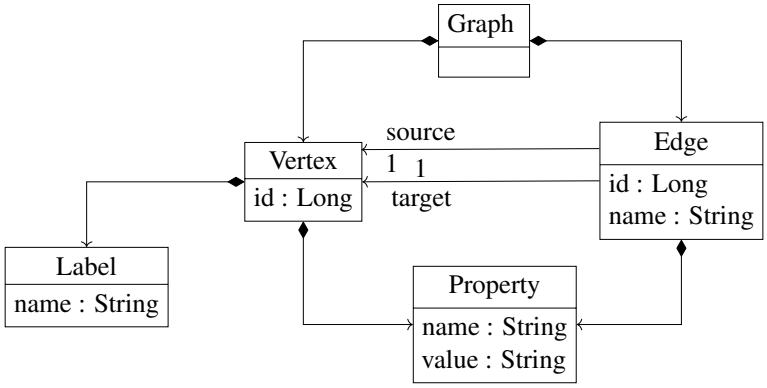


Figure 2.9 – Labeled property graph metamodel

is a directed path that guides the graph visitor through the graph exploration. A graph specification might contain filters on properties or labels to restrict the exploration space.

Most of existing graph databases support vertex-centric and/or relationship-centric indexes. In some cases when it is required to pick up some particular nodes, this feature comes into play. It ensures a fast lookup of graph elements by their `ID`, and thus improves the performance of graph queries. For example, a node index can be useful in identifying the start node for a particular graph traversal.

Conclusion

In this section we introduced the basic concepts and background that will be needed to better grasp our contributions and make the document self-contained. In the first part, we presented some basic concepts of the MDE approach. Especially, we focused on model transformations languages and engines. Then we moved to a specific type of model transformation languages, relational ones. We chose the ATL language to exemplify this family of languages. The second part gave a brief introduction to parallel programming and its different types of models. In particular, we focused on explicit and implicit parallel programming models, and we exhibited the pros and cons of each of them. We concluded this part by giving a closer look at the MapReduce model, one of the well-known implicit distributed programming models. The last part focalized on distributed data storage solutions, particularly, NoSQL databases. We briefly described the different kinds of existing stores, then we targeted two of them, graph stores, and column stores. For each store, we presented the corresponding data model alongside with some of its existing features.

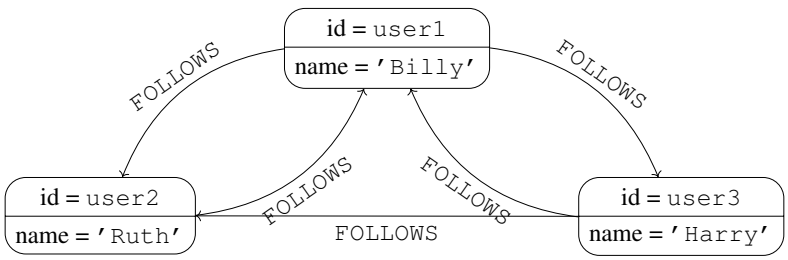


Figure 2.10 – Twitter's data represented by a graph (extracted from [117])

We believe that MDE, especially transformation languages and engines, would benefit from the Cloud to improve the scalability of model-based software development. Next, we outline some potential leads towards improving the scalability of model transformation engines:

Effective and implicit distribution of relational model transformation. The IO/CPU limitations facing complex transformations written in relational languages can be overtaken by distributing computation across a cluster of commodity machines. Following a data-distribution scheme, this process would entail partitioning the input model into chunks, and sending each one to a different machine. The transformation language should however keep its declarative nature, and no additional distribution primitive should be added. Chapter 3 investigates this lead by aligning the semantics of relational model transformation engines to existing distributed programming models. Chapter 4 extends the previous chapter, and investigates model transformation workload balancing.

Adequate model persistence for distributed model transformations. Existing model persistence frameworks are also facing scalability issues when dealing with VLMs. In particular, they are neither memory-friendly nor well-suitable for distributed and concurrent processing. Chapters 5 and 6 investigates the use of NoSQL databases to cope with these limitations.

3

Distributed model transformations

Many approaches aiming at improving the performance of queries and transformations over models have emerged. Whereas some promote incrementality and laziness, others foster parallelization/distribution. Yet, most of these approaches – as we will show in the next section – do not tackle efficiently scalability issues, especially when the model under concern does not fit in a memory of a single machine. One way to respond to this issue is exploiting systems for parallelizing model manipulation (processing) operations over computer clusters. This is made convenient by the recent wide availability of distributed clusters in the Cloud.

MDE developers may already build distributed model transformations by using a GPL and one of the popular distributed programming models such as MapReduce or Pregel. However such development is not trivial. First of all, the transformation logic should be aligned with the execution semantics of the distributed programming model. Moreover, model transformation code written in a GPL is less readable, contains redundancies or verbose expressions [53]. This makes the transformation neither modularizable nor reusable. It may as well introduce a completely new class of errors w.r.t. sequential programming. Finally, it entails complex analysis for performance optimization.

In our contribution, we show that ATL, thanks to its specific level of abstraction, can be provided with semantics for `implicit distributed execution`. As a rule-based language, declarative-ATL allows the declarative definition of correspondences and data flows between elements in the source and target model. By our proposed semantics, these correspondence rules can be efficiently run on a distributed cluster. The distribution is implicit, i.e. the syntax of the MT language is not modified and no primitive for distribution is added. Hence developers are not required to have any acquaintance with distributed programming.

The semantics we propose is aligned with the MapReduce computation model, thus showing that rule-based MT fits in the class of problems that can be efficiently handled by the MapReduce abstraction. We demonstrate the effectiveness of the approach by making

an implementation of our solution publicly available [1] and by using it to experimentally measure the speed-up of the transformation system while scaling to larger models and clusters. We identify specific properties of the ATL language that make the alignment possible and the resulting solution efficient. We also define the minimal set of ACID properties that model access has to guarantee for a consistent and correct model transformation. We achieve this by cross-analyzing the execution semantics of ATL, especially atomic model manipulation operations (as coming from the MOF specification), against the MOF Reflection API. The main principles of our approach can be generalized to other MT languages (e.g. QVT [106] and Epsilon Transformation Language (ETL) [80]), and we plan in future work to investigate this generalization in detail.

This chapter is organized as follows. In Section 3.1 we introduce some related work and identify their limitations. Section 3.2 presents an overview of our framework for distributing model transformation. Section 3.3 details our ATL to MapReduce alignment. Section 3.4 identifies the ACID properties to be preserved at the MOF Reflection API level for sound distributed model transformations in ATL. We close up this chapter by a discussion in Section 3.5.

3.1 Related Work

Parallel and distributed graph transformation is a well-studied problem, and well-defined fundamentals have been proposed in several works. In parallel and distributed graph transformations, rule applications are structured in both temporal and spatial dimensions. Graphs are split into sub-graphs for local transformations, then joined again to form a global graph [49]. Two families of approaches have been proposed, shared memory and message passing. In what follows, we introduce related work attempting to improve graph and model processing performance, namely, parallel and distributed model/graph transformation approaches. We also investigate other alternatives that are implicitly providing MQT engines with advanced execution semantics. Note that related work on persistence solutions for VLMs, as well as task scheduling and graph-data partitioning approaches are to come in the next chapters.

Distribution for graph processing languages

Among distributed graph transformation proposals, a recent one is Mezei et al. [95]. It is composed of a transformation-level parallelization and a rule-level parallelization with four different matching algorithms to address different distribution types. In [72], Izso et al. present a tool called IncQuery-D for incremental query in the Cloud. This approach is based on a distributed model management middleware and a stateful pattern matcher framework using the RETE algorithm. The approach has shown its efficiency, but it addresses only distributed model queries while we focus on declarative transformation rules.

Some approaches mapping a high-level graph transformation language to the Pregel programming model [132, 84] exist. Krause et al. [84] proposed a mapping of a graph transformation language and tool, Henshin, to the "BSP model transformation framework"

1. https://github.com/atlanmod/ATL_MR/

on top of the BSP model. The approach is based on a code generator that translates the graph transformation rules and transformation units into a Pregel program. The matching phase is split into a serie of local steps. A local step checks if a local constraint is satisfied and generates, extends, or merges a set of partial matches. A search plan generation algorithm is responsible for local steps generation.

In a similar approach, Tung et al. [132] implemented their own DSL for the specification of graph transformations. This DSL is also compiled to a Pregel program and executed over a distributed cluster. The DSL inherits most its constructs from UnCAL, a query language and algebra for semistructured data based on structural recursion. The semantics of UnCAL was improved to support the basic Pregel skeletons. In contrast to the previous approach, this one supports successive applications of queries/transformations. Both approaches implement the framework in Giraph, an open-source implementation of the Pregel model.

The only other proposal addressing relational MT distribution is Lintra, by Burgueño et al. [31], based on the Linda coordination language. Lintra uses the master-slave design pattern for their execution, where slaves are in charge of applying the transformation in parallel to sub-models of the input model. The same authors propose a minimal set of primitives to specify distributed model transformations, LintraP [29]. With respect to our approach, Lintra requires to explicitly use distribution primitives, but it can be used in principle to distribute any transformation language by compilation.

Shared-memory parallelization for graph processing languages

Shared-memory parallelization is a closely related problem to distribution. For model transformation, Tisi et al. [128] present a systematic two-steps approach to parallelize ATL transformations. The authors provided a multi-threaded implementation of the ATL engine, where every rule is executed in a separate thread for both steps. The parallel ATL compiler and virtual machine have been adapted to enable a parallel execution and reduce synchronization overhead.

A similar approach for parallel graph transformations in multicore systems [63] introduces a two-phase algorithm (matching and modifier) similar to ours. Bergmann et al. propose an approach to parallelizing graph transformations based on incremental pattern matching [23]. This approach uses a message passing mechanism to notify model changes. The incremental pattern matcher is split into different containers, each one is responsible for a set of patterns. The lack of distributed memory concerns makes these solutions difficult to adapt to the distributed computing scenario. Moreover in these cases the authors investigate task distribution, while we focus on data distribution, especially for handling VLMs.

Alternative execution semantics for scalable transformations

The idea of overloading the execution semantics of model transformation languages in order to improve scalability is not new. In particular several work introduced lazy and incremental computation semantics.

Incremental computing is a software feature aiming at saving the program recomputation time every time a piece of data changes. It identifies the outputs which depend on the changed data, then updates their value. EMF-IncQuery [21, 133] is a declarative model query framework of EMF models using the graph pattern formalism as a query specification language. It aims at bringing the benefits of graph pattern-based queries and incremental pattern matching to the EMF ecosystem. EMF-IncQuery uses the RETE algorithm to achieve incrementality.

Rath et al. [115] investigated the use of derived EMF features in EMF-based models and proposes the use of incremental model queries to define operations over these kinds of features. This approach uses EMF-IncQuery as query engine, which keeps derived features instantly up to date using a change notification and incremental evaluation. Bergmann et al. [20] proposed an approach to integrate incremental pattern matching into existing legacy systems built over RDBs. It translates graph patterns/rules into SQL triggers. Additional tables are added to store cached relationships.

Giese et al. [55] proposed an approach to automatically induce incremental synchronization using Triple Graph Grammar (TGG). The incremental algorithm relies mostly on the correspondence graph (which can also be considered as trace links), and correspondence nodes are thus being used to decrease the matching cost. In [54], Giese et al. presented an algorithm for the synchronization (Change propagation) between different related models. As far as [55], this approach gained from the declarativity of TGG to limit the number of visited elements after a change notification.

Hawk [13] is a model indexing framework that enables efficient global model-element-level queries on collections of models stored in file-based version control systems (VCS). It can operate with different file-based VCSs while providing a query API that can be used by model management tools. Hawk is shipped with features enabling its robustness. Namely, incremental model updates, derived attributes indexed attributes, and scoped queries. Hawk is integrated with the Epsilon Object Language (EOL) query engine. EOL is used to transparently query models in Hawk by using indexes whenever possible. It is also used to define derived attributes.

Jouault et al. [76] introduced an approach for incremental transformations in ATL. This approach relies on two runtime mechanisms, (i) Tracking dependencies of OCL expressions, and (ii) Controlling individual rule execution. Neither of ATL's syntax nor its semantics sustained any change, except for some minimal changes to the compiler.

Another technique for improving the execution of model transformations is the lazy evaluation. It is a strategy which delays the evaluation of an expression until its value is needed. Tisi et al. [130] introduced a lazy execution approach of model transformations in ATL. It enables the generation of target model elements only if accessed (needed). The ATL engine has been extended with an algorithm computing the appropriate binding responsible for generating the accessed feature.

High-level languages for distributed data-parallel computing

Many high-level languages for data parallel computing targeting distributed programming models have been proposed. However, these languages are not suitable for performing

distributed model transformations. They are designed for specific applications such as data warehousing, querying, etc.. Moreover they perform on unstructured and flat data.

Microsoft SCOPE [33], Pig Latin [108], and HiveQL [126] are high level SQL-like scripting languages targeting massive data analysis on top of MapReduce. Pig Latin and SCOPE are hybrid languages combining both forces of an SQL-like declarative style and a procedural programming style using MapReduce primitives. They provide an extensive support for user defined functions. Hive is a data warehousing solution built on top of Hadoop. It comes with an SQL-like language, HiveQL, which supports data definition statements to create tables with specific serialization formats, and partitioning and bucketing columns.

DryadLING is a language designed to target the Dryad [71] engine, a general-purpose distributed execution engine for coarse-grain data-parallel applications. Unlike PigLatin or SCOPE, which introduce new domain specific languages, DryadLINQ is embedded as constructs within existing programming languages. A program written in DryadLINQ is a sequential program composed of expressions specified in LINQ, the **.NET** Language Integrated Query. The expressions perform arbitrary side-effect-free operations on datasets, and can be debugged using standard **.NET** development tools.

Summary

In this section we presented existing approaches aiming at optimizing graph processing, with a particular focus on graph transformation operations. Whilst most approaches in graph/model transformation focus on the parallelization/distribution of graph transformations, only [31, 129] focus on relational model transformation. Moreover, the approach in [129] is still limited to the resources of one machine. The second approach [31] instead, translates the transformation rules together with in(out)put metamodels to a Linda-supported format. Unfortunately, this solution does not integrate well with existing EMF-based solutions. Furthermore, it is not clear how the approach handles concurrent write to the underlying backend. We notice that most existing approaches exhibit either one or more of the following limitations:

1. Except for LinTra, most of the existing approaches apply to graph-transformation languages
2. Some of the existing approaches are still limited by the resources of a single machine
3. Existing transformation engines do not transparently handle and manipulate EMF models, which hampers the integration with existing EMF-based tools
4. Visibly, none of the existing approaches support concurrent write on target models
5. No clear support for complex transformation rules (e.g. rules involving multiple input patterns)

Table 3.1 groups these limitations.

In what follows, we present our solution for distributing model transformations. To elucidate the discussion of our approach, we refer throughout this chapter to the ControlFlow2DataFlow case study (see Section 2.1.3).

Table 3.1 – Current limitations in MTs approaches

PARADIGM		EXEC.MODE		CONCURRENCY		EMF INTEG.
Graph	Rel.	Shared	Dist.	Mem.	Disk	
[95, 132, 84, 63]	[31, 129]	[129, 63, 31]	[95, 31, 132, 84]	[129, 31, 63]	[72]	[129, 72]

3.2 Conceptual Framework

3.2.1 Global overview

In a previous work, Clasen et al. [37] draw the first lines towards a conceptual framework for handling the transformation of VLMs in the Cloud. Their proposal focuses essentially on two bricks, *model access and persistence*, and *model transformation* execution in the Cloud. While this work investigated different possible alternatives to transforming VLMs models in the Cloud, in our proposal, we opt for parallel execution of model transformation following a data-distribution scheme. We argue that, this approach is the most suitable one as graph computations is often data-driven and dictated by the structure of the graph. Moreover, thanks to the properties of the ATL language (below), interactions among rules is reduced. . This solution offers more flexibility to the transformation engine as most operations on graphs are data-driven [90]. More details are to come throughout this chapter.

Figure 3.1 shows a global overview of our distributed transformation framework by means of a simple example. The transformation is composed of a simple rule that changes the shape of nodes (from Square to Hexagon), but keeps the same ID as well as graph topology. Our distributed cluster is composed of a single master node, data nodes, and task nodes. Data nodes and task nodes communicate with each other through a distributed MOF-compliant model access and persistence API. While task nodes are responsible for transforming the input model or composing the output one, data nodes are responsible for hosting the partial input/output models. Although in our illustration we differentiate between data nodes and task nodes, in real world clusters, data can be processed in the same node it resides in.

Our distributed model transformation is divided in three steps, (i) *data distribution*, (ii) *parallel local transformation*, and (iii) *parallel composition*. The coordination phase plays the role of a barrier in which task nodes share their output data between each other for a global model composition.

In the first phase, the master node is responsible for assigning source model elements to task nodes (*data distribution*) for computation. Each task node is responsible for executing the rule application triggered by the assigned elements. These subsets (a.k.a. chunks, splits, or partitions) are designed to avoid any redundant computation in two separate nodes. Moreover, as the master node is limited by its memory, we consider a lightweight and memory-friendly assignment mechanism of source model elements. In our example, the master node assigns {a, b, c, d} to the upper node, and the remaining to the second one (as shown in Figure 3.1).

The data distribution step is crucial for the execution performance and workload balancing (See Chapter 4). The transformation workload assigned to every machine

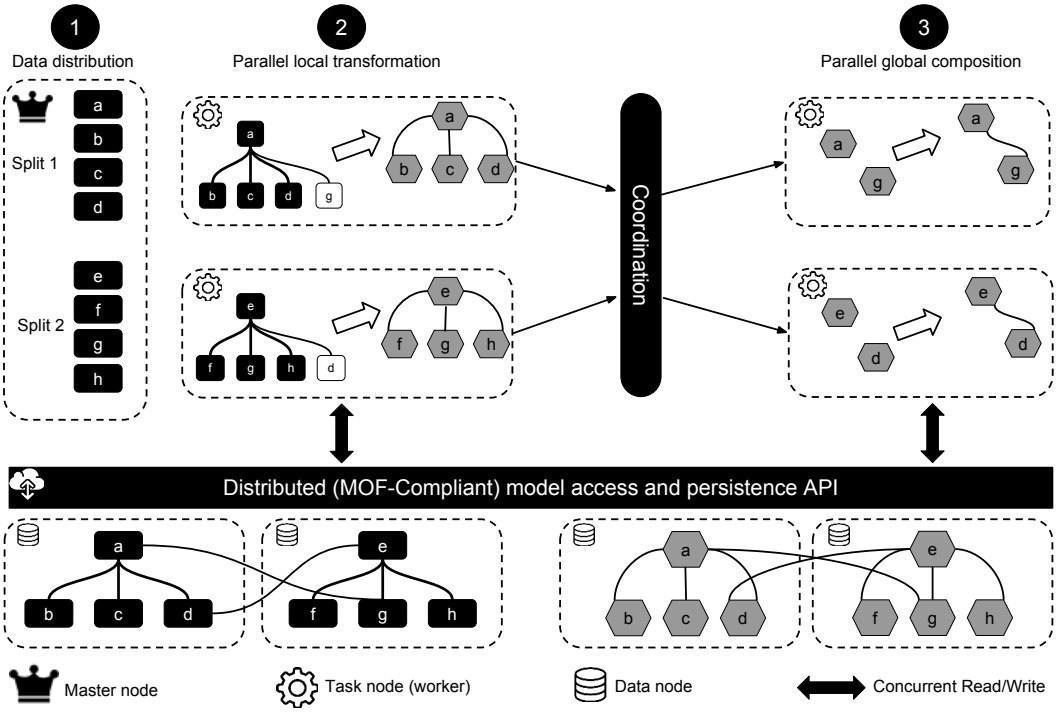


Figure 3.1 – A conceptual framework for distributed model transformation

depends heavily on how data is distributed over the task nodes. As a result of a naive data distribution, machines with lighter workload have to wait for other machines to complete. At other times, machines with heavy workload can crash when hitting their memory limit. Therefore, the master node supports an efficient data distribution strategy, which in one hand, assigns partitions with an even workload, and on the other hand, does not overload the task nodes.

After assigning source model elements to task nodes, they start processing the transformation in parallel. However, due to the complexity of the pattern matching phase, a task node may have to load additional elements in order for its local transformation to complete (*parallel transformation*). For instance, in our example, each square needs to traverse the set of its direct neighboring nodes to reproduce the graph topology. In particular, the upper task node needs the elements "g" while the second node "d". The set of additional elements is known only at runtime, hence, our persistence framework transparently provides task nodes with any input element, at any point in time during the transformation.

At the end of the parallel transformation phase, each task node will result in a local output sub-model together with a set of tracing information that has a twofold role. This information does not only contain information about computation already performed, but also about the one that needs to be performed in order to compose local output sub-models into a global output model.

The coordination phase plays the role of a synchronization barrier, where, once all task nodes finish their parallel transformation, they exchange trace information about the missing links in order to compose the global model. To avoid network congestion, target sub-models are stored in the backend, and only a lightweight serialization of tracelinks is passed through the network.

In the final phase (*parallel composition*), missing tracelinks are evenly split to task nodes, and relationships are established between the output sub-models to compose the final output model. The persistence framework enables task nodes to concurrently establish links between any given model elements. Thus, the persistence framework enables task nodes to store any output element at any point in time; during the transformation execution.

In what follows we exhibit the framework's capabilities in the form of requirements, then we organize them by components (distributed engine, and distributed persistence framework). The aim of this process is to highlight the main properties of our distributed transformation system, then propose a formalization of our system.

3.2.2 Distributed transformation engine

According to the description of our framework (above), the distributed model transformation component guarantees the following points:

1. the assigned subsets must be carefully partitioned in a way that prevents from running the transformation of the same elements more than once, on different machines
2. the transformation engine must support an efficient data partitioning algorithm
3. the union of output elements must result in the same amount of model elements as if the transformation were run sequentially at the end of the parallel transformation phase

In typical relational MT engines (e.g., the standard ATL and ETL engines), the transformation execution starts by loading the input model. Then the engine applies the transformation by selecting each rule, looking for matches corresponding to the input pattern, and finally generating the appropriate output elements [59]. Each execution of a matched input pattern is called a rule application.

From now on, we denote by $\mathcal{E}$ a set of model elements, and $\mathcal{M}$ a set of commodity machines in a distributed system $\mathcal{S}$.

Definition 1 *Let $\mathcal{R}$ be a set of model transformation rules. A rule application is defined by the tuple $(e, rule, d_e)$, where:*

— *$e \in \mathcal{E}$, is the element triggering the execution of the rule application*

— *$rule \in \mathcal{R}$, is the rule whose input pattern is matching the element e*

— *$d_e \subseteq \mathcal{E}$, the subset of model elements needed to execute the rule application triggered by e*

Given Definition 1, we consider a MT execution job as the union of elementary rule application execution jobs, where each job is responsible for transforming a single model element. In the case of rules with n-ary input pattern (matching a sub-graph), we consider the job of applying the rule to be primarily triggered by one input pattern element (*e* in Definition 1). A rule application as defined in Definition 1 may occur more than once, since a model element might be triggering more than one rule application, either of the same rule or a different one. Selecting a primary triggering element for each rule application ensures that, after distributing the source model, a rule application occurs on only one machine, i.e. the one responsible for transforming the triggering element.

The distribution of a transformation based on a data-parallel distribution approach over m machines ($m = |\mathcal{M}|$), consists of dividing the input model into m splits, and assigning disjoint sub-models of the input model to different machines. Each machine will be then responsible for transforming the assigned subset. In what follows we refer to this set of elements assigned to a machine i by $\mathcal{A}_i$. Given a system $\mathcal{S}$ of m machines, the set of assigned elements has the following property:

Property 1 *Each element $e \in \mathcal{A}_i$ is assigned to one and only one set ($\forall i, j \in \mathcal{M},\ i \neq j \rightarrow \mathcal{A}_i \cap \mathcal{A}_j = \emptyset$)*

We also define as $\mathcal{D}_i$, i.e. dependencies of $\mathcal{A}_i$, the set of source elements that need to be accessed to compute the transformation of $\mathcal{A}_i$. This set is determined as the union of the subsets of model elements needed to execute the rule applications triggered by every element e assigned to machine i:

$$\mathcal{D}_i = \bigcup_{e \in \mathcal{A}_i} d_e$$

The presence of $\mathcal{D}_i$ is mandatory for the transformation of the assigned model elements $\mathcal{A}_i$. Consequently, every machine i needs to load all the elements $\mathcal{L}_i$ belonging to $\mathcal{A}_i \cup \mathcal{D}_i$.

This is similar to a well-known problem on graphs community by *Graph-Based Data Clustering with Overlaps* [50]. This problem allows clusters overlapping by duplicating (to a particular extent) graph vertices or edges. In our system, the overlapping elements w.r.t. to i are $\mathcal{O}_i = \mathcal{L}_i \setminus \mathcal{A}_i$ (Property 1).

3.2.3 Distributed model persistence

Storing models in the Cloud is a good solution to break down the complexity in handling VLMs. However, the distributed persistence framework should be compliant with existing modeling tools. Moreover, distribution-derived features, such as the hosting data node location, and network communication should be hidden behind the persistence component.

Additional requirements that must be guaranteed (according to our framework's description) by a well-suitable distributed model persistence component are:

— *Lightweight serialization/de-serialization mechanisms of data communicated across the network*. Exchanging a large amount of data across the network could be a bottleneck. One way of overcoming this could be adopting a compact representation of exchanged data, which is in our case, input model elements and intermediate tracelinks. This point could also be handled by the transformation engine.

— *Concurrent read/write from/to the underlying backend*. Every task node should be able to fetch, on demand, any model element from the underlying backend. Same for concurrent writes to the underlying storage. Nonetheless, the framework should preserve ACID properties for a consistent and sound resulting model.

This leads us to define the following properties:

Property 2 `on-demand loading` *to ensure that only needed elements are loaded, especially when the model is too big to fit in memory*

Property 3 `concurrent read/write` *to permit different machines accessing and writing into the persistence backend simultaneously*

In order to improve the performance of our framework by decreasing the amount of communication between data nodes and task nodes, we extend our framework by the following property:

Property 4 `fast look-up of already loaded elements`*, this can be made possible using caching and/or indexing mechanisms*

3.3 Model Transformation on MapReduce

3.3.1 Distribution of Relational Model Transformation

Distributed model-to-model transformation inherits most of the well-known challenges of efficient parallel graph processing, such as poor data locality and unbalanced computational workloads. In particular, implicit data distribution is not trivial for transformation languages where rules applied to different parts of the model can interact in complex ways with each other. The higher is the number of these interactions, the bigger is the volume of data communicated across the network, both at inter and intra-phase levels. In MapReduce, this turns into, more data to serialize, de-serialize, and less network throughput due to congestion.

As result of ATL's execution semantics, especially, four specific properties of the language (below), we argue that inter-rule communication is made discernible, and the odds of running into race conditions are minimized. More precisely, interaction among ATL transformation rules are reduced to bindings resolution, where a target element's feature needs to reference to other target elements created by other rules:

Property 5 `Locality:` *Each ATL rule is the only one responsible for the computation of the elements it creates, i.e., the rule that creates the element is also responsible for initializing its features. In the case of bidirectional references, responsibility is shared among the rules that create the source and the target ends of the reference.*

Note that if a transformation language does not satisfy this property, a way to lower the data communication cost would be by making sure that different rules sharing update task reside on the same machine.

Property 6 `Single assignment on target properties:` *The assignment of a single-valued property in a target model element happens only once in the transformation execution. Multi-valued properties can be updated only for adding values, but never deleting them.*

If a language does not guarantee this property, one way of communicating less data is by locally aggregating ⟨key,value⟩ pairs responsible for updating a single-valued feature of an element. Let's take for instance the example of a rule that, for every rule application

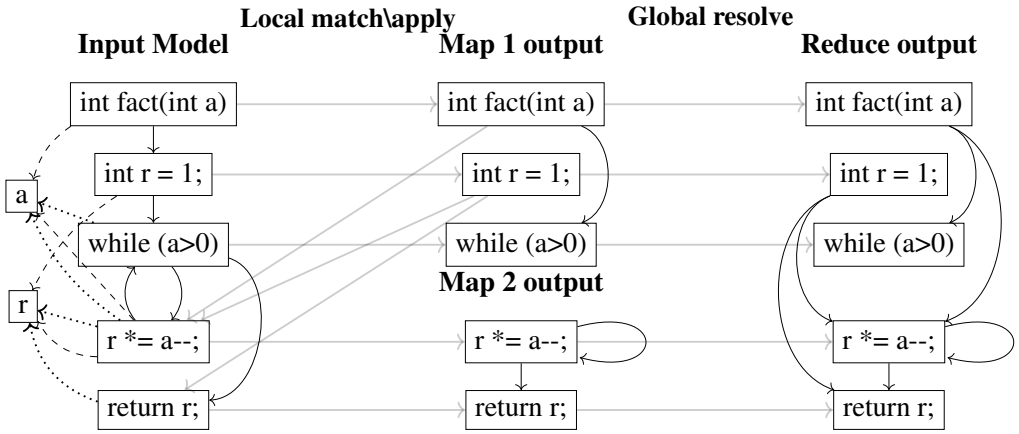


Figure 3.2 – ControlFlow2DataFlow example on MapReduce

increments a variable, instead of sending a bunch of pairs, it would be recommended to aggregate these pairs and send only a single one having as a value the sum of the increment operations.

Property 7 *`Non-recursive rule application`: Model elements that are produced by ATL rules are not subject to further matches. As a consequence, new model elements can not be created as intermediate data to support the computation.*

This differentiates ATL from typically recursive graph-transformation languages. The property should not be confused with recursion in OCL helpers, that are responsible for intermediate computations over the source models, not target ones.

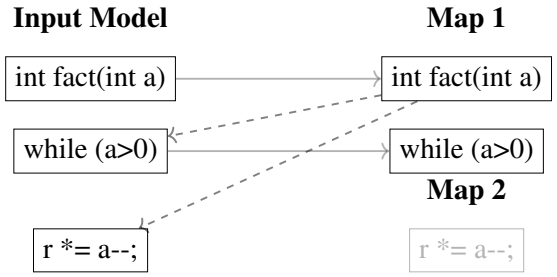
Property 8 *`Forbidden target navigation`: Rules are not allowed to navigate the part of the target model that has already been produced, to avoid assumptions on the rule execution order.*

This property is possible thanks to the resolve algorithm. A way to workaround the non-satisfaction of this property, is by making sure that the target elements creation and update happen in two different phases.

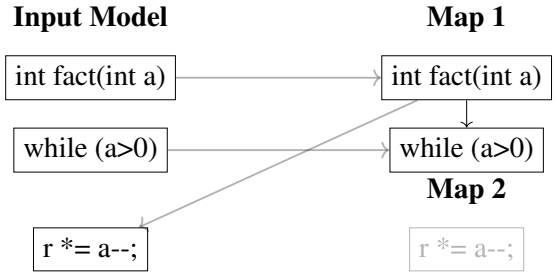
These properties strongly reduce the possible kinds of interaction among ATL rules, and allow us to decouple rule applications and execute them in independent execution units.

3.3.2 ATL and MapReduce Alignment

Mostly, distributed (iterative) graph processing algorithms are data-driven, where computations occur at every node of the graph as a function of local graph structure and its intermediate states. The transformation algorithm in ATL could be regarded as an iterative graph processing algorithm with two phases (match and apply described in Section 2.1.3) and one intermediate state (matched). Each node of the cluster that is computing in parallel takes charge of transforming a part of the input model.



(a) results of dfNext binding for {int fact (int a)}



(b) Resolving the results of dfNext binding for {int fact (int a)}

Figure 3.3 – Local resolve of dfNext for {int fact (int a)}

In our approach we propose a mapping that aims at reducing cross-machines communication cost. That is by adopting some good practices for scalable graph processing in MapReduce. This is made possible, thanks to the alignment of the ATL distributed execution semantics with MapReduce described below.

As an example, Figure 3.2 shows how the ATL transformation of our running example could be executed on top of a MapReduce architecture comprising three nodes, two maps and one reduce workers. The input model is equally split according to the number of map workers (in this case each map node takes as input half of the input model elements). In the map phase, each worker runs independently the full transformation code but applies it only to the transformation of the assigned subset of the input model. We call this phase *Local match-apply*. Afterwards each map worker communicates the set of model elements it created to the reduce phase, together with trace information. These trace links (grey arrows in Figure 3.2) encode the additional information that will be needed to `resolve` the binding, i.e. identify the exact target element that has to be referenced based on the tracing information. The reduce worker is responsible for gathering partial models and trace links from the map workers, and updating properties value of unresolved bindings. We call this phase *Global resolve*.

In the following (i) we describe the distributed execution algorithm we propose for ATL decomposing it in the *Local match-apply* (parallel trasformation phase in Section 3.2) phase assigned to mappers and the *Global resolve* (model composition phase in Section 3.2) phase assigned to reducers; (ii) we define the trace information that needs to be passed between mappers and reducers to allow the re-composition of the global model after distribution.

Algorithm 2: Map function

input : *Long key, ModelElement element*

```
1  foreach rule ∈ getApplicableRules(element) do
2      if isMatched(element, rule) then
3          link ← createLink(element, rule);
4          addLink(linkSet, link);
5          foreach binding ∈ getBindings(link) do
6              if isAttribute(binding) then
7                  applyAttribute (binding);
8              else
9                  foreach computedElement ∈ computeBindingExp(binding) do
10                     if isLocal(computedElement) then
11                         resolvedElement ← resolveTarget(computedElement);
12                         applyReference(resolvedElement, binding);
13                     else
14                         addElementToTrace(computedElement, binding);
15         storeLink(generatedKey, link);
           // generatedKey to decide to which reducer this link
              will be assigned
```

Local Match-Apply

At the beginning of the phase, input splits are assigned to map workers. Each of these splits contains a subset of the input model for processing. Although, each worker has a full view of the input models in case it needs additional data for bindings computation. This is due to data-locality, which particularly occurs when links between vertices can be irregular. Note that while intelligent assignment strategies could improve the algorithm efficiency by increasing data locality, by default we perform a random assignment. Intelligent assignment strategies for model elements, especially based on static analysis of the transformation code and data access patterns are investigated in the next chapter. In our example, the elements {'`int fact(int a)`','`int r = 1;`','`while (a>0)`','`a`',} are assigned to machine 1, while the rest is sent to machine 2.

The pseudo-code for the processing in this phase is given in Algorithm 2. For every model element in the split, the map function verifies if a rule guard matches and in this case instantiates the corresponding target elements (line 3), same as in the regular execution semantics (Section 2.1.3). In the case of rules that match multiple elements, the map function would consider the element of the split as the first element [2] of the matched pattern, and look for combinations of other elements satisfying the guard. For instance, in Figure 2.6 the `Method` and `FlowInstr` rules instantiate the method signature and the instructions that define or use variables (all the instructions of the example). Variables (*a* and *r*) are not instantiated since no rule matches their type. For each instantiated output element, a trace link is created connecting source and target elements of the applied rule. For example, executing the binding *dfNext* over the method *fact(int a)*, results in {*while(a)>0*, *r*=a--;*} (dashed grey arrows in Figure 3.3 (a)).

2. The first input pattern element as specified in the ATL transformation rule

Subsequently, the algorithm starts processing the list of property bindings for the instantiated target elements. We extended the behaviour of the `resolve algorithm` to enable handling elements transformed in other nodes, we call this algorithm `local resolve`. In the case of attribute bindings, the same standard behaviour is preserved, the OCL expression is computed and the corresponding feature is updated accordingly (lines 6-7). While bindings related to references connecting elements transformed by different rule applications, potentially on different nodes, the resolution is performed in two steps: (i) the OCL expression of the binding computes to a set of `elements in the source model` and ATL connects the bound feature to these source elements using trace links; (ii) the source-models elements are resolved, i.e. substituted with the corresponding target element according to the rule application trace links. If source and target elements of the reference are both being transformed in the same node, both steps happen locally (lines 10-12), otherwise trace links are stored and communicated to the reducer, postponing the resolution step to the `Global resolve` phase (lines 11). Going back to our example in Figure 3.3, the result of executing the binding *dfNext* was {*while(a)>0*, *r*=a--;*}. Since *while(a)* and *fact(int a)* reside in the same node, a *dfNext* reference between them is created in the target model. On the contrary, a trace property is created between *fact(int a)* and *r*=a--;* because they belong to different nodes (Figure 3.3(b)).

Global Resolve

At the beginning of the reduce phase, all the target elements are created, the local bindings are populated, and the unresolved bindings are referring to the source elements to be resolved. This information is kept consistent in the tracing information formerly computed and communicated by the mappers. Then each reducer is responsible for resolving the set of assigned bindings. This proceeds by iterating over the trace links and simply refer to the corresponding target model elements, as depicted in Algorithm 3. For each trace link, the reducer iterates over the unresolved elements of its property traces (line 4), resolves their corresponding element in the output model (line 5), and updates the target element with the final references (line 6). In the right-hand side of Figure 2.6 all the trace properties have been substituted with final references to the output model elements.

Algorithm 3: Reduce function

```
input : String key, Set<TraceLink> links
1 foreach link ∈ links do
2     foreach property ∈ getTraceProperties(link) do // unresolved
          properties
3
4         foreach elmt ∈ getSourceElements(property) do
5             resolvedElement ← resolveTarget(elmt);
6             updateUnresolvedElement(property, resolvedElement);
              // equivalent to applyReference (element, binding)
                 in map function
```

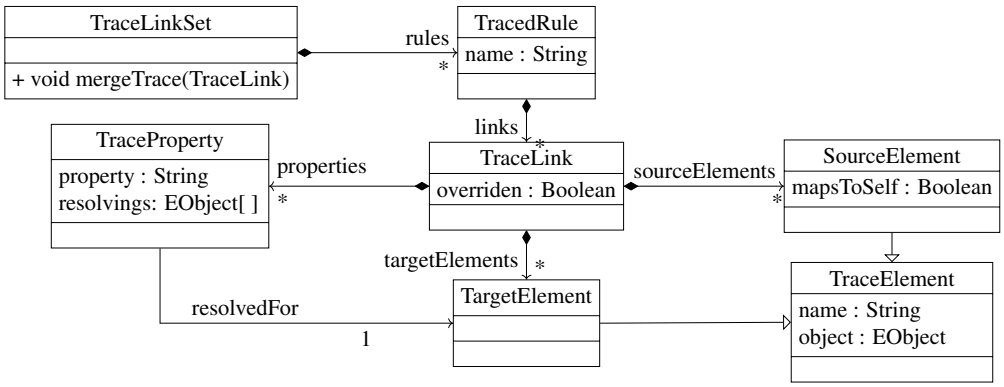


Figure 3.4 – Extended Trace metamodel

Trace Metamodel

MT languages like ATL need to keep track during execution of the mapping between source and target elements [140]. We define a metamodel for transformation trace information in a distributed setting (see Figure 3.4).

As in standard ATL, traces are stored in a *TracelinkSet* and organized by rules. Each *TracedRule* is identified by its name, and may contain a collection of trace links. A link maps a set of source pattern elements to a set of target pattern elements. Both source and target pattern elements are identified by a unique name within a trace link (same one in the rule). Likewise, source elements and target elements refer to a runtime object respectively from input model or output model. This layered decomposition breaks down the complexity of traversing/querying the trace links.

This trace information (source elements, rule name, and target elements) is not sufficient for the distributed semantics, that requires transmitting to the reducer trace information connecting each unresolved binding to the source elements to resolve. Thus, we extended the ATL trace metamodel with the *TraceProperty* data structure. Trace properties are identified by their name that refers to the corresponding feature name. They are contained in a trace link, and associated to the source elements to be resolved along with the target element to be updated.

3.4 Model Consistency in distributed MT with ATL

Distributed computation models like MapReduce are often associated with persistence layers for accessing distributed data. In solutions for Big Data, where scalability is the main requirement, persistence layers sacrifice ACID properties in order to achieve better global performance. In networked shared-data systems, this is also known as *Consistency, Availability and Partitioning tolerance (CAP)* theorem [26]. Such properties need to be implemented by the application layer on a need basis.

The *CAP* stands for: *Consistent*, all replicas of the same data will be the same value across a distributed system; *Available*, all live nodes in a distributed system can process operations and respond to queries; *Partition Tolerant*, the system is designed to operate in the face of unplanned network connectivity loss between replicas. The theorem considers the three properties as continuous. While availability scales from 0% to 100%, consistency

and partitioning come in the form of nuances. A typical definition says that you cannot have strong consistency and 100% availability in the presence of partition tolerance. This theorem constrains a system designer to tune the properties he needs to preserve against the ones he wants to sacrifice.

Hereafter, we reason about the execution semantics of ATL, especially, atomic model manipulation operations (as coming from the MOF specification). Our objective is to exhibit the minimal set of ACID properties to be guaranteed for a *consistent* output model. This reasoning is conducted while taking into consideration the model access modes and the possible interactions between model manipulation operations.

As no definition for a transaction has been carried in the formal MOF specification, in our study, we consider that *each model manipulation operation on a single element runs in a separate transaction*. This definition is similar to the one adopted by the CAP theory. In our analysis we rely on the semantics of the MOF reflection API, especially the methods used by the ATL engine. A brief description of the MOF's Reflection API can be found in Section 2.1.2.

3.4.1 Consistency in distributed atomic model manipulation operations in ATL

Table 3.2 – Access modes in ATL transformations

	MATCH	APPLY
Input	READ-ONLY	
Output	WRITE-ONLY	
Trace	WRITE-ONLY	READ-ONLY

In model-to-model transformations with ATL, models are never accessed in read-write mode but only in read-only mode or write-only mode. Precisely, while source models are always read-only and target models are always write-only, trace models are, depending on the phase, either read-only (apply phase) or write-only (match phase). Table 3.2 summarizes different access modes per model- kind\phase.

Moreover, in our distributed transformation framework, we identify two different scopes of computation, depending on the phase. The computation can run either in a local scope (*Local match/apply*), or a global scope (*Global resolve*). In the local scope, all the write operations happen in local. Data is always created and manipulated by the same task node. As a result, model manipulation operations on the same model element never run concurrently. On the other hand, concurrency may take place in the global scope, and different machines can modify the same element at the same time.

Regardless of the computation scope, models that are accessed in read-only mode are subject only to side-effect free queries [3]. Likewise, during the local scope, trace models are always consistent because they are considered as local intermediate data. In the remaining of this section, we are interested only in the global scope, in particular, the output models.

Output models undergo a building process creating a model that grows monotonically (Property 6 and 7). However, during the global computation, there exist some specific

3. The accessed data is always consistent since no write operation occurs

cases that may lead to inconsistency. In particular, (i) when having operation invoking a change in more than one element (e.g. containment and bidirectional references), or (ii) updating multi-valued references, for instance, when moving objects from the `root` element of the resource [4] (when newly created) to its new container (when setting up links). On these terms, and despite Property 8, the object should be first removed from the `root`, then added to the new container. Similarly, in the case of updating a containment or a bidirectional reference. These operations need to either entirely fail or succeed.

Table 3.3 depicts the set of operations in the MOF Reflection API as well as the maximum read/write operation count. Note that, operations on full objects such as elements creation are not considered. Finally, the table depicts the ACID properties that need to be fulfilled when the operation is performed during model transformation, in order to guarantee the correctness of the output model. The properties between parenthesis can be relaxed [5], while the others should be strongly preserved.

ACID properties do not behave completely orthogonally with each other. More importantly, Atomicity plays an important role by rolling back in case of a system failure or a consistency rule violation. Furthermore, since output models are accessed in write-only mode, the consistency property can be checked only at the end of the transformation, instead of at every atomic operation. Durability is also an important property to be preserved, but is strongly coupled to the database.

Following our discussion in Section 3.3.1, we showed how ATL, thanks to its properties, reduces inter-rules interactions. Beyond lowering data communication over the network, the interest of these properties extends to reducing the chances of running into concurrent modifications, and thus the *Isolation* property can be relaxed for some operations. Especially, thanks to Property 6, updates on single-valued properties occur only once in the transformation lifetime. In the case of updates on multi-valued properties, the chances of having concurrent updates are definitely considerable. Nevertheless, in ATL only *add* or *remove* operations might run into concurrent updates. The distributed persistence framework should make sure that two concurrent *add* or *remove* operations will leave the system in a consistent state [6].

As noticed in the *Properties* column, `Durability` and `Consistency` are two mandatory properties to be preserved. The correctness of the output results is tied to guaranteeing these properties in every single method. Supposing that the underlying backend guarantees ACID properties at the finest-grained level (single CRUD operation), methods involving updates in more than one element (two-phased commits) need to be atomic, while update operations on multi-valued properties (`add` and `remove`) need to be isolated. These methods should execute in two steps, first the latest value is looked-up, then the property's value is updated according to the method behaviour. However, thanks to the monotonical building process of models in ATL, even if a two phased commit does not happen in the same transaction, the model will eventually be consistent (i.e. we will end up having the same output model). Relaxable properties are depicted between parenthesis. In case one of these methods fails, the system should rollback.

4. A resource can be considered as an index for root elements
5. Supported only in specific scenarios
6. It is worth mentioning that in MOF, only type, cardinality, and default properties are natively checked for Consistency. Model constraints, described as OCL invariants, are validated only when invoked.

Table 3.3 – Summary of accesses counts of MOF Reflection operations

METHODS	MAX COUNT*		PROPERTIES
	READ	WRITE	
OPERATIONS ON PROPERTIES			
`get*`	1	0	_C_D
`set*`	2	2	AC_D
`isSet*`	1	0	_C_D
`unset*`	0	1	_C_D
OPERATIONS ON MULTI-VALUED PROPERTIES			
`add`	2	2	AC(I)D
`remove`	2	2	AC(I)D
`clear`	0	1	_C_D
`size`	1	0	_C_D

* Note that only max access count is taken under consideration

3.5 Conclusion

In this chapter we presented our conceptual framework for distributed model transformations with the relational transformation language ATL. We identified the essential bricks for achieving distributed model transformation. Then, we determined the system's requirements that should be guaranteed in order to have a consistent output model. Moreover, we showed that model transformation with rule-based languages like ATL is a problem that fits in the MapReduce execution model.

While frameworks such as MapReduce and Pregel lack a high-level transformation language, several high-level query languages have been proposed. Their software stack is usually composed of a standalone distributed system, an execution engine, and a language that targeting the execution engine. In our approach we used a similar stack, being a persistence layer on top of a standalone distributed system, a distributed model transformation engine, and finally, a declarative model transformation language. Hence, the declarative MT paradigm can be thought of as a high level abstraction for data transformation on MapReduce.

We exhibited specific properties of the ATL language that make the alignment possible and the resulting solution efficient, then we introduced a semantics for ATL's distributed execution on MapReduce. The proposed solution hides completely the internal complexities related to parallelism. The syntax of the MT language is not extended with distribution primitives. Finally, we examined the execution semantics of ATL against the Reflective MOF specification in order to extract the minimal set of ACID properties guaranteeing a consistent output. Our analysis was based on the semantic of MOF operations and their possible interaction inside the ATL engine. Our approach is validated through the implementation and evaluation of a distributed engine, ATL-MR.

Our proposed engine is inherently integrable with the EMF-modeling ecosystem. Moreover, thanks to our data distribution scheme, we are able to support complex model transformation with multiple input pattern elements. At last, we have shown the requirements for supporting minimal set of ACID properties on model transformations in ATL in order to handle concurrent read/write on output models efficiently. As a result of these features, our solution responds to the limitations exhibited by related work. Implementation details as well as a concise evaluation are to be found in Chapter 6 and Chapter 7 respectively.

In Section 3.2 we identified the two essential components for efficient distributed model transformations. In the next chapters, we introduce our solution for efficient data partitioning which aims at balancing the machines workload and increasing data locality. We also present a distributed persistence backend suitable for distributed model transformations in ATL. This brick enables concurrent read/write on in(out)put models.

Partitioning VLMs for distributed transformations. With a naive data-distribution scheme, the execution time can be monopolized by the wait time for model elements lookup. With an unbalanced distribution, machines with light workload have to wait for other machines to complete. Other times, machines with heavy workload can crash when hitting their memory limit. Such challenges are not only limited to model transformations, but extend to several distributed graph processing tasks [90]. An algorithm should be defined to overcome these limits and assist the partitioning process. In Chapter 4 we propose an algorithm based on static analysis of relational transformations.

Accessing and persisting VLMs for distributed transformations. Data distribution can be considered a crucial phase in the distribution of a model transformation, in particular, the parallel transformation phase. Models are usually densely interconnected, and MTs may contain rules with very different complexity. Moreover, having XMI as its default serialization format, QVT-like engines, in particular ATL, are facing several scalability issues when handling VLMs. This is due to this file-based representation XMI adopts. In Chapter 5, we propose our solution to store model-data in (distributed) NoSQL databases, and we show how we support the mandatory ACID properties to be preserved.

4

Efficient data partitioning for distributed MTs

As we showed in the previous chapter, popular distributed programming models like MapReduce [45] may simplify the development of distributed model transformation. However, some characteristics of MT make efficient parallelization challenging. For instance, typical MapReduce applications work on flat data structures (e.g. log entries) where input entries can be processed independently and with a similar cost. Hence, a simple data distribution scheme can perform efficiently. Conversely, models are usually densely interconnected, and MTs may contain rules with very different complexity. Moreover, most of the computational complexity of MT rules lies in the pattern matching step, i.e. the exploration of the graph structure.

Because of this, model transformations witness higher ratio of data access to computation w.r.t. typical scientific computing applications. With a naive data-distribution scheme, the execution time can be monopolized by the wait time for model elements lookup. With an unbalanced distribution, machines with light workload have to wait for other machines to complete. At other times, machines with heavy workload can crash when hitting their memory limit. Such challenges are not only limited to model transformations, but extend to several distributed graph processing tasks [90].

Several task-parallel [35] and data-parallel [120] distribution techniques have been proposed, but none of them addresses the specific case of relational model transformation. In this chapter, we argue that when a MT is defined in a relational language, an efficient data-distribution [1] scheme can be derived by statically analysing the structure of the transformation rules.

From static analysis, we derive information on how the input model is accessed by the transformation application, and we encode it in a set of so-called *transformation footprints*. Footprints allow us to compute an approximation of the transformation (task-

1. a.k.a. data partitioning

data) dependency graph [2]. We exploit this graph in a data distribution strategy, minimizing the access to the underlying persistence backend, and hence, improving the performance of our model transformation execution and memory footprint.

We adapt existing formalization of uniform graph partitioning to distributed MTs using binary linear programming. The formalization takes into account task-data overlapping maximization based on the dependency graph. Given a cluster of commodity machines, the objective is to minimize the amount of loaded elements in each of these machines. Then, we propose an algorithm for the extraction of transformation footprints based on the static analysis of the transformation specification. Finally, we propose a fast greedy data-distribution algorithm, which partitions an input model over a cluster, based on the extracted footprints.

The rest of the chapter is structured as follows. Section 4.1 describes related work. Section 4.2 motivates the need of efficient partitioning strategies for distributed MTs by the means of a running example. Section 4.3 formalizes our problem using Integer Linear Programming. Section 4.4 introduces our footprints extraction algorithm, and our greedy algorithm for data distribution. Section 4.5 concludes the chapter.

4.1 Related work

Performance improvement in distributed systems is a well-studied problem. Impulsed by different and yet sound motivations, different families of approaches emerge. As in our distributed transformation framework we focused on the MapReduce programming model, in what follows we introduce generic related work on makespan [3] minimization in MapReduce. Later, we describe existing approaches to improve model transformation and query in the MDE field (from Section 3.1) by using built-in indexers. Finally, we describe other applications of static analysis of model transformation for different purposes.

Task-driven scheduling approaches

Observing that the order on which tasks are executed influences the overall completion time, some works [143, 139, 36] proposed scheduling heuristics to improve the system makespan, especially on MapReduce. Each of these heuristics considers one or more specific factors such as the cost matrix, task dependency, and machines heterogeneity.

Assuming that map tasks are usually parallelizable but not reduce tasks, Zhu et al. [143] introduce an offline and bi-objective scheduler to minimize makespan and total completion time of reduce tasks. Algorithms for both of preemptive and non-preemptive have been developed proving a performant worst ratio in accordance with the number of machines in the system.

HaSTE [139] is a Hadoop YARN scheduling algorithm aiming at reducing the MR jobs' makespan based on task dependency and resource demand. The scheduler consists of

2. dependency graph is a directed graph representing the computation dependency of every objects towards the nodes needed to perform the transformation execution

3. Makespan is the workload execution time, which is the difference between the earliest time of submission of any workload task and the latest time of completion of any of its tasks

two components, an initial task assignment and a real time task assignment. The first one is intended to assign the first batch of tasks for execution while the rest remains pending. A greedy algorithm using dynamic programming is used for this regard. The real time task assignment component is triggered with resource capacity update. The scheduler then decides with tasks of the remaining ones to assign considering two metrics, the resource availability and the dependency between tasks.

In another work, Chen et al. [36] propose a task scheduler that joins all three phases of MR process. The problem assumes that tasks are already assigned to the machines. Though it only deals with tasks execution ordering. A precedence graph among tasks is used to represent tasks that should start after others finish. The model system is formulated in linear programming and solved with column generation.

Data-driven partitioning approaches

Stanton and Kliot [120] compared a set of lightweight algorithms for Streaming Graph Partitioning for Large Distributed Graphs and compare their performance to some well-known offline algorithms. They run their benchmark on large collections of datasets, and showed up considerable gain in average.

Charalampos et al., introduce a framework for graph partitioning, FENNEL [131]. The framework uses a formulation that relaxes the hard constraint on edge cuts, and replaces it by two separate costs, the edges cut and the cluster size. The formulation enables the accommodation of already existing heuristics. The framework was also integrated to Apache Giraph, an open-source implementation of Pregel, a distributed graph processing framework.

Kyrola et al. describes GraphChi, large-scale graph computation system that runs on a single machine by developing a method called Parallel Sliding Windows. It processes a graph from disk, with only a limited number of non-sequential disk accesses, while supporting the asynchronous model of computation. The approach uses graph partitioning to maximize data locality and minimize disk access counts.

Distributed model indexing

In [72], Izso et al. present an incremental query engine in the Cloud, called IncQuery-D. This approach is based on a distributed model management middleware and a stateful pattern matcher framework using the RETE algorithm. The approach has shown its efficiency, but it addresses only distributed model queries while we focus on declarative model transformations.

Hawk [13] is a model indexing framework that enables efficient global model-element-level queries on collections of models stored in file-based version control systems (VCS). It can operate with different file-based VCSs while providing a query API that can be used by model management tools. Hawk is shipped with features enabling its robustness. Namely, incremental model updates, derived attributes indexed attributes, and scoped queries.

While both approaches focus on performance improvement of model querying, our solution is designed for improving distributed model transformations specified in relation

MT languages. Moreover, both tools use built-in indexers to improve the query execution performance. Instead, in our approach we rely on efficient data partition to improve the performance of transformation execution.

Static analysis of model transformations

Static analysis is the analysis of programs code without actually executing them. It is generally used to find anomalies or ensure conformance to coding guidelines. Static analysis has been also applied to model transformations. In our approach, we use static analysis to extract transformation footprints in order to guide our partitioning algorithm. There also exist other known usage such as proving properties of model transformations like the absence of rule conflicts [49], to detect errors [39] or to compute and visualize tracing information [60] to enhance maintainability.

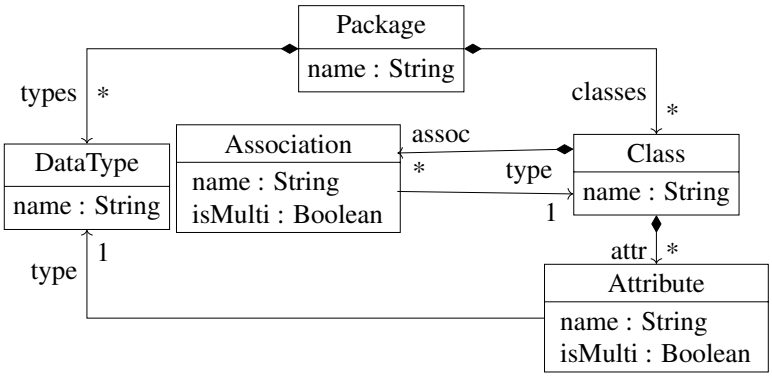
Meta-model footprints are computed using the information gathered via static analysis. In [73] a technique to estimate the model footprint of an operation is presented. Essentially, a given model is just filtered based on the computed static meta-model footprint. In our case, the model footprint is streamed as the original model is traversed. In [101] the type information available in the abstract syntax tree of Kermeta programs is used to generate test models. [30] compute footprints of ATL rules in order to determine which rules are involved in the violation of a transformation contract.

Summary

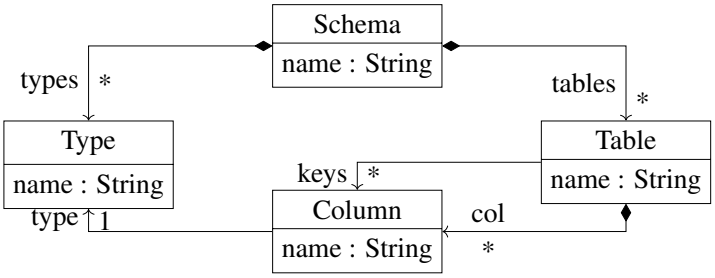
In this section, we introduced existing related work. We started by describing some work on MapReduce task scheduling and data partitioning heuristics. Task-driven approaches support a very coarse-grained definition of tasks (map reduce tasks). Moreover, they do not consider data locality and complex data dependencies. These approaches are not suitable for distributed model transformations due to the variating complexity of rule applications. As these approaches are generic, we believe that our approach can be coupled with them. On the other hand, existing graph partitioning approaches reason only on the adjacency of nodes. In our approach, not only adjacent nodes should reside in the same split, but all dependent nodes. Other existing approaches focus on the improvement of query frameworks by relying on the built-in indexers. To our knowledge, our work is the first to handle efficient data partitioning in relational model transformation. Finally, we illustrated other application to static analysis of model transformations. In our case, we use transformation footprints to compute an approximation of the transformation dependency graph.

4.2 Motivating example

In order to show other application domains to MDE, we choose to motivate the need for intelligent data partitioning using a different case study. Throughout this chapter, we use the Class2Relational transformation, a de-facto standard benchmark for MT. Given a class diagram, this transformation generates the corresponding relational model. Listing 4.1



(a) Class metamodel excerpt



(b) Relational metamodel excerpt

Figure 4.1 – Simplified Class and Relational metamodels

shows a subset of the transformation rules and Listing 4.2 an excerpt of its OCL queries (helpers). The full transformation code can be found on the tool's website [4].

Figure 4.1 refers to the source and target metamodels of the transformation respectively. A *Package* (see Figure 4.1a) contains *classes* and *types*. A *Class* in its turn, contains *attributes* and *associations*. An *Attribute* has a *type*, and can be of cardinality *multi-valued*. Same for *Associations*. In the Relational metamodel (see Figure 4.1b) a *Schema* contains *tables* and *types*. A *Table* contains *columns*, and has *keys*. Finally, a *Column* has a *type*.

The rule *Package2Schema* is responsible for transforming packages to schemas (Listing 4.1). All non abstract classes along with multivalued associations are transformed into tables, and package types are transcribed into *Relational*'s corresponding types. Target patterns defined as OCL bindings depict this mapping. The *Class2Table* shows how tables are produced out of classes. The rule produces two elements. The first one represents the *Table*, while the second one represents its *key*. A set of OCL *bindings* specify how to fill each of the features (attributes and references) of the produced elements. The binding at line 18 copies the name of the *Class*, the binding at line 19 computes its corresponding columns.

Listing 4.2 shows a simple helper example, *getObjectIdType*. It has as context the transformation *module* itself and returns *DataType* (line 7). The helper *getMultiValuedAssocs* instead has as context a *Package* and returns a list of all multi-valued associations belonging to the package's classes (line 1).

Figure 4.2 illustrates a source model that conforms to the *Class* metamodel. We refer to containment references using the dashed arrows, and type references by continuous

4. https://github.com/atlanmod/ATL_MR/

Listing 4.1 – Class2Relational - ATL transformation rules (excerpt)

```
module Class2Relational;
create OUT : Relational from IN : Class;
rule Package2Schema {
  from
    p : Class!Package
  to
    s : Relational!Schema (
      tables <- p.classes->reject(c | c.isAbstract)
       ->union(p.getMultiValuedAssocs),
      types <- p.types
    )
}
rule Class2Table {
  from
    c : Class!Class (not c.isAbstract)
  to
    out : Relational!Table (
      name <- c.name,
      col <- Sequence{key}
       ->union(c.attr->select(e|not e.multi-valued))
       ->union(c.assoc->select(e|not e.multi-valued)),
      keys <- Sequence{key}
       ->union(c.assoc->select(e|not e.multi-valued))
    ),
    key : Relational!Column (
      name <- c.name+'objectId',
      type <- thisModule.getObjectIdType
   )
```

Listing 4.2 – Class2Relational - OCL helpers (excerpt)

```
helper context Class!Package def :
getMultiValuedAssocs :Sequence(Class!Association) =
  self.classes ->reject(c | c.isAbstract)
    ->collect(cc| cc.assoc->select(a| a.multi-valued))
    ->flatten();

helper def: getObjectIdType : Class!DataType =
  Class!DataType.allInstances()
    ->select(e | e.name = 'Integer')->first();
```

Table 4.1 – Model elements dependencies

MODEL ELMT.	DEPENDENCIES
p1	{p1,c1,a1,c2,t1}
c1	{c1, a1, att1, t1}
a1	{a1,c2}
att1	{att1,t1}
c2	{c2, att2, att3, t1}
att2	{att2,t1}
att3	{att3,t1}
t1	{t1}

arrows. The model is composed of a package (*p1*) containing two classes (*c1* and *c2*). Table 4.1 lists, for each source element, the set of elements that the ATL engine will need to read in order to perform its transformation (*Dependencies*). We denote by $d_{(j)}$ the set of dependencies of j, as referred by the column *Dependencies* in Table 4.1. For instance, $d_{(a1)} = \{a1, c2\}$.

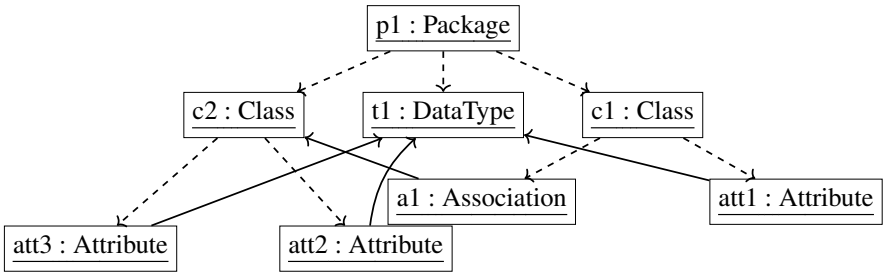


Figure 4.2 – Sample Class model

Considering a system $\mathcal{S}$ of 2 machines (m_1 and m_2) computing in parallel a model transformation. A set of input model elements is assigned to each machine. We argue that, by using an advanced data distribution strategy to the cluster nodes, it is possible to optimize the number of loaded elements, and hence, improve the performance of the model transformation execution.

Let's consider a random uniform distribution scenario [5] of our sample model over $\mathcal{S}$. We can assume that this model is stored in distributed persistence backend, permitting the loading of model elements on-demand. We randomly assign elements {p1,att1,c2,t1} to m_1, and the remaining to m_2. The splits assigned to m_1 and m_2 are denoted by $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. We identify the weight of a split assigned to a machine i ($\mathcal{W}_i$), as the number of elements to be loaded in order to perform its transformation. Assuming that every machine has a caching mechanism that keeps in memory elements that have already been looked-up, loading model elements is considered only once.

Therefore, $\mathcal{W}(\mathcal{A}_1) = |\cup_{e \in \mathcal{A}_i} d_e| = |\{$p1, c1, a1, att1, c2, att2, att3 ,t1$\}| = 8$. Likewise, $\mathcal{W}(\mathcal{A}_2) = |\{$c1, a1, att1, c2, att2, att3, t1$\}| = 7$. As noticed, both machines need almost all the model to perform the transformation of the split assigned to them, resulting in a overall remote access to the underlying persistence backend of 15 (=8+7) elements. However, this naive distribution scenario is not optimal. As the elements {p1,c1,a1,att1} share a heavy dependency between each other, it makes more sense to assign them to the same machine (e.g. m_1). The rest are hence assigned to m_2. This scenario results in better weights. Precisely, $\mathcal{W}(\mathcal{A}_1) = 6$ and $\mathcal{W}(\mathcal{A}_2) = 4$, with an overall remote access of 10 (6+4).

Computing such efficient split becomes especially challenging when we consider model graphs with million of nodes. Without a fast heuristic, the cost of traversing these large graphs to compute the splits may exceed any possible benefit of data distribution.

4.3 Problem formalization

In this section we show how an efficient distribution of a model transformation in a distributed system could be realized to improve the performance of a model transformation execution. We rely on our system's description introduced in Chapter 3.2 to propose a formalization of our system in linear programming.

For the sake of simplicity, we introduce hereafter a brief summary of the main definitions and properties introduced previously.

5. Note that the order in which model elements are transformed does not affect the execution results of the transformation.

— d_e is the subset of model elements needed to execute a rule application triggered by a model element e

— $\mathcal{A}_i$ is the set of model elements assigned to a machine i

— $\mathcal{D}_i$ is dependencies of $\mathcal{A}_i$, the set of source elements that need to be accessed to compute the transformation of $\mathcal{A}_i$: $\mathcal{D}_i = \bigcup_{e \in \mathcal{A}_i} d_e$

Balanced partitioning formalization

As discussed in our conceptual framework, the set of elements to be loaded ($\mathcal{A}_i \cup \mathcal{D}_i$) by a machine i is controlled by the elements to be assigned to i for transformation. The number of loaded elements is determined by the cardinality of $|\mathcal{A}_i \cup \mathcal{D}_i|$. Intuitively we want elements that share most dependencies to be assigned to the same split. This implies minimizing model elements being loaded in different machines, and perform it only once in a single machine (Property 2). Hence, by assigning elements to machines in an intelligent manner, it is possible to reduce the amount of elements to be loaded by each machine. On the other side, it is also important to balance the load across the cluster in order to enable a good scalability of the transformation application. In other words, our objective is to balance the charge of loaded elements while maximizing the overlapping of dependencies.

Balanced graph partitioning is a problem that consists in splitting the graph into partitions of about equal weight, while minimizing the number of cross-partitions edges. In our approach, we define this weight as the cardinality of the set of elements fetched from the persistence backend, and needed to transform the assigned subset. This set is computed using the `transformation dependency graph`, which is a directed graph representing the rule application dependency between model elements. Each element e, has outgoing edges towards all the elements belonging to his set of needed elements d_e. These edges are nothing more but the result of the traversal represented by the pattern matching of a rule application.

Finally, we determine the weight associated to an element e and a machine i, respectively as:

$$\begin{aligned} \mathcal{W}(e) &= |e \cup (d_e)| \\ \mathcal{W}(\mathcal{A}_i) &= |\mathcal{L}_i| = |\mathcal{A}_i \cup \mathcal{D}_i| \end{aligned} \tag{4.1}$$

Balanced model partitioning for distributed model transformation

As discussed previously, our objective is to assign model elements in a way to reduce (minimize) the number of elements loaded in each machine. As rule applications share transformation dependency to model elements, one way to achieve this objective is by minimizing the number of loaded elements while maximizing the overlapping of transformation dependencies.

Consider the transformation of a model of size n ($n = |\mathcal{E}|$) over a system $\mathcal{S}$ of m machines. Our problem can be defined in linear programming as follows:

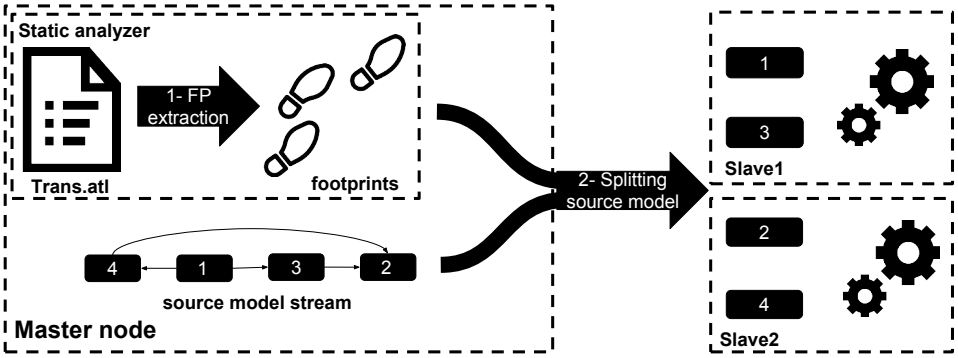


Figure 4.3 – Execution overview of our proposed solution

$$minimize \quad f(X) = \max_{i \in \mathcal{M}} \Big\{ \sum_{j=1}^{n} (\bigvee_{j \in \mathcal{E}} (X_{i,j} \times \mathcal{D}_j)) \Big\} \tag{4.2}$$

where,

$$X_{i,j} = \begin{cases} 0 & \text{if j is not assigned to the machine i ,} \\ 1 & \text{otherwise} \end{cases} \tag{4.3}$$

and,

$$\mathcal{D}_{j,k} = \begin{cases} 0 & \text{if j does not need element k} \\ & \text{for its transformation,} \\ 1 & \text{otherwise} \end{cases} \tag{4.4}$$

subject to:

$$\forall j \in \mathcal{E}, \sum_{i \in \mathcal{M}} X_{i,j} = 1, \tag{4.5}$$

$$\lfloor \frac{n}{m} \rfloor \leq \sum_{j=1}^{n} (X_{i,j}), \tag{4.6}$$

and

$$\forall i \in \mathcal{E}, \lfloor \frac{n}{m} \rfloor + 1 \geq \sum_{j=1}^{n} (X_{i,j}) \tag{4.7}$$

In this formulation, we denote `the transformation dependency graph` by a matrix ($n \times n$), where $\mathcal{D}_{j,k}$ equals to 1 if j needs k, and 0 otherwise. As to $\mathcal{D}_j$, it is a boolean vector representing the projection of this matrix on a single element j (Equation 4.4). The operation $\bigvee$ plays the role of an $inclusive\text{-}or$ over the resulting vector, and returns 1 if the element is needed at least once, and 0 otherwise. This insures that the loading of a given element is considered only once (as stated in Property 4). Finally, The constraint (4.5) is responsible for making sure that a model element is assigned to one and only one machine. Whilst constraints (4.6) and (4.7) restrict the search space of the elements assigned to each machine X_i to even number of elements over all the machines.

4.4 Data partitioning for distributed MT

Although such a problem can be solved by building a full dependency graph and using existing linear programming solvers, their computational cost is not suitable to our scenario. In case of very large graphs, the global execution time would be dominated by (i) the construction of the full dependency graph and (ii) the data distribution algorithm. In this section, we propose a greedy algorithm instead that can efficiently provide a good approximation of the ideal split.

We rely on static analysis of the transformation language to compute an approximated dependency graph. The building process of this graph relies on the transformation footprints. These footprints represent an abstract view of the navigation performed by a given rule application. Footprints are a simplification of the actual computation, and they originate a dependency graph that is a super-graph approximation of the actual one. We then apply a greedy data-distribution algorithm from recent related work to the approximated dependency graph.

The execution overview of our proposed solution is depicted in Figure 4.3. The framework starts by passing the transformation to the static analyser in the master node for footprints extraction. These footprints are fed to the splitting algorithm to assist its assignment decision making. As the model stream arrives, the master decides to which split a model element should be assigned. Once the stream is over, the slave nodes proceed with the transformation execution.

4.4.1 Static analysis of MT: footprint extraction

We analyse rule guards and bindings to extract their footprints. Since we are only interested in navigating the source model, we consider mainly *NavigationCallExp*. The extraction is achieved by traversing the tree representation of the OCL expression symbolizing either the guard or the binding. For instance, Figure 4.4 shows a simplified AST representation of the *getMultivaluedAssocs()* helper, which is illustrated in Listing 4.2. Notice that the AST traversal follows a left-to-right depth-first order traversal.

Table 4.2 – Footprints of Class2Relational transformation

RULES	FOOTPRINTS
Package2Schema	Package.classes.assoc Package.types
Class2Table	Class.assoc Class.attr DataType.allInstances
Attribute2Column	Attribute.type
MVAttribute2Column	Attribute.type Attribute.owner
Association2Column	DataType.allInstances
MVAssociation2Column	Association.type DataType.allInstances

Algorithm 4: Footprint extraction algorithm

```
Input  : ATLModule module
Output : Set⟨ Set⟨ FootPrint⟩ ⟩ perRuleFps
1  foreach rule ∈ getRules(module) do
2      foreach guard ∈ getGuards(rule) do
3          perRuleFps[rule] ∪ {extractFootprint(guard)}
4      foreach binding ∈ getBindings(rule) do
5          perRuleFps[rule] ∪ {extractFootprint(binding)}

6  Function extractFootprint (OCLExp exp)
7      if isInstanceOf(exp, LiteralExp) then
8          if hasSource(exp) then
9              fps := extractFootprint(exp.source)
10         else
11             fps := ∅
12     if isInstanceOf(exp, NavigationOrAttributeCallExp) then
13         if isAttributeCall (exp) then
14             fps := ∅
15         else if isHelperCall(exp) then
16             helper := getHelperByName(exp.name) fps :=
               extractFootprint(exp.source) ⊗ extractFootprint(helper)
17         else //isNavigationCall
18             fps := extractFootprint(exp.source) ◁ exp.refferedProperty
19     if isInstanceOf(exp, OperatorCallExp) then
20         fps := extractFootprint(exp.firstOperand) if
           isBinaryOperator(exp) then
21             fps := fps ⊕ extractFootprint(exp.secondOperand)
22     if isInstanceOf(exp, OperationCallExp) then
23         fps := extractFootprint(exp.source)
24     if isInstanceOf(exp, VariableExp) then
25         if hasContainer(exp) then
26             fps := getType(exp)
27         else
28             fps := ∅
29     if isInstanceOf(exp, IteratorExp) then
30         fps := extractFootprint(exp.source) ⊗ extractFootprint(exp.body)
31     if isInstanceOf(exp, IfExp) then
32         if hasElseClause(exp) then
33             fps := extractFootprint(exp.cond) ⊗
               (extractFootprint(exp.then) ⊕ extractFootprint(exp.else))
34         else
35             fps :=
               extractFootprint(exp.cond) ⊗ extractFootprint(exp.then)
```

```
package.classes -> reject (c | c.isAbstract)
                -> collect (cc | cc.assoc -> select (a | a.multiValued)) -> flatten();
```

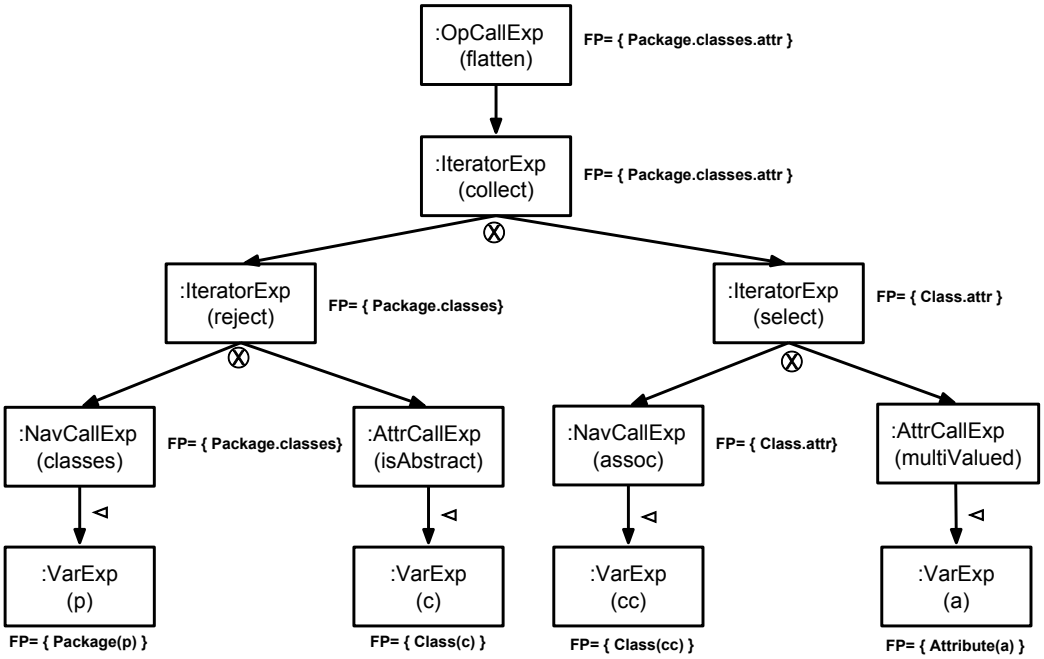


Figure 4.4 – Simplified AST representation of the helper 'getMultivaluedAssocs' augmented with footprint values. For simplicity sake, we add the helper's definition at the top.

Our footprints extraction process is described in Algorithm 4. It proceeds by traversing all the rules of an ATL module (line 1), and computing the footprint of each rule for both guards and bindings (lines 2..5). For each rule, guard footprints and bindings footprints are aggregated. The function *extractRuleFootprint* (line 6) is the one responsible for computing the footprint of an OCL expression. It recursively traverses the AST representation of an OCL expression and returns the corresponding footprint according the node type. Figure 4.4 shows an example resulting from traversing the AST tree of the *getMultivaluedAssocs* helper.

We use three different symbols to represent different operations in our algorithm. The operator $\triangleleft$ is used to denote navigation call chaining. It is used to chain the *source* of a *NavigationCallExp* with its corresponding referred type. For example, the footprint of the *NavigationCallExp* '*classes*' (in Figure 4.4) is, $Package \triangleleft classes$. In our representation, the $\triangleleft$ is denoted by '.', as in OCL (see Table 4.2). Whilst, the operation $\oplus$ is used to decouple the expression to two separate footprints, one corresponding to the LHS of the operator, and the second to its RHS. This operator is mainly used in '*ConditionalExps*', and '*BinaryOperators*'. The $\otimes$ operator instead, behaves like $\triangleleft$ operator when the chaining is possible, otherwise it behaves as a $\oplus$ operator. A chaining is feasible when the type of the OCL expression in the LHS, corresponds to the initiating type in the RHS. For instance, the operator $\otimes$ footprint of the *IteratorCallExp collect* behaves like $\triangleleft$. This happens because the type of the collection of the footprint '*Package.classes*' (in LHS), coincides with the initiating type of the footprint '*Class.attr*' (in RHS). As noticed, the

algorithm does not cover all the set of *OCLExpressions*. Operations on collection types such as *flatten()*, *first()*, or *last()*, are omitted as they don't affect the computation of the transformation footprints.

The set of footprints resulting after using our algorithm in the *Class2Relational* transformation is summarized in Table 4.2. This set is organized by rules. We represent footprints as a series of *NavigationCallExpression* starting from a particular Type. They take the form of $[source.type][.'('?[propertyName][')'?*]?]^+$. For instance, the binding '*tables*' in rule *Package2Schema* contains a multi-level navigation '*{Package.classes.attr}*'. A chunk of the footprint can take part in a recursive call. We use '$*$' to denote it. We argue that multi-step navigation calls provide a better approximation of the dependency graph. The same argument holds for recursive navigation calls.

4.4.2 A greedy model-partitioning algorithm for distributed transformations

Although the footprints of the transformation help us approximate a transformation dependency graph, data distribution necessitates traversing the model according to these footprints. This traversal process can be expensive as it may possibly visit model elements more than once. In our approach, we propose a greedy partitioning algorithm which visits every model element only once. For each element, we only visit the first degree neighbours as stated by the footprints with initiating type. Thanks to this algorithm, it is possible to instantly decide to which machine a model element should be assigned.

We first decompose all existing multi-level navigation footprints into single-navigation footprints in order to visit only first degree neighbours. Then, we order them according to the navigation call chain in the original footprint. Single-level footprints have by default an order equals to 1. For example , the footprint {Package.classes.attr} decomposes to two single-level footprints with the corresponding order, FP_1 = {{Package.classes,1}, {Class.attr,2}}. Later, we show how the order of footprints is used to enable visiting only first degree neighbours.

The initiating type of the resulting footprints is inferred from the navigation call referring type. The order of the decomposed footprints is important, it enables processing the right single footprint, and discard the succeeding ones. Moreover, it allows keeping track of the next footprint to be taken into consideration during the partitioning process. Later, we show how, by using these footprints, we monotonically build our transformation dependency graph.

It is theoretically impossible to have a good streaming graph partitioning algorithm regardless of the streaming order. The ordering on which model elements arrive, may sometimes cause it to perform poorly. Moreover, sometimes, it can be discouraged to decide to which partition a model element should be assigned. Hence, the use of a buffer to store these model elements for a future processing is favoured. The buffer should have a capacity that, once is reached (hopefully, after the master has enough information to decide to which machine these elements should be assigned), it gets cleared, and belonging elements should be processed.

To alleviate this process, we distinguish between high-priority and low-priority elements [120]. High-priority elements are elements that can participate in building the

Algorithm 5: Data distribution algorithm

Input : Stream<Vertex> $stream$, int $avgSize$, int $bufCap$, int m

```
1  assignElementToMachine (stream.first(), 1)
2  while stream.hasNext() do
3      element := stream.next()
4      if isHighPriorityElement(element) then
5          machineId := ComputeMachineId(element)
6          storeElement-MachineDeps(element, clusterId)
7          assignElementToMachine(element, clusterId)
8      else
9          addToBuffer(element)
10     if |buffer| == bufCap then
11         processBuffer()
12 processBuffer()

13 Function ComputeMachineId (element)
       Output : int index
14     index := max_{i∈m} ( |dependentElementsInMachine(element, i)| *
       (1 − currentSize(i) / avgSize) )

15 Function storeElement-MachineDeps (element, clusterId)
16     foreach fp ∈ getFirstOrderFPsForType(element.type) do
17         dependent_elements := getPropertyElements(element, fp.property) →
           reject(alreadyAssigned)
18         addDependencies(clusterId, dependent_elements)
19     foreach fp ∈ getFPsFromDeps(element, clusterId) do
20         dependent_elements := getPropertyElements(element, fp.property) →
           reject(alreadyAssigned)
21         addDependencies(clusterId, dependent_elements)

22 Function processBuffer ()
23     while buffer.hasNext() do
24         element := buffer.next()
25         clusterId := ComputeMachineId(element)
26         storeElement-MachineDeps(element, clusterId)
27         assignElementToMachine(element, clusterId)
```

transformation dependency graph. These elements are immediately processed and assigned to a split. Whilst, low-priority elements do not directly participate in building the transformation dependency graph. For example, the elements of a type that does not figure in the LHS of any footprints may be considered low-priority element. Low-priority elements are stored in a buffer for later processing.

Algorithm 5 describes our greedy data distribution approach. It takes as input the graph stream, the average machine size '*avgSize*', the buffer capacity '*bufCap*', and finally the number of commodity machines '*m*'. The algorithm starts by assigning the first element to the first machine. Next, for each streamed element, if the element is high-priority (need to be assigned immediately), then it computes the index of the appropriate machine to be assigned to (line 5). Later, it stores its dependencies (line 6). Finally, the element is assigned to the given machine (line 7). If the element is low-priority, it is directly stored in the buffer (line 9).

In order to monotonically build our transformation dependency graph, the dependencies of each element are stored after its assignment. A dependency has the following structure, {element_id, next_footprint, split_id}. This reads like, the element with 'ID' *element_id*, along with the elements referred to by *next_footprint*, are needed in the split with 'ID' *split_id*. The dependencies storage process happens in two phases (line 15). It starts by recovering all footprints with initiating type as the element type and order equals to 1 (line 16). For each footprint, we check if it has any successor. If so, the dependency is stored together with the next footprint to be resolved if the dependent element is assigned to the same machine. In case the footprint is originally single, we only store the dependency of the element to the machine. Dependencies referring to elements that have been already assigned are not created (line 16).

For example, after assigning the element 'p1', supposedly to split m_1, we collect its corresponding footprints, which are, {*Package.classes, Package.types*}. For the footprint '*Package.classes*', we generate the following dependencies: {*c1, Class.assoc,* m_1} and {*c2, Class.assoc,* m_1}. The elements *c1* and *c2* are resulted from visiting '*classes*' property. As for the footprint *Package.types*, only one dependency is generated, which is, {*c1, null,* m_1}. The value of *next_footprint* is null, as *Package.types* is originally single-level.

In the second step, we collect all the dependencies corresponding to the current element and the assigned split, with a non-null value of *next_footprint* (line 19). If any, additional dependencies are created, similarly to the previous step. This step enables building the dependecy graph while visiting only first degree neighbours. For example, if the element 'c1' was assigned to m_1, then, we recover the dependency {*c1, Class.attr,* m_1}, and we generate the additional dependencies. After the dependencies creation, existing dependencies referring to the current element are deleted.

The function *ComputeMachineId* (line 13) is responsible for deciding the appropriate machine. It assigns a model element to the machine with the highest number of elements depending on it, and adds it to the list of already visited (assigned) elements, thanks to the dependencies generated at every model assignment. The amount of dependencies to an element e in a particular machine m refers to, the number of dependencies with *element_id* equals to e , and *split_id* equals to m. This value is weighted by a penalty function based on the average size a machine can take, and penalizing larger partitions. If the splits have similar score, then the element is assigned to the split with the lowest index. While different penalty functions can be used, we go for a linear weighted function, as it

has experimentally shown good performance results [120]. This forces our distribution to be uniform. Experimenting our algorithm with different penalty functions is left for future work.

4.5 Conclusion

In this chapter, we argued that distributed model transformations in relational languages can be improved by using efficient distribution strategies. To cope with this, we first showed that our problem can be related to the problem of uniform graph partitioning. Later, we adapted existing formalization to our system. Then, we proposed a greedy data distribution algorithm for declarative model transformations, based on static analysis of transformation rules. We introduced an algorithm to extract the knowledge on how the input model is accessed by the transformation application. Finally, we use this knowledge to help out the algorithm deciding the appropriate assignment.

With respect to existing approaches, we defined tasks at a very fine-grained level, allowing us to have more precise reasoning on data distribution for distributed model transformations. Moreover, while existing data-driven approaches analyze graph-data itself to ensure a better data locality, in our approach, we rely on an approximate dependency graph. This graph is computed thanks to the transformation footprints, which is used to help our data distribution algorithm decide to which split a model element should be assigned.

Chapter 6 validates our approach by introducing an open-source implementation of our solution. We apply it to our distributed MT engine, ATL-MR [17] (see Chapter 3) coupled with a distributed persistence backend, NEOEMF/COLUMN. We build our footprints extraction algorithm on top of anATLyzer [39], a static analysis tool for model transformation in ATL. Finally, we implement our heuristic as a custom split algorithm for Hadoop/HBase applications. Chapter 7 reports on our evaluation results and shows that thanks to our greedy algorithm, we improved the access to our distributed backend by up to 16%. In the next chapter, we introduce our solution for storing and manipulating VLMs on top of NoSQL databases.

5

Scalable model persistence

With large-scale software engineering becoming a compelling necessity in several industrial contexts, companies need tools that are capable of scaling efficiently. Some of these companies embracing the main concepts of MDE are witnessing an increasingly growing size and complexity of their model-data. For instance, the modernization of very complex legacy systems results in models with several millions of elements. These models need to be stored, manipulated, even sometimes transformed to modernize a legacy application. Model manipulations part of such tools are mainly based on reading from and/or writing to a model.

In practice, modeling tools are built around modeling frameworks, which provide basic model-management functionalities and interoperability to the modeling ecosystem. The EMF framework is the most popular one. This is testified by the amount (counting more than two hundred) of tools coming from both industry and academia. Such frameworks offer a common base for implementing tools for different application domains, e.g. reverse engineering [27], model transformation [106, 75], and code generation [24, 103]. EMF adopts XMI (XML Metadata Interchange) as its default serialization format. This file-based representation faces several issues related to scalability when handling VLMs. Systematically, tools built around EMF show similar limitations as well.

While several solutions to access and store EMF-based models exist, they are still limited for the several reasons – as we will see in the next section. As a result of these limitations, NeoEMF was proposed, a multi-persistence backend for manipulating and storing EMF-based models in NoSQL databases. Currently, NeoEMF is implemented in three different backends, each, is well-suitable for specific modeling scenario. NeoEMF is highly extensible and advanced EMF developers can easily adapt it to specific scenarios. NeoEMF is open source and available online [1].

This chapter is structured as follows. In Section 5.1 we investigate the limitations of state of the art tools. Later in Section 5.2, we present a global overview of our solution,

1. http://www.neoemf.com/

and we describe its common features. Section 5.3 describes the model-data mapping to the underlying backends. Finally, Section 5.4 details our implementation, while Section 5.5 concludes the chapter.

5.1 Related Work

The interest on scalable model persistence has grown significantly in recent years. Nonetheless, existing approaches are still not suitable to manage this kind of artefacts both in terms of processing and performance.

XMI-based approaches. Models stored in XMI need to be fully loaded in memory for persistence. The lack of support for lazy or partial loading of models hampers handling VLMs not fitting in a memory of a single machine. Moreover, this persistence format is not adapted to developing distributed MDE-based tools. Hereafter, we investigate the state-of-the-art tools and frameworks for persisting EMF-based models and draw down their limitations.

One way to tackle scalability issues while sticking to the XMI representation, is by decomposing the model into fragments, and thus, alleviating its complexity. Amálio et al. [4] proposed a mathematical ground that tackles the scalability issues of models in MDE by breaking them down into smaller chunks. This theory is based on the ideas of modularity and separation of concerns.

EMF fragments [93] is a hybrid persistence layer for EMF models aimed at achieving fast storage of new data and fast navigation of persisted one. EMF-Fragments uses annotations to decompose a model into smaller documents. A similar approach is EMF Splitter [52], it borrows the modularity structure used in Eclipse for Java projects organization to decompose the models. Both approaches rely on the proxy resolution mechanism used by EMF for inter-document relationships. EMF Fragment support MongoDB, Apache HBase and regular files, while EMF Splitter supports only XMI files. The backends assume to split the model into balanced chunks. This may not be suitable to distributed processing, where the optimization of computation distribution may require uneven data distribution. Moreover, model's distribution is not transparent, so queries and transformations need to explicitly take into account that they are running on a part of the model and not the whole model.

Relational-based approaches. CDO is the *de facto* standard solution to handle large models in EMF by storing them in a relational database. It was initially envisioned as a framework to manage large models in a collaborative environment with a low memory footprint. However, different experiences have shown that CDO does not scale well to very large models [119, 111, 112]. CDO implements a client-server architecture with transactional and notification facilities where model elements are loaded on demand. CDO servers (usually called repositories) are built on top of different data storage solutions. However, in practice, only relational databases are commonly used. Furthermore, the client-server architecture constitutes a bottleneck and does not exploit a possible alignment between data distribution and computation distribution.

NoSQL-based approaches. Barmpis and Kolovos [14] suggest that NoSQL databases would provide better scalability and performance than relational databases due to the interconnected nature of models. Morsa [111] was one of the first approaches to provide persistence of large scale EMF models using NoSQL databases. Morsa is based on a NoSQL database. Specifically, Morsa uses MongoDB [100], a document-oriented database, as its persistence backend. Morsa can be used seamlessly to persist large models using the standard EMF mechanisms. As CDO, it was built using a client-server architecture. Morsa provides on-demand loading capabilities together with incremental updates to maintain a low workload. The performance of the storage backend and their own query language (MorsaQL) has been reported in [111] and [112].

Mongo EMF [28] is another alternative to storing EMF models in MongoDB. Mongo EMF provides the same standard API as previous approaches. However, according to the documentation, the storage mechanism behaves slightly different than the standard persistence backend (for example, when persisting collections of objects or saving bidirectional cross-document containment references). For this reason, Mongo EMF cannot be used without performing any modification to replace another backend in an existing system.

Summary. In this section, we presented state-of-the-art tools for the persistence of VLMs. These approaches vary in techniques used as well as persistence backends. We observed that, while most of existing approaches scale in some scenarios, they however expose one or more of the following limitations:

1. Existing solutions based on XMI still require a good fragmentation from the user. Moreover, the lazy-loading is supported at only the chunck level. As a consequence, complex queries may lead to loading the whole model even though some elements are not accessed (e.g. *EMF Fragments* and *EMF Splitter*).

2. Existing solutions are not well-suitable to specific modeling scenarios. Some works [14, 59] have shown that graph database and relational databases are not well-suitable to repetitive atomic operations performed on models, while graph databases perform better in the case of complex queries.

3. Existing solutions are not transparent with regard to model distribution. Queries and transformations need to explicitly take into account that they are running on a part of the model and not the whole model (e.g. *EMF fragments* and *EMF Splitter*).

4. Existing solutions are not suitable for distributed processing. The backend assumes to split the model in balanced chunks (e.g. *EMF Fragments*). This may not be suitable to distributed processing, where the optimization of computation distribution may require uneven data distribution.

5. Existing solutions using a client-server architecture use a single access point. Even when model elements are stored in different nodes, access to model elements is centralized, since elements are requested from and provided by a central server (e.g. CDO over a distributed database). This constitutes a bottleneck and does not exploit a possible alignment between data distribution and computation distribution. With a growing size of clients, single access point can rapidly turn into a limitation.

Type	Requirements
Interoperability	- Compliance with the modeling ecosystem - Extensible caching facilities - Replaceable backend engine
Performance	- Memory-friendly - Fast response

Table 5.1 – Requirements of persistence backends in modeling ecosystem

5.2 NEOEMF: Global Overview

Gómez et al. identified five requirements for designing a solution to persisting VLMs. These requirements care to improve model persistence solutions for a scalable execution time and memory occupancy. But more importantly, to ensure that the persistence solution is compatible with existing tools in the modeling ecosystem, for different application domains and different scenarios. Requirements are grouped in two categories `interoperability` and `performance` requirements as depicted in Table 5.1. Next we introduce NEOEMF, our solution for scalable persistence of model-data in NoSQL databases. Then, we show how it responds to these requirements thanks to its characteristics and features.

Figure 5.1 shows a high-level overview of the proposed architecture, implemented on top of the EMF framework. EMF-based tools would not be aware of the NEOEMF framework, as communications between the tool and NEOEMF are always passing through the EMF layer (`compliance`). The persistence manager communicates in its turn with the underlying backend by means of a specific driver for serializing and deserializing data. Each driver is responsible for mapping/translating EMF API operations to backend API calls of one of the supported persistence backends. In particular, NEOEMF supports three different NoSQL persistence backends, the NEOEMF/GRAPH [15], NEOEMF/-COLUMN [58], and NEOEMF/MAP [59] (`Replaceable backend engine`). In this chapter, we only cover NEOEMF/GRAPH and NEOEMF/COLUMN in details.

NEOEMF fulfills the performance requirements through the support for `lazy loading` to fetch only needed element and `save if needed` mechanisms to reflect only new updates to the persistence backend. While the first mechanism reduces memory footprint and allows programs to load and query large models in systems with limited memory, the second mechanism avoids making a full traversal in order to store small model changes. This is handled through two techniques. The first technique is `direct-write`. It propagates the model changes directly to the underlying backend after each EMF API call involving a write operation. The second technique, `auto-commit`, caches the write operations, and persists them (in one shot) when the buffer size limit is reached. This technique relies on transactions (if supported by the underlying backend) to commit the change operations. The maximum number of operations per transaction [2] is costumizable. The changes can also be persisted if the save operation is invoked by the application.

2. Buffer size limit

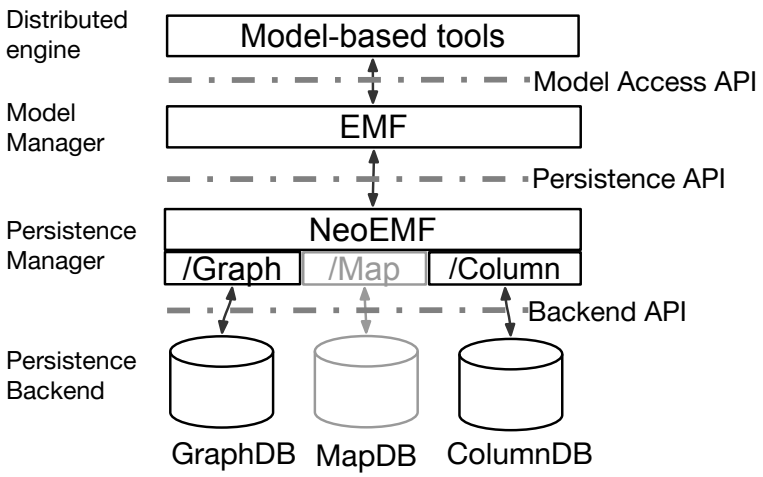


Figure 5.1 – NEOEMF: a general view and architecture

NEOEMF is shipped with a set of caching strategies plugged to the drivers. Each cache offers a specific optimization to improve one of performance requirements depicted in Table 5.1. **Features-cache** is responsible for storing loaded objects by their accessed feature. The **isSet-cache** keeps the result of `isSet` calls to avoid multiple accesses on the database. **The size-cache** keeps the result of `size` calls on multi-valued features to avoid multiple accesses on the database. Finally, the **record-cache** maintains a list of backend records each representing a model element to limit costly serializations/deserializations.

As modeling scenarios may variate from one application domain to another (e.g. querying pattern, multiple-client access, etc.), it is required to implement solutions that can be adapted to each one of these scenarios. One way to answer to this issue, is relying on the characteristics and capabilities of different existing NoSQL stores. NEOEMF offers three different implementations, each, targeting a specific scenario.

NEOEMF/GRAPH uses GraphDB to persist model-data in its natural form. The data model mapping is straight forward, except for model elements type. It is represented through a relationship of type *INSTANCE_OF* towards a vertex representing the type. Graph databases have shown great performance results for manipulating connected data, which makes it more suitable for performing complex graph traversals.

NEOEMF/COLUMN is designed to enable the development of distributed MDE-based applications by relying on a distributed column-based datastore. In contrast to the two other implementations, NEOEMF/COLUMN offers concurrent read/write capabilities and guarantees ACID properties at the model element level. It exploits the wide availability of distributed clusters in order to distribute intensive read/write workloads across a cluster's data-nodes.

NEOEMF/MAP relies on a map-based data model to store model elements. Based on the hashtable data structure, model elements and their properties are represented as a collection of $\langle key, value \rangle$. The main advantage of this representation lies in its powerful capacity to access records in a constant time. Thanks to this feature, NEOEMF/MAP is designed for performing model queries as a set of atomic operations such as accessing a single element/attribute, and navigating a single reference.

5.3 Model-data-mapping

Data in MDE is represented in its essential form by *Directed Typed Attributed Graphs*. Different ways have been proposed to represent these graphs for different intentions, both in-memory or on-disk. Each of these representations was designed for a particular purpose, depending on the kind of computation the graph data is involved in and the available resources. For instance, one would leverage data memory/disk occupancy over the execution time of a particular operation, or vice-versa.

In this section we focus on the on-disk representation of these graphs [3], and the mapping to the data model of the underlying persistence technology. In particular, we introduce our mapping proposal to Graph stores and Column stores. A data model should be able of mapping every model element and its properties, except for derived ones, they can be inferred at runtime. Other information that can be inferred at runtime is the type of attributes. Instead, information about the concrete type of model elements should be stored. Finally, the mapping should differentiate between properties and attributes by means of their qualified names.

In order to meet the performance requirements presented in Table 5.1, we design the underlying data model to minimize the response time of each method of the MOF Reflection API when possible (see Section 3.4). Hence, we identify the following design principles as key features for improved performance:

— `Indexing of root elements`: root elements help recontructing the tree-like representation of models. In particular, root elements are used in model traversal (tree iterator). In order to speed up these operations, our mapping takes into consideration the indexing of root elements.

— `Using Universal Unique IDentifiers`: for databases with indexing or fast look-up capabilitites it would be recommended to index model elements. We use `UUIDs` to track these objects and be able to fetch them from the underlying storage immediately.

— `Conformance relationship`: graphs in MDE are typed, and being able to know element's type is a requirement. We design our mapping to store metaclass information as well.

— `Representing the containment references`: another relationship that might be represented to improve the response time of some methods of the *Reflection API* is the containment one. Although this information is already represented by simple references, representing it separately speeds up the response time.

Hereafter, we show our data model mapping to Column-stores and Graph-stores. In order to exemplify our data model, we refer to the DataFlow metamodel presented in Figure 2.2a along with its model instance, as depicted in Figure 2.2b.

The de facto implementation of the MOF standard is EMF. It has its own metamodel called `ECore`. This metamodel is slightly different than the MOF's one (see Section 2.1), and reduces the number of model elements. Mainly, EMF simplifies its metamodel by referring to Class properties as `Attributes` and Association properties as `References`.

3. we suppose that the in-memory representation of model instances in MOF should be handled by the modeling framework.

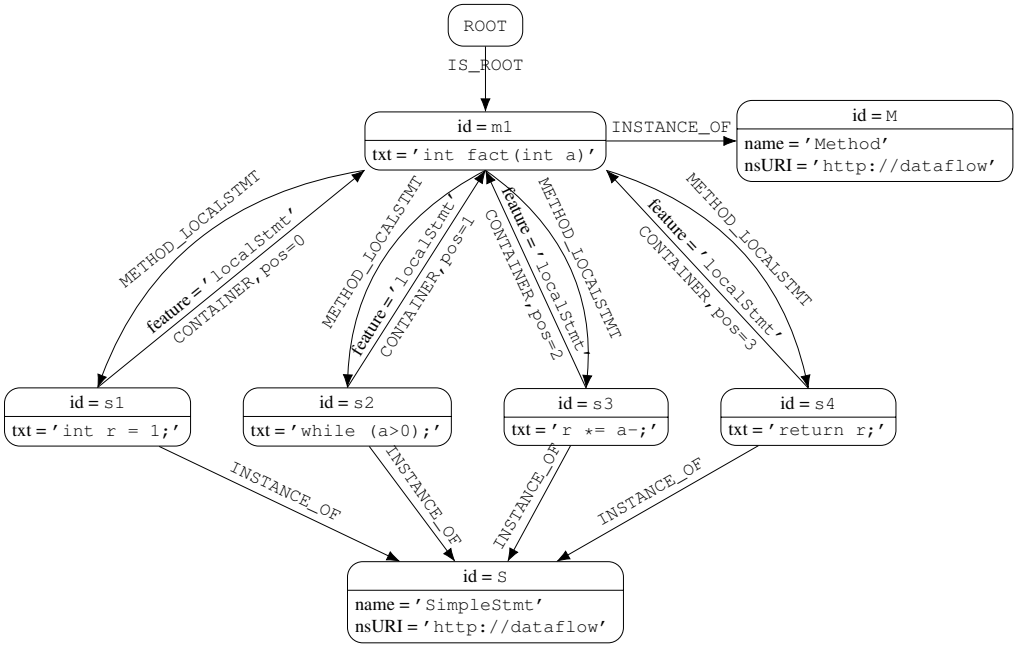


Figure 5.2 – Representation of the sample instance in Figure 2.2b model in Graph-store. For readability reasons, only the `localStmts` reference is represented.

References in `Ecore` are owned by the first member of the association and has for type the second member. The metaclass `Association` does not exist in `Ecore`. Henceforth, we use the EMF nomenclature.

5.3.1 Graph-based Store Mapping: NEOEMF/GRAPH

Models in MDE can naturally be mapped to concepts in Graph-stores. This natural mapping is the main motivation that leads us to choose a native graph database. Another motivation is the promising performance graph stores have shown for connected data operations. However, graph stores are not natively typed, hence, information about metaclasses should be explicitly represented.

The metaclass information in NEOEMF/GRAPH is represented using NODES. Metaclass nodes have two attributes, `name` and `nsURI`. Model elements refer to their corresponding metaclass using an outgoing relationship of type `INSTANCE_OF`. The node with ID `m1` refers to the node `M` as its metaclass as shown in Figure 5.2. Likewise, `s1`, `s2`, `s3` and `s4` are instances of the metaclass `SimpleStmt` .

MODEL ELEMENTS are also represented as NODES. `Nodes` (in Figure 5.2) `m1`, `s1`, `s2`, `s3` and `s4` are examples of this, and correspond to the elements `m`, `s1`, `s2`, `s3` and `s4` shown in Figure 2.2b. *Element attributes* are represented as *node properties* contained in the corresponding `node`. Most of existing graph-stores have native support for primitive types. The value of the attribute `txt` of the element `m` is equal to `'int fact(int a)'`.

ROOT ELEMENTS are referred to by the relationship `IS_ROOT`, outgoing from the node `ROOT`. Thanks to the built-in node indexer that the `ROOT` node is pointed at, and

simply traversed to return the set of root elements [4]. The node `m1` is a root element as it has an incoming relationship of type `IS_ROOT` as depicted in Figure 5.2.

REFERENCES are represented as RELATIONSHIPS. To avoid naming conflicts in RELATIONSHIPS, we use the following convention for assigning names: `CLASS_NAME_ _REFERENCE_NAME`. Arrows labelled `METHOD_LOCALSTMT` in Figure 5.2 are examples of references. Bidirectional references would be represented with two separate directed graph `relationships`. In the case of multi-valued references, we use labels to guarantee that model elements are returned in the same order as expected. Finally, containment information is also represented as a relationship between the container and the contained nodes. The relationship is called `CONTAINER`. It has a property `feature`, which refers to the containing feature. The elements `s1`, `s2`, `s3`, and `s4` are contained in the element `m1` as illustrated in Figure 5.2. The name of the containing feature is `localStmt`. Note that for readability and simplicity reasons, the representation of the `dfNext` reference is omitted.

5.3.2 Column-based store mapping: NEOEMF/COLUMN

One of the common ways of representing graphs is by using adjacency lists. For each node in the graph, a list of vertices adjacent to it is stored. This representation has a low storage cost on disk, and the time to access the list of adjacent nodes is constant. According to existing work [59], such a representation has shown promising results for applications with massive reads. Moreover Google's original use case for BigTable was the storage and processing of web graph information represented as sparse tables. Our design takes advantage of the use of UUIDs design principle to flatten the graph structure into a set of key-value mappings.

NEOEMF/COLUMN uses a single table with three column families to store the information of the models: (i) a `property column family`, that keeps all objects' data stored together; (ii) a `type column family`, that tracks how objects interact with the meta-level (such as the `INSTANCE_OF` relationships); and (iii) a `containment column family`, that defines the model's structure in terms of containment references. Table 5.2 [5] shows how the sample instance in Figure 2.2b is represented using this datamodel. Row keys are the object `unique identifier`. The `property column family` stores the object's actual data. This column family contains all the existing structural features in the metamodel (attributes and properties). As it can be seen, not all rows are populated. Thanks to sparsity property in column-stores that the storage cost is very cheap, only populated properties are stored. Features with the same name but residing in different namespaces will still be stored in the same column. This solution causes no conflict since each stored element is uniquely identified regardless of the feature type. All data is stored as byte literals and NEOEMF/COLUMN is responsible for parsing the resulting value to its appropriate type.

The way how data is stored depends on the `property type` and `cardinality` (i.e., upper bound). For example, values for single-valued attributes (like the `txt`, which stored in the `txt` column) are directly saved as a single literal value; while values for many-valued attributes are saved as an array of single literal values (Figure 2.2b does not

4. The node indexer is also used to fetch model elements with a cheap cost
5. Actual rows have been split for improved readability

Table 5.2 – Example instance stored as a sparse table in Column-store

	PROPERTY			
KEY	ECONTENTS	TXT	DFNEXT	LOCALSTMTS
'ROOT'	'm1'			
'm1'		'int fact(int a)'	{'s2','s3'}	{'s1','s2','s3','s4'}
's1'		'int r = 1;'	{'s3','s4'}	
's2'		'while (a>0);'		
's3'		'r *= a--;'	{'s3'}	
's4'		'return r;'		

	CONTAINMENT		TYPE	
KEY	CONTAINER	FEATURE	NSURI	ECLASS
'ROOT'			'http://dataflow'	'RootEObject'
'm1'	'ROOT'	'eContents'	'http://dataflow'	'Method'
's1'	'm1'	'localStmts'	'http://dataflow'	'SimpleStmt'
's2'	'm1'	'localStmts'	'http://dataflow'	'SimpleStmt'
's3'	'm1'	'localStmts'	'http://dataflow'	'SimpleStmt'
's4'	'm1'	'localStmts'	'http://dataflow'	'SimpleStmt'

contain an example of this). Values for single-valued references are stored as a single literal 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 `localStmts` containment reference. The element `m1` stores {'s1','s2','s3','s4'} in the cell identified by ⟨`c1`, `localStmts`⟩.

The `containment column family` maintains a record of the container of every persisted object. The `container` column records the identifier of the container object, while the `feature` column records the name of the `property` that relates the container object with the child object (i.e., the object to which the row corresponds). Table 5.2 shows that, for example, the container of the `m1` is `ROOT` through the `eContents` property (i.e., it is a root object and is not contained by any other object). The next depicts the `SimpleStmt s1` is contained in the `m1` through the `localStmt` PROPERTY.

The `type column family` groups the type information by means of the `nsURI` and `EClass` columns. For example, the table specifies the element `m1` as an instance of method `Method` class of the `Dataflow` metamodel (that is identified by the `nsURI` `http://dataflow`).

5.4 Implementation

NEOEMF is implemented as a set of open source Eclipse plugins on top of the EMF Framework [6]. In its common API, NEOEMF receives CRUD operations from the EMF API, and delegates them to the appropriate persistence manager. This component is then in charge of the serialization/deserialization of model elements. NEOEMF is easily parameterizable, and users are free to use any of the available stores, according to their modeling scenario. An underlying backend can have more than one store, each with a particular behaviour. As follows, we show some examples of the available stores.

6. www.neoemf.com

Another feature that assists the scalability in NEOEMF is the *lazy-loading* mechanism. Other model persistence solutions such as MongoEMF [28] and CDO [1] support this mechanism as well. Lazy-loading works by loading objects in memory only when they are accessed, in contrast to XMI's approach that parses the whole model in memory in one shot. Lazy-loading is implemented as a common feature in NEOEMF. It consists of a wrapper delegating all its method calls to an embedded *EStore*, which directly manipulates elements at the store level. Caching Strategies come into play to reinforce the lazy-loading mechanism by maintaining only a few number of elements in memory (the ones that have not been saved), and thus, reducing the memory consumption of modeling applications.

NEOEMF's compliance with the EMF API comes at a low price, and EMF's features such as *code generation*, and the use of *dynamic & reflexive* APIs are still supported. Besides that, NEOEMF is delivered with a model *importer*, which enables an optimized model serialization from XMI to a dedicated database, alongside with a model *exporter* for the opposite operation. These two features are designed to enhance NEOEMF's integration in existing environment, while maintaining a low memory footprint.

NEOEMF has been released as part of the MONDO [82] European project and has proved an effortless and transparent integration with existing EMF-based tools coming from the industrial partners. This transparent integration is provided thanks to the specialization of the EMF classes. Finally, NEOEMF is well documented and experienced EMF users can safely extend it with their own caching strategies, stores, or even underlying backends.

NEOEMF/GRAPH relies on Blueprints [127], an API designed to access and manipulate property graph data models. It was intended to unify graph databases under a common API. Indeed, it has been implemented by the world's leading graph databases, such as Neo4j and TitanDB[10]. In this perspective, users can integrate NEOEMF/GRAPH with any graph databases of their choice, as long as it implements the Blueprints API. In particular, NEOEMF/GRAPH is provided with a convenience wrapper for Neo4j. NEOEMF/GRAPH is shipped with two stores, `Auto-commit` and `Dirty-saving` [83].

The `Auto-commit` store is suitable for backends having restrictions on size (e.g. size of transactions). It has a limit of write operations that can be stored in memory. Once this limit is reached the store commits all the changes to the underlying backend. Instead, the behaviour of `Dirty-saving` [83] is slightly different. Once the limit is reached, it saves model elements to the underlying storage, but tags them as dirty, and finally unloads them. If later on, the `save` method is triggered, then the tags are removed and data is cleaned. Otherwise, if the resource shuts down without triggering the `save` method, then data is simply removed from the database. Both of these stores are designed to enable the efficient manipulation of VLMs in limited resources.

Graph databases are not well-suitable to compute atomic accesses of single reference or attribute, which are typical queries computed by graphical model editors. Mogwaï [41], a querying framework for NEOEMF/GRAPH, exploits the capabilities of advanced query and traversal provided by graph databases to improve the response time and memory footprints of complex queries. The framework maps OCL expression to graph navigation traversals, which returns the set of requested elements.

NEOEMF/COLUMN is built on top of Apache HBase [124]. NEOEMF/COLUMN hides the model distribution from client's applications. Model access to remote model elements in NEOEMF/COLUMN is decentralized, which avoids the bottleneck of a single access point, and alleviates the alignment between data distribution and computation distribution.

NEOEMF/COLUMN is delivered with two stores, the first one is `Direct-write` and the second one is `Read-only`. The `Direct-write` store uses a different strategy for optimizing memory usage by reflecting model changes directly to the underlying backend. And thus, make the new changes available to other clients. Inconveniently, all clients have to fetch properties values from the underlying backend due to recurrent model edits by other clients. In future work, we plan to supply NEOEMF/COLUMN with a distributed notification mechanism to alert clients of changes in model elements.

NEOEMF/COLUMN is well-suitable to distributed modeling applications with massive reads, typically, in model-to-model transformations. We provide `Read-only` store to avoid multiple fetching of data values in read-only scenarios. In contrast to the `Direct-write` store, this one relies on a caching mechanism. NEOEMF/COLUMN is integrated with *ATL-MR* [16]. It exploits the lazy-loading mechanism to allow each client to load only the part of the model needed. It also enables the cluster's nodes to share read/write rights on the same set of input/output models.

Integrating NEOEMF/COLUMN in modeling applications takes more efforts than NEOEMF/GRAPH and NEOEMF/MAP. Users need to adapt their applications to the distributed programming logic of NEOEMF/COLUMN. In particular, users need to consider that ACID properties are only supported at the object level, which relies completely on HBase. A thorough description of ACID properties support is provided in Section 6.1.

NEOEMF/MAP is built around the key-value store MapDB, which provides *Maps*, *Sets*, *Lists*, *Queues* and other collections backed by off-heap or on-disk storage. MapDB describes itself as hybrid between `Java Collections` and an `embedded database engine` [92]. It provides advanced features such as ACID transactions, snapshots, and incremental backups. NEOEMF/MAP relies on the set of `Maps` provided by MapDB and uses them as a key-value store. NEOEMF/MAP comes also with two stores, `Auto-commit` and `Direct-write`.

5.5 Conclusion

In this chapter we proposed NEOEMF, a persistence solution for VLMs in EMF. NEOEMF overcomes the limitations of existing EMF persistence frameworks by providing different backends, each one suitable for a particular modeling scenario. Especially, NEOEMF/MAP was designed for modeling applications that undergo repetitive, yet atomic operations performed on models. For scenarios with more complex and elaborate query patterns, NEOEMF/GRAPH comes to the rescue. It relies on advanced traversal capabilities of graph databases to query models efficiently while maintaining a low memory-footprint. In contrast to NEOEMF/MAP and NEOEMF/GRAPH, the NEOEMF/COLUMN backend offers concurrent read/write capabilities and guarantees ACID properties at the model

element level. It also offers a transparent and flexible model distribution by virtue of a decentralized architecture.

Moreover, we showed how NEOEMF can achieve good scalability results in terms of memory occupancy and response time by relying on its advanced features such as, lazy-loading, pluggable caching, and other features. NEOEMF can be transparently integrated with any EMF-based legacy application at a cheap price, except for the NEOEMF/COLUMN implementation. It needs the application's business logic to be aligned with its distributed programming logic. Indeed, NEOEMF/COLUMN is integrated with ATL-MR to enable concurrent read on the input models and concurrent writes to the output ones. Chapter 6 details the integration specificities.

6

ATL-MR: distributed MTs with ATL on MapReduce

In the previous chapters, we introduced our approach for distributing model transformations on MapReduce. We also presented our solution to efficiently partition input models based on static analysis of model transformations and by applying a state-of-the-art algorithm for the partitioning of graph streams. In this chapter, we validate our solution by building a prototype MT engine for the distribution of ATL transformations on MapReduce. Our engine is implemented on top of Hadoop, the Apache's open-source implementation of MapReduce. We provide our solution with a distributed persistence backend based on our model persistence framework NEOEMF/COLUMN. Section 6.1 gives an overview of our distributed engine. It details our implementation, then introduces our data distribution and access modalities. Finally, it depicts how the framework reacts to failures. Section 6.2 describes how we guarantee the consistency of persisted models in distributed model transformations while relying on ACID properties as provided by HBase, an open-source implementaiton of BigTable.

6.1 ATL on MapReduce (ATL-MR): a global overview

6.1.1 Global overview

Figure 6.1 shows the high-level architecture of our distributed engine on top of the EMF. ATL-MR runs on an Apache Hadoop [7] cluster. Hadoop is the leading open-source implementation of MapReduce. The *Master* node is responsible for distributing the input model, monitoring the ATL-MR *Slaves*, and finally returning the output results. Each node in the Hadoop cluster transparently communicates with the underlying backend through an

EMF model management interface. And hence, making our distributed framework unaware of the persistence backend type. Indeed, besides supporting the de-facto serialization format, XMI [107], ATL-MR is also coupled with a multi-database model persistence framework, NEOEMF [40]. ATL-MR relies on the HDFS to distribute all of its input and output models, together with the transformation specification.

HDFS is the primary distributed storage used by Hadoop applications as it is designed to optimize distributed processing of multi-structured data. It is well suited to distributed storage and distributed processing using commodity hardware. It is fault tolerant, scalable, and extremely simple to exceed. In our framework, XMI files are duplicated and stored in different data nodes for a better data locality and high availability. However, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency, which makes it neither suitable nor optimized for atomic model operations. Moreover, HDFS resources cannot be written concurrently by multiple writers without locking, which results in locking delays. Also writes are always made at the end of the file. Thus, writing in the middle of a file (e.g. changing a value of a feature) involves rewriting the whole file, leading to more significant delays.

ATL-MR exploits the capabilities of NEOEMF/COLUMN, implemented on top of HBase to distribute the storage of EMF models and enable concurrent R/W. HBase is a better choice for low-latency access. HBase allows fast random reads and writes. HBase is row-level atomic, i.e. inter-row operations are not atomic, which might lead to a dirty read depending on the data model used. Additionally, HBase only provides five basic data operations (namely, `Get`, `Put`, `Delete`, `Scan`, and `Increment`), meaning that complex operations are delegated to the client application, which in its turn, must implement them as a combination of these simple operations.

Each node in the system executes its own instance of the ATL VM but performs either only the local match-apply or the global resolve phase. Configuration information is sent together with the tasks to the different workers, so that they can be able to run

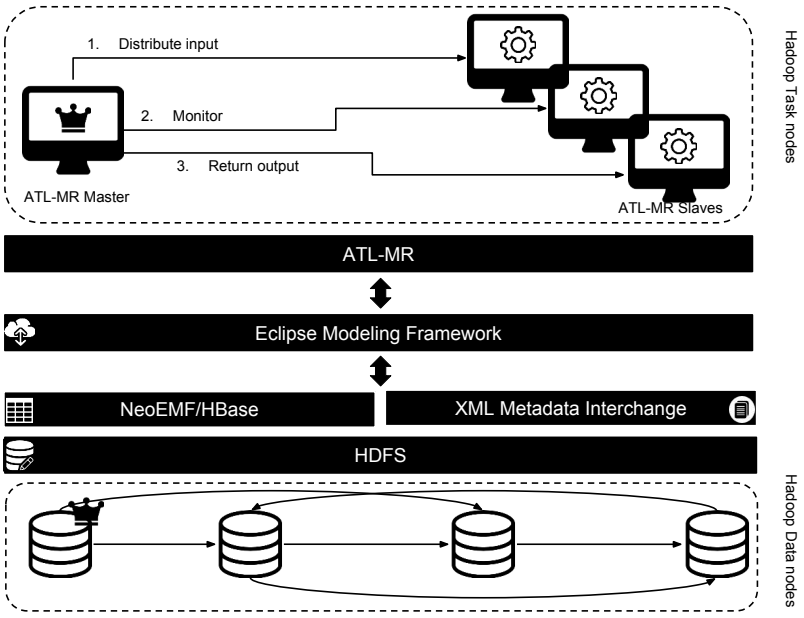


Figure 6.1 – ATL-LR: a general view and architecture

Table 6.1 – API extension

CLASS NAME	OPERATION	DESCRIPTION
ExecEnvironment	matchSingle(EObject)	Matching a single object
	localApplySingle(TraceLink)	Applying if possible
	globalResolve()	Resolving unresolved bindings and assignements in the global scope
TraceLinkSet	mergeTrace(TraceLink)	Add traceLink if does not exist and resolve input and output cross references

their local VMs independently of each other. This information includes the paths of the transformation, the models, and the metamodels in the distributed file system.

Depending on the execution phase, the master node takes care of assigning either a set of input model elements (map phase) or intermediate traces (reduce phase) for processing. However, during the map phase, all the nodes share read rights over the input model in case other elements are needed to execute a rule application. In the reduce phase instead, it's the tracing information that is shared, as it holds information about not only rule application, but also associated input and output model elements. ATL-MR supports two input data distribution modes, *greedy* and *random*.

6.1.2 Implementation

We implemented our approach as a distributed ATL engine. The engine is built on top of the ATL Virtual Machine (EMFTVM [125]). In the standard ATL VM, the transformation engine iterates the set of matched rules, and looks for the elements that match its application condition (guard). Instead, our VM iterates over each input model element, and checks if it is matched by an existing rule. In this perspective, we extended the ATL VM with a minimal set of functions (see Table 6.1) allowing the VM to run either in standalone or distributed mode. In particular, the distributed VM is required to factorize and expose methods for launching independently small parts of the execution algorithms. For instance the distributed VM exposes methods to perform the transformation of single model elements. Typically the methods `matchSingle(EObject)` and `localApplySingle(TraceLinkl)` called in the map phase, and `globalResolve()` called in the reduce phase.

Our implementation adopts some of the existing good practices and design patterns for scalable graph processing in MapReduce, namely the `InMapperCombiner` and the `Shimmy` patterns proposed by Lin et al. [88].

Like normal `Combiners`, the inMapperCombiner pattern aims at reducing the number of pairs emitted by each mapper. Apart from that, this pattern runs within the mapper instead of running as a separate step. In this case, the emission of ⟨key,value⟩ pairs is held until the processing ends, then invokes the inMapperCombiner, and hence reducing the amount of data passed to the *shuffle* phase. This prevents the framework from unnecessary creation and destruction of intermediate objects.

The second pattern relates to mappers-to-reducers dataflows. It discourages diffusing the graph structure in the network. Instead, it recommends passing only the metadata of the graph elements' state, which are mostly less densely inter-connected. In our implementation, target elements are persisted while tracelinks are passed along to reducers. Hadoop's Map and Reduce Classes are provided with two API hooks, being `Initialize` and `Close` hooks. While the `initialize` hook is used to set-up the transformation environment, according to running phase (`Local Match-Apply` or `Global Resolve`), the `close` hook is used to execute the InMapperCombiner pattern. That is by iterating the local tracelinks (intermediate pairs), and invoking the method `localApplySingle()` for each tracelink. This resolves the local properties and thus reduce the amount of data communicated to reducers.

6.1.3 Model-data distribution and access

Distributing input model. Data locality is one of the aspects to optimize in distributed computing for avoiding bottlenecks. In Hadoop, it is encouraged to run map tasks with input data residing in HDFS since Hadoop will try to assign tasks to nodes where data to be processed is stored.

Each mapper is assigned a subset of model elements by the splitting process. In the case of XMI models, we start first by flattening the structure of our model, which consists in producing a file containing model elements URIs as plain strings, one per line. Hadoop then takes care of shredding input data. With NEOEMF/COLUMN, this step is not necessary since models are already stored in table-like topology.

Hadoop provides several input format classes with specific splitting behaviour. Accordingly, we use different splitters depending on the persistence format of our input models. For XMI models, we use an *NLineInputFormat* on the flattened file, it takes as argument a text file and allows to specify the exact number of lines per split. Finally, the default record reader in Hadoop creates one record for each line of the input file. As a consequence, every map function in ATL-MR will be executing on a single model element. When running our distributed transformation on top of models stored in HBase, we use *TableInputFormat*. By default, this format divides at region boundaries based on a scan instance that filters only the desired cells. In our, we use a *KeyOnlyfilter*. This filter accesses just the keys of each row, while omitting the actual data. These keys are used later to access model elements stored in NEOEMF/COLUMN.

For an efficient distribution of the input model, ATL-MR is shipped with two data execution modes, a greedy mode and a random mode. While the random distribution is applicable to both of XMI and NeoEMF/Hbase persistence formats, the greedy mode is only applicable to NEOEMF/COLUMN. Note that the greedy algorithm improves only data access time, but since in XMI the whole model should be loaded, the greedy mode is not applicable. The greedy algorithm comes with default parameters (buffer size, variance, etc.), however, they can be set to the desirable values.

The greedy algorithm is implemented on top of a Hadoop `TableInpuFormat` class. We override its default behaviour and enable the use of our custom splitting solution. The implementation relies on the HBase storage scheme, where, ⟨key, value⟩ pairs are grouped by family and stored in increasing lexicographical order. Table splits are defined by a start key and end key. In our solution, we associate to each split an `ID`. Later on, we use this id

to salting the row keys belonging to the same splits. We applied our splitting approach to the `LocalResolve` phase of ATL-MR algorithm. The Global resolve uses a random assignment approach.

In order to perform the automated footprint extraction operation preceding our greedy assignment strategy, we rely on a static analysis tool of ATL transformations, anATLyzer [39]. Although, it was initially implemented to automate errors detection and verification in ATL transformations, in our solution, we use anATLyzer internals, especially, typing information inference of OCL expressions, in order to construct the footprints of ATL rules.

Accessing input models. Due to the file-based representation of XMI, models stored using this representation need to be fully loaded in memory, even though in practice, all the nodes need only a subset of these model elements. This prevents the framework from transforming VLMs. NEOEMF/COLUMN in the other hand, overcomes this limitation by providing a lazy loading mechanism, which enables the ATL-MR *slaves* to transparently load only the set of needed elements. However, because of the distributed nature of NEOEMF/COLUMN, it is mandatory to guarantee the consistency of the local values and the remote ones. For this, every time a client needs to read a value of a property, NEOEMF/COLUMN fetches it from the underlying backend. To our convenience, we take advantage of the fact that input models are read-only [1], and we use the *read-only* store. This store has the capacity to cache model elements after and properties values, after fetching them for the first time.

6.1.4 Failure management

One of the two fundamental bricks of Hadoop is Yarn. It is responsible for managing resources in a Hadoop cluster. Figure 6.2 shows the Yarn's architecture. The resource manager (master) and the node managers (slaves) altogether form the computation nodes. Node managers are in charge of launching containers. The application master is a per-application is tasked with negotiating resources and communicating with resource managers. The application startup processes as follows. First a client submits an application to the Resource Manager, then the Resource Manager allocates a container and contacts the related Node Manager. Later the Node Manager launches the container which executes the Application Master. Finally, the Application Master takes care of negotiating appropriate resources and monitoring them.

In distributed applications on MapReduce, four different kinds of failure may occur, namely, *task*, *Application Master*, *Node Manager*, and *Resource Manager* failures. Failures can take place for many reasons, e.g. downtime [2], runtime exceptions, etc..

Task failure happens when the JVM reports a runtime exception during either a map or reduce task. This exception is sent back to the application master, which marks the task attempt as failed, then frees the container to make the resource available for another task. Another scenario is when having hanging tasks. Here as well, tasks are marked as failed

1. We assume that input models are also not subject to changes during the transformation's execution
2. time during which a machine, especially a computer, is out of action or unavailable for use

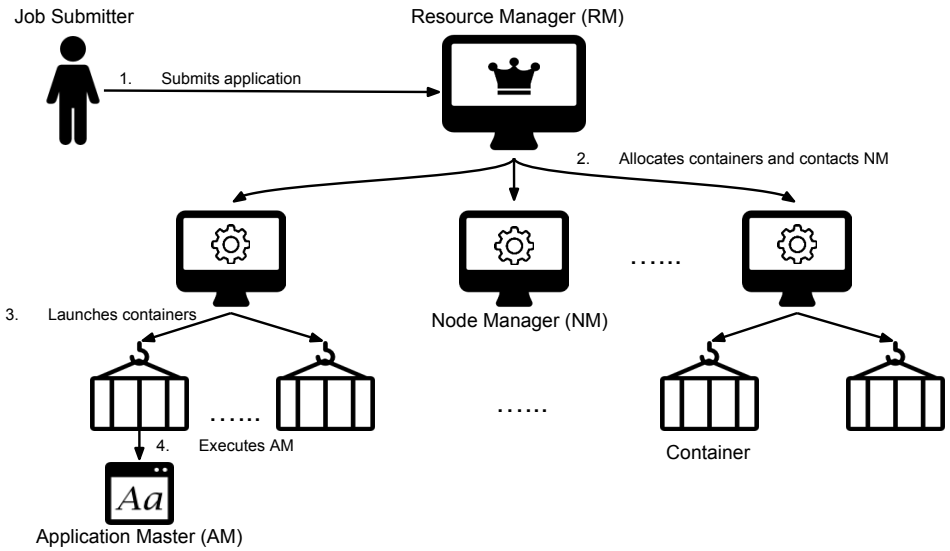


Figure 6.2 – Hadoop Yarn architecture

when the application master stops receiving heartbeats after some period of time. The timeout period is set to 10 minutes by default. When a job fails after a specific number of times, the whole Job is aborted. By default, the maximum attempts are four.

Application master failure happens when the resource manager stops receiving application's heartbeats. Such as tasks, application masters has also a maximum number of attempts that is set to two in the default configuration.

Node manager failure happens when the node crashes or runs slowly. In this case it fails to send heartbeats periodically. The timeout period is also set to 10 minutes and all the tasks running in this node are abborted and re-executed.

Resource manager failure is the most serious kind. It constitutes a single point of failure. If a resource manager crashes the whole job fails and no data can be recovered. To avoid running in such a scenario, it is necessary to run a pair of resources managers in the same cluster.

In ATL-MR, in case a task solely fails (for any of the reasons above) during the map phase, then the local output model together with the intermediate traces are cleaned, and the task is relaunched. Otherwise, if a reduce task fails, then updates on the global output model are left intact, and they get overridden when the reduce task is run for the next time. In case the whole job fails, the master node cleans all the data, and the transformation is re-executed from the beginning.

6.2 Distributed Persistence for MTs in ATL

As mentioned in section 5.3, NEOEMF/COLUMN is a persistence layer maintaining the same semantics as the standard EMF. Modifications in models stored using NEOEM-

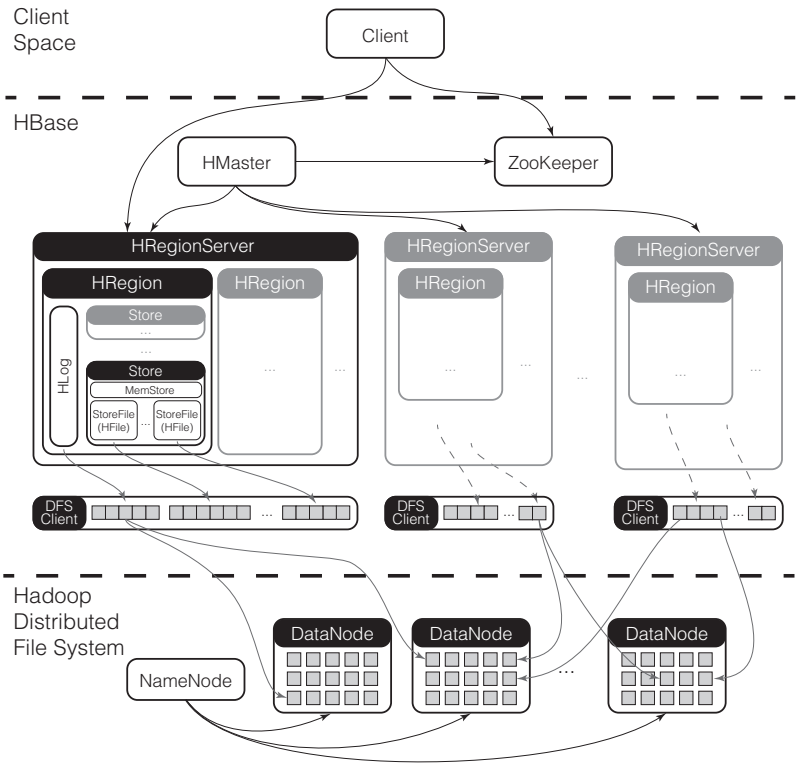


Figure 6.3 – HBase architecture

F/COLUMN are directly propagated to the underlying storage, making changes visible to all possible readers. Moreover, in Section 3.4, we showed how and when updates on model elements may occur concurrently. More importantly, we reasoned on the set of ACID properties that should be guaranteed for every MOF::Reflection method call so that the output models are consistent.

In what follows, we introduce the architecture of HBase and the characteristics of its client API. Soon after, we show how, while relying on these characteristics, we relax ACID properties to provide our transformation engine with a distributed persistence backend preserving the consistency of the output results.

6.2.1 HBase Architecture

Figure 6.3 shows how HBase is combined with other Apache technologies to store and lookup data. Whilst HBase leans on HDFS to store different kinds of configurable size files, ZooKeeper [123] is used for coordination. Two kinds of nodes can be found in an HBase setup, the so-called `HMaster` and the `HRegionServer`. The `HMaster` is the responsible for assigning the regions (`HRegions`) to each `HRegionServer` when HBase is starting. Each `HRegion` stores a set of rows separated in multiple column families, and each column family is hosted in an `HStore`. In HBase, row modifications are tracked by two different kinds of resources, the `HLog` and the `Stores`.

The `HLog` is a store for the write-ahead log (WAL), and is persisted into the distributed file system. The WAL records all changes to data in HBase, and in the case of a `HRegionServer` crash ensures that the changes to the data can be replayed. Stores in a region contain an in-memory data store (`MemStore`) and a persistent data stores

Table 6.2 – Summary of accesses counts to the underlying column-based storage system

METHODS	MAX COUNT*		ROWS	PROPERTIES	
	GET()	PUT()**		ATL	NEOEMF
OPERATIONS ON PROPERTIES					
`get*`	1	0	1	_C_D	ACID
`set*`	3	2	2	AC_D	AC(I)D
`isSet*`	1	0	1	_C_D	ACID
`unset*`	0	1	1	_C_D	ACID
OPERATIONS ON MULTI-VALUED PROPERTIES					
`add`	3	2	2	AC(I)D	AC(I)D
`remove`	3	2	2	AC(I)D	AC(I)D
`clear`	0	1	1	_C_D	ACID
`size`	0	1	1	_C_D	ACID

* Note that only max access count is taken under consideration, likewise for rows
** Updating two cells in the same row counts for one single `Put`

(`HFiles`, that are persisted into the distributed file system) `HFiles` are local to each region, and used for actual data storage.

6.2.2 Model Consistency in Distributed Model transformations with ATL-MR

HBase is not a strongly ACID-compliant database, it only provides ACID semantics on a per-row basis. It is up to users to employ this semantics to adapt ACID properties according to their applications behaviour. Notably, HBase is shipped with the following warranties:

— Any `Put` operation on a given row either entirely succeeds or fails to its previous state. This holds even across multiple column families. Same for the `Append` operation

— The `CheckAndPut` is atomic and updates are executed only if the condition is met, similar to the Compare And Set (CAS) mechanism

— Any `Get` or `Scan` operation returns a complete row existing at the same point in the table's history

— `Get` operations do not require any locks. All reads while a write in progress will be seeing the previous state

Table 6.2 extends Table 3.3 with the ACID properties that are guaranteed by NEOEMF/COLUMN, and the number of rows involved in each update. This time, the properties between parenthesis, in the column NEOEMF/COLUMN, refer to the properties partially supported by NEOEMF/COLUMN.

Hereafter, we describe how we guarantee the set of properties needed by ATL:

Atomicity — Modifications on a single object's properties are atomic. Modifications involving changes in more than one object, are not. One way to provide atomicity in this case is by manually implementing a rollback operation that undoes the changes affected by this commit. Since commits can be composed by at most 2 `Put` calls, the cost of rolling-back is low and changes can be tracked within the methods. In case the first `Put` succeeds but not the second, then there is only one `Put` to undo. Otherwise no rollback is needed.

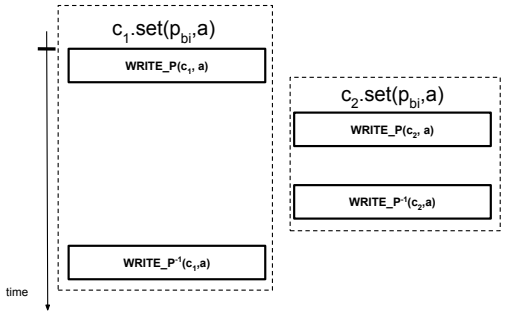


Figure 6.4 – Fail scenario of two concurrent calls on a 1-to-1 bidirectional reference. After both updates finish, they leave the system in an inconsistent state where, $c_1 \underset{p}{\rightarrow} \emptyset$, $c_2 \underset{p}{\rightarrow} a$, and $a \underset{p}{\rightarrow} c_1$

Consistency — Non-concurrent modifications on object's properties are always consistent. For that, we rely on EMF, which validates some consistency rules before a commit happens. The validation of rules issued from OCL invariants is not supported by EMF, likewise for NEOEMF/COLUMN. Failures in single-valued properties are handled by HBase, while multi-valued are handled by the framework (See Atomicity and Isolation).

Isolation — There is particular cases where isolation is not guaranteed in NEOEMF/COLUMN for the `set`, `add`, and `remove` methods (as shown in Table. 3.3). Figure 6.4 shows a fail scenario of two concurrent calls on a 1-to-1 bidirectional reference. Such a concurrent scenario would not occur in ATL. In fact, concurrent operations occur only on multi-valued properties in order to perform a monotonic update, either by adding or removing an element at a specific index. In order not to have inconsistent result in absence of write-write synchronization, concurrent writes should be provided with `isolation`. To do so, two possible solutions exist. The first one is using row locking on every write operation. This solution is not encouraged in HBase [3], and the API allowing row-locking has been removed in the latest versions of HBase. The seconcd one (what we use instead) is the CAS mechanism. In HBase, `CheckAndPut` operations is executed only when matching a specified condition on a property value. In NEOEMF/COLUMN, we use it for removing elements from multi-valued properties. The operation first reads an object it plans to later update, and commits the changes only if the value is still as current. All of this, in one atomic isolated operation. To add elements to properties we use `Append` operation, which is also atomic and isolated. For add operations at a specific index, we infer the append offset from the index itself, since all objects' UID have the same length. It can be noted that all the case scenarios where isolation is needed in ATL are supported by NEOEMF/COLUMN.

Durability — Modifications on a given object are always reflected in the underlying storage, even in the case of a `Data Node` failure, thanks to the replication capabilities provided by HBase.

3. https://issues.apache.org/jira/browse/HBASE-7315

6.3 Conclusion

In this chapter, we presented our prototype engine for distributed model transformations with ATL on MapReduce. ATL-MR is coupled with a decentralized model persistence framework, NEOEMF/COLUMN. It is also supplied with two input data partitioning modes for memory efficient transformations, a random mode and a greedy mode. ATL-MR transparently integrates with EMF-Applications.

In the next chapter, we evaluate the scalability of our proposed engine by comparing how the transformation of our running examples performs in different test environments. The transformation covers a sufficient set of declarative ATL constructs enabling the specification of a large group of MTs.

Thanks to our publicly available execution engine, in a form a Docker [4] cluster, users may exploit the availability of MapReduce clusters on the Cloud to run model transformations in a scalable and fault-tolerant way.

4. https://www.docker.com/what-docker

7

Evaluation

In this chapter we evaluate the scalability of our solutions by running them against some existing benchmarks. These benchmarks are issued from our initiative for building an openset benchmark for scalable query and transformation engines [18] (can also be found in the appendix section). The experiments are run on different setups and environments. Input data used in the experiments come from real-world use cases as well as our random generator available online [1]. The wiki contains instructions about how to use our generator. These evaluations are presented in detail in our previous work [15, 17, 19].

In the first section, we show how ATL-MR scales up in a complex transformation example with quadratic complexity. We run two experiments with two different setups and environments. The experiments are complementary and use the *Control2DataFlow* transformation. The second section evaluates the performance of our proposed solution for data partitioning while running it on a well known model transformation, *Class2Relational*. It also validates our integration with the NEOEMF/COLUMN framework. Finally, we evaluate the scalability of the access time of NEOEMF/GRAPH in increasingly large scenarios, and we compare it against CDO and XMI. This last experiment uses models reverse engineered from Java Projects using MoDisco [27].

7.1 Distributed model transformation

Description

In this experiment we show how ATL-MR scales up in a complex transformation example with quadratic complexity. We evaluate the scalability of our proposal by comparing how the *Control2DataFlow* (see Section 2.1.3) transformation performs in two different test environments. In order to be able to compare our distributed VM to the standard one,

1. https://github.com/atlanmod/neoEMF-Instantiator

we opt for a complex transformation with quadratic time complexity over XMI models that can fit in memory.

We use as input different models of diverse sizes. The original case study [61] already includes a set of input models for the benchmark. These models are reverse-engineered from a set of automatically generated Java programs, with sizes up to 12 000 lines of code. For our benchmark we used the same generation process but to stress scalability we produced larger models with sizes up to 105 000 lines of code. We consider models of these sizes sufficient for benchmarking scalability in our use case. In our experimentation, processing in a single machine the largest of these models takes more than four hours. All the models we generated and the experimentation results are available at the tool's website.

In what follows we demonstrate the scalability of our approach through two different but complementary experimentations. The first one shows a quasi-linear speed-up [2] w.r.t. the cluster size for input models with similar size, while the second one illustrates that the speed-up grows with increasing model size.

7.1.1 ATL-MR on XMI: Speed-Up curve

Input data

For this experiment we have used a set of 5 automatically generated Java programs with random structure but similar size and complexity. The source Java files range from 1 442 to 1 533 lines of code. We used a simple M2T to generate these java files. The execution time of their sequential transformation ranges from 620s to 778s.

Environment setup

The experiments were run on a set of identical *Elastic MapReduce* clusters provided by *Amazon Web Services*. All the clusters were composed by 10 EC2 instances of type *m1.large* (i.e. 2 vCPU, 7.5GB of RAM memory and 2 magnetic Hard Drives). Each execution of the transformation was launched in one of those clusters with a fixed number of nodes – from 1 to 8 – depending on the experiment. Each experiment has been executed 10 times for each model and number of nodes. In total 400 experiments have been executed summing up a total of 280 hours of computation (1 120 normalized instance hours[3]). For each execution we calculate the distribution speed-up with respect to the same transformation on standard ATL running in a single node of the cluster.

Results

Figure 7.1 summarizes the speed-up results. The approach shows good performance for this transformation with an average speed-up between 2.5 and 3 on 8 nodes. More importantly, as it can be seen on the upper side, the average speed-up shows a very similar curve for all models under transformation, with a quasi linear speed-up indicating good scalability w.r.t. cluster size. We naturally expect the speed-up curve to become sub-

2. an increase in speed in machine's rate of working

linear for larger cluster sizes and very unbalanced models. The variance among the 400 executions is limited as shown by the box-plots in the lower side.

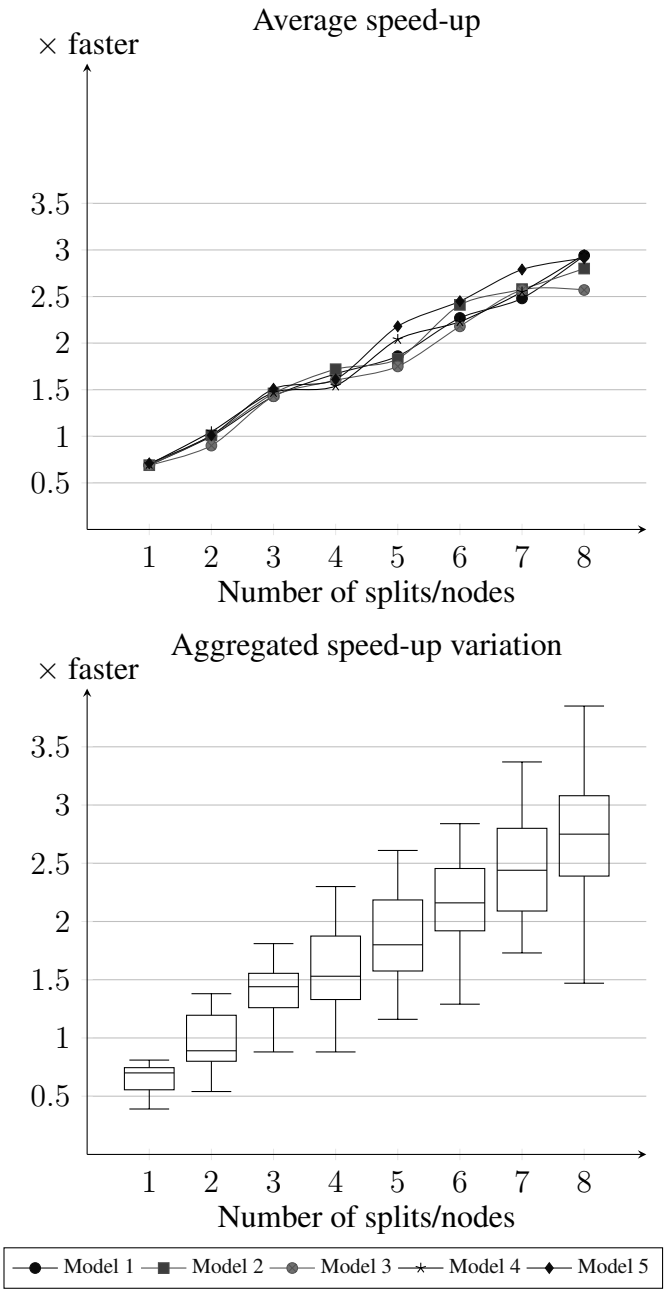


Figure 7.1 – Speed-up obtained in Experiment I

Table 7.1 – Execution times and speed-up (between parentheses) per model

#	Size	Elts	Std. VM	Distributed VM using x nodes (time and speed-up)							
				1	2	3	4	5	6	7	8
1	∼4MB	20 706	244s	319s (×0.8)	165s (×1.5)	128s (×1.9)	107s (×2.3)	94s (×2.6)	84s (×2.9)	79s (×3.1)	75s (×3.3)
2	∼8MB	41 406	1 005s	1 219s (×0.8)	596s (×1.7)	465s (×2.2)	350s (×2.9)	302s (×3.3)	259s (×3.9)	229s (×4.4)	199s (×5.1)
3	∼16MB	82 806	4 241s	4 864s (×0.9)	2 318s (×1.8)	1 701s (×2.5)	1 332s (×3.2)	1 149s (×3.7)	945s (×4.5)	862s (×4.9)	717s (×5.9)
4	∼32MB	161 006	14 705s	17 998s (×0.8)	8 712s (×1.7)	6 389s (×2.3)	5 016s (×2.9)	4 048s (×3.6)	3 564s (×4.1)	3 050s (×4.8)	2 642s (×5.6)

7.1.2 ATL-MR on XMI: Size/Speed-Up correlation

Input data

To investigate the correlation between model size and speed-up, we execute the transformation over 4 artificially generated Java programs with identical structure but different size (from 13 500 to 105 000 lines of code). Specifically, these Java programs are built by replicating the same imperative code pattern and they produce a balanced execution of the model transformation in the nodes of the cluster. This way, we abstract from possible load unbalance that would hamper the correlation assessment.

Environment setup

This time the experiments have been executed in a virtual cluster composed by 12 instances (8 slaves, and 4 additional instances for orchestrating Hadoop and HDFS services) built on top of OpenVZ containers running Hadoop 2.5.1. The hardware hosting the virtual cluster is a Dell PowerEdge R710 server, with two Intel® Xeon® X5570 processors at 2.93GHz (allowing up to 16 execution threads), 72 GB of RAM memory (1 066MHz), and two hard disks (at 15K rpm) configured in a hardware-controlled RAID 1.

Results

As shown in Figure 7.4 and Table 7.1, the curves produced by Experiment II are consistent to the results obtained from Experiment I, despite the different model sizes and cluster architectures. Moreover, as expected, larger models produce higher speed-ups:

for longer transformations the parallelization benefits of longer map tasks overtakes the overhead of the MapReduce framework.

Discussion

Being based on the EMF, ATL uses its de-facto serialization format, XMI [107]. This file-based representation faces many issues related to scalability. In particular, models stored in XMI need to be fully loaded in memory, but more importantly, XMI does not support concurrent read/write. This hampers distributed model transformations on ATL at two levels, first, all the nodes should load the whole model even though they only need a subset of it. This prevents us from transforming very big models that would not fit in memory. The second one concerns the reduce phase parallelization, and this is due to the fact that only one reducer can write to the output XMI file at once. In order to overcome this issue, we use NEOEMF/COLUMN to enable concurrent read/write. In our next experiment, we report our results while integrating ATL-MR with NEOEMF/COLUMN.

Choosing the right number of splits has a significant impact on the global performance. Having many splits means that the time that is taken to process each split will be small compared to the time to process the whole input. On the other hand, if splits are too small, then the overhead of managing the splits and creating map tasks for each one of them may dominate the total job execution time. In our case we observed better results where the number of splits matches the number of available workers. In other words, while configuring Distributed ATL, the number of lines per split should be set to $\frac{model\ size}{available\ nodes}$.

Finally, one should be careful with Hadoop (Yarn) configuration options. In the case of memory or time consuming model transformations it is required to set up correctly these options in order to avoid the system's failure. Table 7.2 depicts some of these options. Note that Node Manager resources are shared among all the containers running under it. The number of containers is configurable as well (see Table 7.2).

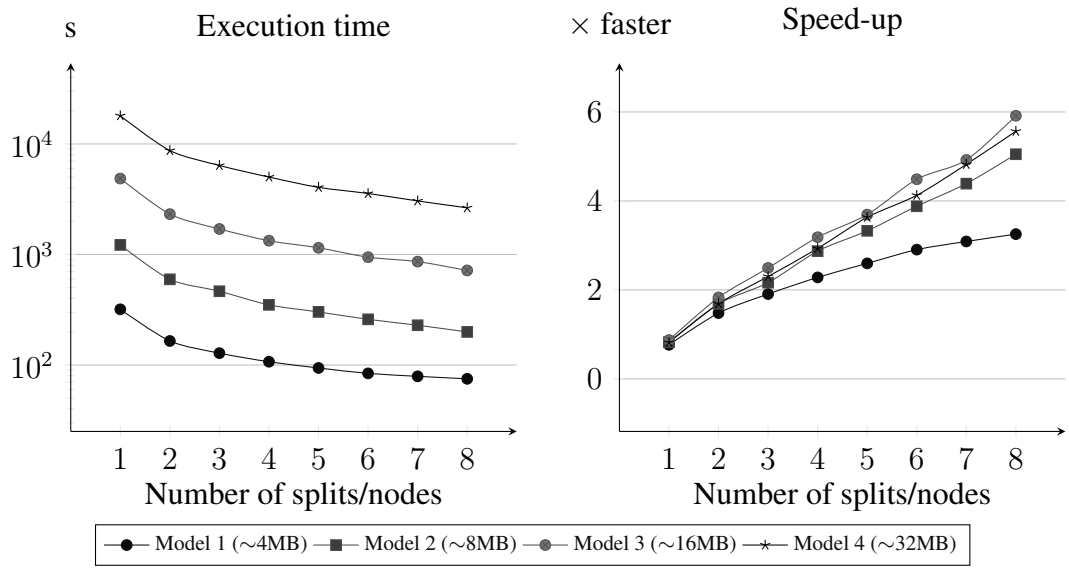


Figure 7.2 – Execution times and speed-up on Experiment II

Table 7.2 – Yarn Options

OPTION	DESCRIPTION
yarn.nodemanager.resource.memory-mb	Amount of physical memory for all containers
yarn.nodemanager.container-monitor.interval-ms	How often to monitor containers
yarn.nodemanager.resource.cpu-vcores	Number of CPU cores that can be allocated for containers
yarn.nodemanager.disk-health-checker.max-disk-utilization-per-disk-percentage	The maximum percentage of disk before failure

7.2 Optimized data partitioning

7.2.1 Description

This experiment evaluates the performance of our greedy data partitioning algorithm against the random approach. The Hadoop framework comes with tools to help analyzing data being processed. This feature gathers statistics about the job, for different purposes like quality control or application level statistics. Each counter is maintained by a mapper or reducer, and periodically sent to the application master so it can be globally aggregated. Counter values are definitive only once a job has successfully completed. Hadoop allows user code to define user-defined counters, which are then incremented as desired in the mapper or reducer. We defined a Hadoop counter for globally reporting metrics about data access and update. This counter was integrated with our distributed persistence backend, NEOEMF/COLUMN. It globally reports *remote access count, local access count* at the end of every successful job. We believe that such metrics are relevant as they can be decoupled of the internal complexities of both the transformation and persistence engines.

Algorithm 6 describes how we proceed with our experimentation. For each model size, we generate three random models. Each execution of the transformation on the generated model, is launched with a fixed number of nodes (2..8).

Algorithm 6: Data distribution algorithm

Input : Stream<Vertex> $stream$, int $avgSize$, int $bufCap$, int m

```
1 foreach size ∈ sizes do
2     foreach pass ∈ 1..3 do
3         input ← generateInputModel(size) foreach map ∈ 2..8 do
4             transformUsingRandomDist(input, map)
5             transformUsingGreedyDist(input, map)
6             cleanRound()
7     deleteInputModel()
```

Input data

We evaluate the performance of our proposed solution while running it on a well known model transformation, *Class2Relational* (See Section 4.2). We use as input randomly generated models with diverse sizes (we provide generation seeds to make the generation reproducible). The generator is highly configurable. The default configuration takes in input, the model size, the density of references, and the variation. The variation applies to both the model size, and the number of references to be generated per property. In our experiment, we used a density of 8, and a variation of 10%.

Environment setup

This experiment has been executed on a virtualized cluster composed of 12 instances. This time we use Docker [3] to build and deploy our machines on top of an AWS instance of type r3.4xlarge. This instance type allows up to 8 execution threads and has a 61Gb of RAM, and an Intel® Xeon® E5-2670 v2 (Ivy Bridge) Processor at 2.50 GHz. The Docker images [4] to reproduce the experiment along with the instructions on how to build and deploy them are availble online [5].

Results

Table 7.3 summarizes the average improvement results by number of access counts to the underlying backend. Each access either fetches an element or its properties. Our distributed solution shows an average improvement up to 16.7%. However, in some cases, the random distribution performed better than our greedy algorithm, mostly for the set of models with sizes 5000 and 10000. For the smallest set of models, it did not bring any improvement in average for 5 and 6 splits respectively.

Figure 7.3 shows the performance's distribution of our algorithm per number of splits. For the 'Class2Relational' transformation and models with sizes ranging between 5000 and 20000, partitioning the model into 2 to 5 splits, guarantees better performance, where it reached up to more than 26%.

Discussion

A success key to our algorithm is the approximation quality of our dependency graph. The graph structure grows gradually as model elements arrive. Each element may or may not participate in the construction process of the graph (i.e. the element has dependencies or not). The sooner (during our partitioning process) we have a good quality approximation of our dependency graph, the better splitting we can achieve.

Our partitioning algorithm is subject to various internal as well as external factors that may impact its performance. Inconveniently, these factors are completely orthogonal to each other. More importantly, the order in which elements arrive impacts the quality of the

3. https://www.docker.com/what-docker
4. https://hub.docker.com/u/amineben/
5. https://github.com/atlanmod/hadoop-cluster-docker

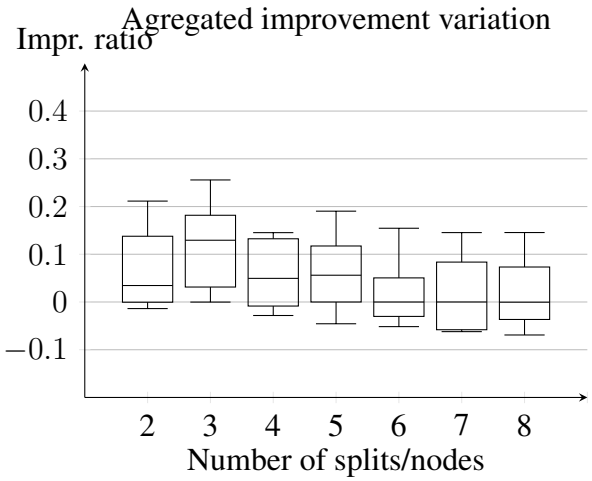


Figure 7.3 – improvement obtained in the experiment (Table 7.3)

dependency graph approximation, and hence, the performance of our algorithm. Hereafter, we elaborate on the repercussions of these factors.

The choice of the appropriate parameters is crucial. On teh one hand, both of the average size (which implicates the number of splits) and the buffer capacity depend on the memory capacity of a single commodity machine [6]. On the other hand, the size of the buffer plays an important role in delaying the assignment of some model elements – low-priority ones. This delay is beneficial when the dependency matrix evolves from the time a model element was buffered until it is cleared from the buffer. With big enough buffer capacity, the master node can improve the decision making of the assignment to a split after having a better approximation of the dependency graph. However, the performance of the algorithm can be compromised when receiving a stream of low-priority elements in a row, until the buffer reaches its limit. In this case, the quality of the dependency graph does not improve.

Furthermore, in the default behaviour of our algorithm, we define a high-priority model element as any element that participates in the construction of the approximated dependency graph. Still, participating elements can have a different priority. The priority order can be quantified by the amount of contribution to the construction of the graph approximation. Elements with a bigger amount of dependency elements contribute more. In some cases, where the model transformation and the possible topology can be known in advance, this definition can be customized to improve the performance of our algorithm. In our running example, elements of type *Package* and *Class* can have high-priority as they are the more likely to have a big number of dependencies. Nonetheless, we showed in our experiments that the default behaviour can bring a considerable improvement in any case.

Finally, the performance of our approach may be reduced when (i) having elements with big amounts of dependencies (sometimes exceeding the average size of the split), (ii) having approximate graph containing false positive dependencies, and (iii) having an unfavourable order of streamed elements. The first limitation is typically encountered when having complex OCL expressions starting with the operation *allInstances()* or when having recursive helpers. The second limitation happens when having OCL expressions involving several occurrences of filtering operations (*select()* or *reject()*) followed by a

6. Assuming that the cluster is homogeneous

collect. While the *collect()* operation operates only on the filtered model elements, in our approach, non-filtered elements are also traversed. In case the number of filtered elements is considerable, our algorithm reports elements as false positive. This may lead to grouping non-dependent elements in the same split.

Table 7.3 – Remote access count (per properties) : Random Vs. Greedy

DIST. MODES	SIZES	SPLITS						
		2	3	4	5	6	7	8
Random	5000	10655	10089	8968	8087	8085	8320	8377
	10000	19871	19286	17198	18529	16390	17342	16386
	15000	31402	28582	26937	25214	24785	24617	24520
	20000	35060	35053	38325	42405	38109	35045	35045
Greedy	5000	9853 (7.5%)	8698 (13.7%)	8146 (9.17%)	8088 (-0%)	8086(-0%)	8493 (-2.0%)	8083 (3.5%)
	10000	17211 (13.3%)	16396(14.9%)	16493(4.1%)	16395(11.5%)	16673(-1.7%)	17227(0.0%)	16949(-3.4%)
	15000	27863(11.2%)	23912(16.3%)	24850(7.7%)	23908(5.1%)	23905(3.5%)	23904(2.9%)	23900(2.5%)
	20000	32855(6.29%)	32691(6.74%)	36207(5.53%)	38688(8.77%)	35232(7.55%)	32362(7.67%)	32361(7.66%)

7.3 Scalable persistence

Description

Thanks to the extensibility of NEOEMF, it was provided with three different backends, each one of them is suitable for a specific scenario. While we showed in the previous section the performance of NEOEMF/COLUMN in a distributed scenario, in this section we evaluate how the access time of NEOEMF/GRAPH scales in increasingly large scenarios. NEOEMF/GRAPH and NEOEMF/MAP come with embedded databases, which is suitable for local model-based applications. We compare the performance of NEOEMF/COLUMN against CDO (with H2 as relational database backend) and XMI. The performance evaluation of NEOEMF/MAP is reported in another work [58].

We run two different experiments:

Experiment I : Model traversal. In a first experimentation we execute a model visitor that starting from the root of the model traverses the full containment tree in a depth-first order. At each step of the traversal the visitor loads the element content from the backend, and modifies the element (changing its name). Only the standard EMF interface methods are used by the visitor, that is hence agnostic of which backend he is running on. During the traversal we measure the execution times for covering 0.1%, 1%, 10% 50% and 100% of the model.

Experiment II : Java reverse engineering. In a second experimentation we execute a set of three simple queries on the Java metamodel that originate from the domain of reverse-engineering Java code. While the first of these queries is a well-known scenario in academic literature, the other two have been selected to mimic typical model access patterns in reverse engineering, according to the experience of our industrial partner.

1. Grabats (GB): it returns the set of classes that holds static method declarations having as return type the holding class (e.g., Singleton) [75].
2. Unused Method Declarations (UnM): it returns the set of method declarations that are private and not internally called.
3. Class-Attribute Map (CA-M): it returns a map associating each Class declaration to the set of its attribute declarations.

All these queries start their computation by accessing the list of all the instances of a particular element type, then apply a filtering to this list to select the starting points for navigating the model. In the experience of our industrial partner this pattern covers the quasi-totality of computational-demanding queries in the reverse-engineering domain. For this reason we added a method *getAllInstances* to the EMF API and we implemented it in all the three back-ends. In CDO we implemented this method by a native SQL query, achieved through the union of the tables containing elements of the given type and its subtypes. In NEOEMF the same result is achieved by a native Neo4j query traversing the graph nodes via the relationship `INSTANCE_OF`, for the given type and all of its subtypes. The user-code of each of the three queries uses this method to start the computation in all implementation, hence remaining backend-agnostic.

Execution environment

Experiments are executed in a laptop computer running Windows 7 Enterprise 64. The most significative hardware elements are: an Intel Core i7 processor 3740QM (2.70GHz), 16 GB of DDR3 SDRAM (800MHz), and a Samsung SM841 SATA3 SSD Hard Disk (6GB/s). Experiments are executed on Eclipse version 4.3.1 running Java SE Runtime Environment version 1.7 (specifically, build 1.7.0_40-b43).

Input data

These experiments are performed over 3 EMF models that conform to the Java Metamodel proposed in MoDisco [27] and reverse-engineered from existing code using the MoDisco Java Discoverer. As starting code we used 3 sets of Eclipse plugins, of increasing size. Table 7.4 details how the experiments vary in size and thus in the number of elements. In order to compare the three technologies, we generate three different EMF access APIs, starting from the Java MoDisco Metamodel respectively with 1) EMF standard parameters, 2) CDO parameters, and 3) NEOEMF generator. We import the 3 experimental models, originally in XMI format to CDO and NEOEMF/GRAPH, and we verify that all the imported models contain the same data.

Table 7.4 – Overview of the experimental sets

#	Plugin	Size	Number of elements
1	org.eclipse.emf.ecore	24.2MB	121.295
2	org.eclipse.jdt.core	420.6MB	1.557.007
3	org.eclipse.jdt.*	984.7MB	3.609.454

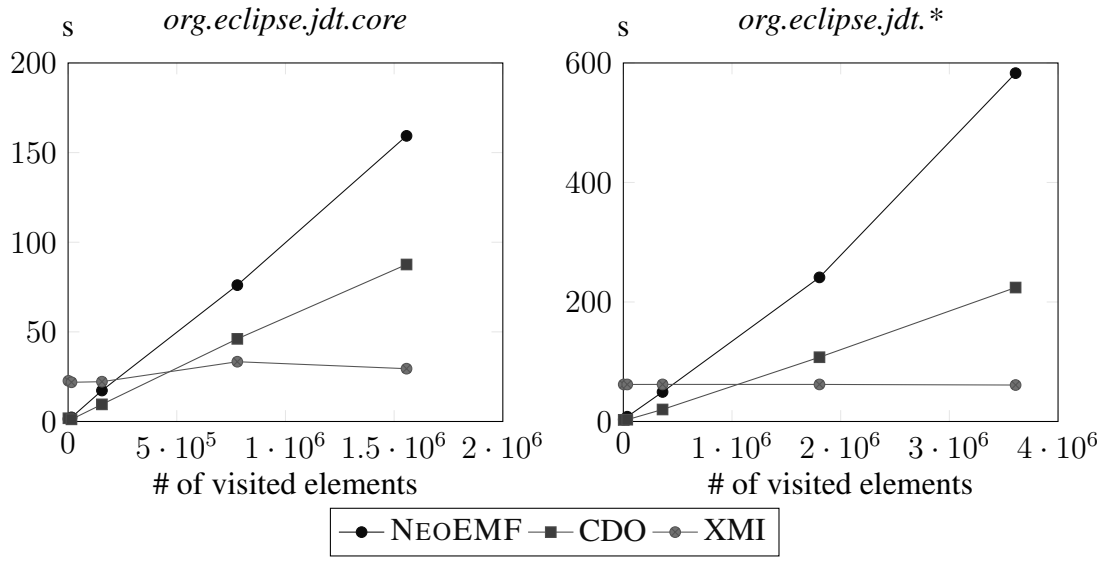


Figure 7.4 – Results for model traversal on test models 2 and 3.

Results

Figure 7.4 shows the results of the first experimentation over the two largest test models (org.eclipse.jdt.core and org.eclipse.jdt.*). Markers in the graph curves refer respectively to percentages previously mentionned. The results from Figure 7.5 show that both NeoEMF and CDO outperform XMI. The test also confirms the previous result, showing execution times from CDO consistently lower than NeoEMF.

The results of the two experimentations are consistent with each other. Figure 7.4 shows that while in XMI the access time to each model element is negligible with respect to the initial model-loading time (since the whole model is loaded in memory), the two backends with on-demand loading mechanisms have a constant access time (giving linear complexity to the query). This shows that the backends can scale well for even larger sizes. In both experiments in Figure 7.4 the backends with on-demand loading mechanisms outperform XMI when the part of the model that needs to be accessed is lower than a certain ratio of the full model. The graphs show that this ratio is approximately constant, independently of the size of the model and it amounts to 14.12% and 12.46% for NeoEMF and 29.54% and 27.84% for CDO. The CDO backend performs better than NeoEMF, by an approximately constant factor that in the two experiments is respectively of 1.38 and 2.6.

Discussion

Summarizing, while resulting in a better solution than XMI for the use case under study, the current version of NeoEMF does not exhibit the performance optimizations prefetching of more mature solutions like CDO. CDO supports partial collection loading that gives the possibility to manage the number of elements to be loaded when an elements is fetched for the first time. Likewise, CDO provides a mechanism to decide how and when fetching the target objects asynchronously. In recent work [42], NeoEMF/Graph was integrated with a framework for the prefetching and caching of EMF models.

We also remark that the acceptable performances of XMI may be misleading in a real-world scenario: the amount of memory we used allowed to load the whole models in memory, avoiding any swapping in secondary memory that would have made the XMI solution completely unusable for the scenario. Moreover the use of an SSD hard disk significantly improved the loading & saving times from file. On-demand loading allows to use only the necessary amount of primary memory, extending the applicability of MDE tools to these large scenarios.

We did not measure significant differences in memory occupation between CDO and NEOEMF, but we noticed several problems in importing large models in CDO. For instance CDO failed to import the test model 3 from its initial XMI serialization on a 8Go machine, as a `TimeOutException` was raised.

Finally, the comparison with relational databases backend should also take into account several other features, besides execution time and memory in a single processor configuration. NEOEMF allows existing MDE tools to make use from now of the characteristics of graph databases like Neo4j, including clustering, online backups and advanced monitoring.

Conclusion

In this chapter we demonstrated the scalability of our approaches while running them against some well-known case studies.

In the first experience we proved the scalability of our solution by running it on a complex transformation over XMI models fitting in a memory of single machines in the cluster. We run two complementary experiments. The first one shows a quasi-linear speed-up w.r.t. the cluster size for input models with similar size, while the second one illustrates that the speed-up grows with increasing model size. ATL-MR runs up to ∼6× faster compared to the regular VM while distributing it over 8 machines. An experience in Hadoop cluster administration is still required to properly configure Hadoop options in order not to run into a system failure.

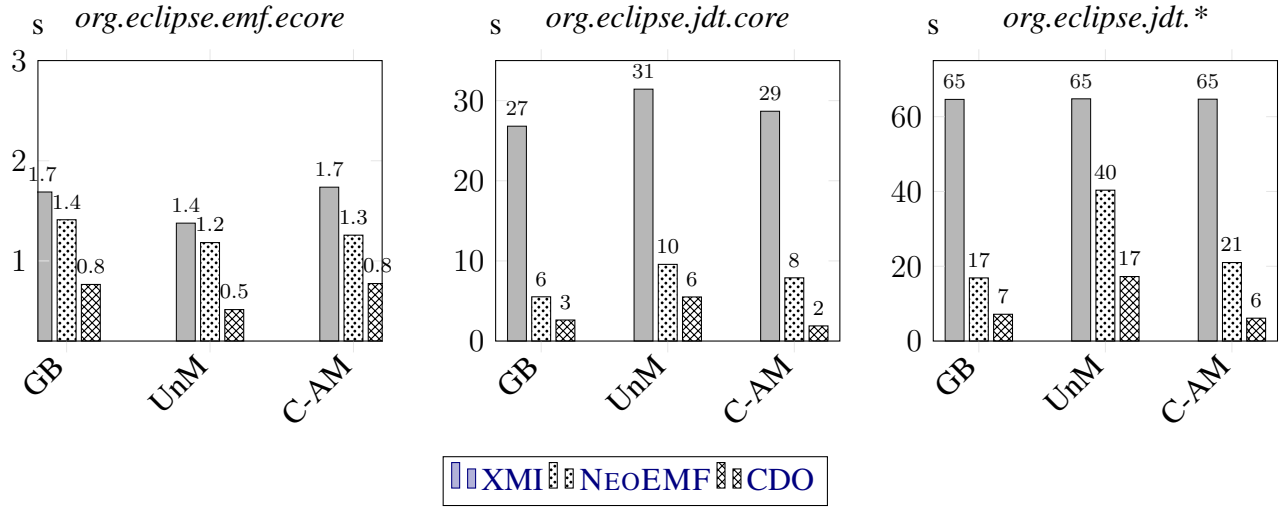


Figure 7.5 – Results for scenario 2

In the second experiment, we evaluated the performance of our greedy algorithm. We tested our solution against random partitioning, and showed a consistent benefit in access times up to 16%. However, the performance of our approach may be reduced when having elements with big amounts of dependencies, or an approximate graph containing false positive dependencies, or an unfavourable order of streamed elements.

In the last experiment, we assessed the scalability of the access times of NEOEMF/-GRAPH in increasingly large scenarios, and we compared it against CDO and XMI. The experiments showed a considerable improvement when in NEOEMF/GRAPH in CDO when the part of the model that needs to be accessed is lower than a certain ratio of the full model. The experiment also shows that CDO outperforms NEOEMF/GRAPH with constant factor that in the two experiments is respectively 1.38 and 2.6. This finding joins our earlier claim that is graph databases are not well-suitable for queries performed using EMF API calls (atomic operations). Related work [41] show that NEOEMF/GRAPH has better performance when having complex queries when built-in traversals. Another work [59] shows that NEOEMF/MAP is a better fit when performing querying using the EMF API. Indeed, NEOEMF/MAP is up to $\sim 9\times$ faster than CDO.

Many reasons hampered us from providing a thorough evaluation while comparing our solution to existing MT engines. Existing solutions either run the transformation completely in memory [31], or do not rely on the same modeling framework [84, 132] for accessing and persisting model-data. In future work, we intend to conduct a benchmark involving various model transformations for different intents and with different complexities. Our objective will be to decide on precisely, on which circumstances using ATL-MR becomes beneficial.

8

Conclusion: distributing MTs with ATL-MR & beyond

8.1 Introduction

Model transformation is one of the main concepts in MDE as it provides means to manipulate and understand models. Relational model transformation languages, thanks to their declarative nature, ease the specification of such operation in a compact and concise way. Besides, relational model transformations take implicitly care of the internal complexities of model transformation execution.

Recent application in model-based software industry, are characterized by big amounts of data that may be represented by VLMs (e.g. models with millions of elements). Against models of such size, current execution engines for MT languages easily hit the limit of the resources (IO/CPU) of the underlying machine. Moreover, existing persistence frameworks are not suitable for handling models alike.

On the other hand, the availability of large data processing and storage in the Cloud is becoming a key resource for part of today's industry, within and outside of IT. It offers a good alternative for companies to process, analyze, and discover new data insights, yet in a cost-efficient manner. Thanks to existing Cloud computing companies, this facility is extensively available for rent. This ready-to-use IT infrastructure is equipped with a wide range of distributed processing frameworks, for companies that have to occasionally process large amounts of data.

In this thesis, we exploited the recent emergence of systems and programming models for distributed and parallel processing to leverage the distributed transformation and persistence of VLMs. In particular, we relied on a well-known distributed programming model, MapReduce, in order to enable the distributed execution of model transformation in a reliable and fault-tolerant manner. We also adopted NoSQL databases as a new solution

for persisting VLMs. We coupled our solution with different kinds of NoSQL stores, where each store is suitable for a specific modeling scenario.

8.2 Objectives revisited

We proposed a solution for distributing relational MTs based on the MapReduce paradigm. In our solution we opted for a data-driven scheme as most existing graph processing algorithms reason on data. We identified three key ingredients for efficient and consistent distributed model transformations, and we identified the requirements to be guaranteed by each one of these components.

Distributed model transformation engine– We showed that relational model transformations thanks to their execution semantics, can be run in parallel in two steps. In the first step, each mapper is assigned a subset of model elements using a splitting process on which it runs the transformation locally. Every machine has a global view of the input model for resolving other dependencies than the elements in the chunk. For each model element, if it matches the input pattern of an existing rule, then a target element is created together with tracing information, and target properties corresponding to locally assigned elements are resolved. Information about non-local elements is stored in traces and sent to the reduce phase for a global resolve. This step is responsible for composing the sub-models resulting from the previous phase into a single global output model. The syntax of the MT language is still the same and no distribution primitives are added. However, an expertise in Hadoop cluster management is still required to tune the cluster's performance and avoid system failure in case of very complex model transformations. Moreover, although in our approach we do not support ATL's imperative blocks, most of MTs in MDE can be implemented using only the declarative part. Finally, we demonstrated the scalability of our approach in a complex transformation example with quadratic complexity. Instructions on how to use our prototype together with a ready-to-use Docker cluster are available in the tool's website [1].

Efficient data partitioner for distributed model transformation– We presented our solution for efficient partitioning of distributed MTs as a greedy algorithm. Based on the observation that the transformation's footprints can provide us with basic knowledge about the dependencies of a model element, we relied on the static analysis of model transformations to extract this knowledge. First, we showed that our problem can be related to the problem of uniform graph partitioning. Later, we adapted existing formalization to our system. Then, we proposed a greedy data distribution algorithm for declarative model transformations. However, the performance of our approach may be reduced when having elements with big dependencies or receiving model elements in an disadvantageous order. This solution was integrated with our distributed MT engine, ATL-MR.

Efficient model persistence framework– Finally, we introduced our solution for efficient distributed storage and access of models in EMF, NEOEMF/COLUMN. We examined the execution semantics of atomic model manipulation operations in ATL and exhibited the minimal set of ACID properties to preserve the consistency of the output model. NEOEMF/COLUMN is part of NEOEMF, a framework for efficient

1. https://github.com/atlanmod/ATL_MR/

persistence of models in EMF. NEOEMF implements three different backends, each of them is well-suitable for a specific modeling scenario. NEOEMF comes with a set of caching strategies and a lightweight loading and saving mechanisms to boost its performance. NEOEMF/COLUMN was integrated with ATL-MR to enable concurrent read/write.

8.3 Future work

In this section, we gather some directions we already briefly mentioned throughout this thesis to improve our solutions. We briefly them, then we investigate other lines of future work.

8.3.1 Improving the efficiency of our distributed transformation engine

We intend to improve the efficiency of our distributed transformation engine by exploring the following lines:

— Reducing the number of ⟨key,value⟩ pairs transmitted between the *Local Match-Apply* and the *Global Resolve* phases can improve the time to perform the *shuffling* phase. In future work, we want to cut down the amount of transmitted elements based on the static analysis of the bindings of the transformation rules as well as inter-rules dependency. An example of tracing information that can be omitted, are the ones involving model elements that will not be resolved by any other target element.

— A similar line is improving the distribution of, not only the input model of the transformation, but also its intermediate transformation data (tracing information). This can be achieved by assigning to the same machine elements sharing the resolution operation of the same set of model elements.

— Given the high density of models implicated in MDE, we plan to extend our work on efficient model partitioning to balanced edge partitioning, and conduct a thorough study on the impact of the model density and the use of different penalty functions in the partitioning strategy.

— In the current version of ATL-MR, the fault-tolerance relies completely in failure-management as provided by Yarn. Since in ATL-MR, output models are directly written to the persistence store, we can extend ATL-MR with a centralized tracking system that helps the transformation engine recovering from the previous state and carry on with the transformation.

— In model transformation-based applications where data access patterns are known beforehand, it is possible to group dependent elements on the same datanode in order to simplify their processing. We plan to extend the NEOEMF/COLUMN framework with the ability to decide, at model-element creation time, which machine should host which element. We can exploit our static analysis technique for this matter.

— In our solution for efficient data partitioning, we focused more on minimizing data access to the persistence backend as well as the number of elements loaded in each

machine. In some scenarios where IO is not a concern, another aspect that can be improved is the computation cost. In this case, the objective function turns into balancing the computational complexity in every machine. In this perspective, we also plan to adapt our formalization to extend NEOEMF/COLUMN with additional metadata (e.g. collection size) storage to help better predicting binding execution time and complexity.

8.3.2 Application to other relational transformation languages

Some relational MT languages like ETL and QVT/Relations share most of their properties with ATL. For instance, ETL and QVT also rely on tracing information for resolving target elements. The ETL also runs in two step as in ATL. First, the ETL engine allocates target model elements for source model elements that have been matched. Then, each allocated target element is initialized by executing the associated bindings. QVT instead has a different execution semantics where for every rule application, target elements together with their properties are initialized. However, by analysing the transformation rules it is possible to determine the number of map phases needed to run the transformation based on the causality dependency of the transformation rules. In future work we intend to generalize our approach to this family of languages.

8.3.3 A fully distributed OCL engine

OCL is a central component in model transformation languages. It is used to specify guards, query models, target sets, etc.. Collection operations are the most computationally expensive operations in OCL. One way to break down their complexity is by providing parallelism at collection operations level.

Existing frameworks such as Spark and FlumeJava built on top of MapReduce, may simplify this as they are provided with a higher level of abstraction at a collection (dataset) level. FlumeJava presents parallel collections and their operations. FlumeJava compiles parallel operations over these collections to a dataflow-like execution plan that is later optimized on appropriate MapReduce primitives. However, OCL does not fit in the class of application that can be efficiently handled as acyclic dataflows. For instance, it is not possible to handle recursive calls or closure operation using a fixed acyclic dataflow.

Spark instead, introduces Resilient Distributed Datasets (RDD) as an abstraction of MapReduce. An RDD is a read-only collection of objects partitioned across a set of machines that can be reproduced in case a partition is lost. Spark is more suitable for iterative jobs and interactive processing such as in querying consoles. Moreover, Spark comes with an *Accumulator* operator that can be aligned with the OCL closure operation.

We believe that providing a distributed OCL engine can alleviate the development of not only distributed MT, but also other model-based solution, such as distributed query engines. We also believe that it would answer to some open challenges in the MDE community such as **Streaming model transformations** and **Pipelining and scheduling distributed model transformations**.

8.3.4 On the application of our work outside MDE

Hereafter, we look at our work from a different angle. We introduce another distributed programming model, Pregel, then we highlight its difference with MapReduce and its application to graph transformation. Then we shortly give a possible extent to our work outside the MDE community.

Other implicit distribution programming models

Despite our focus on the MapReduce model, there are other distribution frameworks, even specifically for graph processing. Most of the graph processing models are inspired by the BSP (Bulk Synchronous Processing) model. The computational model consists of a sequence of computations (called supersteps). In each step, a user-defined function is executed for each vertex. The function consumes messages from the previous superstep and produces messages to be sent to all vertices for the next superstep. A vertex deactivates itself when it has no further work to do, and it reactivates when receiving a new message. When all vertices are inactive, the computation terminates.

Pregel-like systems are well-suited to iterative algorithms where the number of iterations is not known on beforehand. In contrast to MapReduce, algorithms implemented in Pregel do not necessarily have a fixed number of supersteps. n MapReduce instead, the number of rounds is dictated by the number of `map` or `reduce` jobs being pipelined. However, in Pregel-like systems the supersteps can be identified by their number, which increments after every step. This feature can be used to branch on different operations when implementing the algorithm logic, and all vertices can vote to halt after a fixed number of iterations.

Our choice for MapReduce was motivated by the intuition that the MapReduce programming model could be efficiently aligned with the particular structure of relational (non-recursive) transformation languages. Its API hooks and the wide range of input/output format it supports could also reduce cross-machines communication cost. On the other hand, Pregel-like systems have been a good fit for alignment with graph transformation semantics [132, 84] with recursive rule applications.

ATL as high-level language for data transformation on MapReduce

The interest of this work extends also outside the MDE community, as we perform the first steps for proposing the rule-based MT paradigm as a high level abstraction for data transformation on MapReduce. High-level languages have already been proposed for data querying (as opposed to data transformation) on MapReduce [126, 108, 33]. In the MapReduce context, they allow for independent representation of queries w.r.t. the program logic, automatic query optimization and maximization of query reuse [87]. While all these query languages compile down to execution plans in the form of series of MapReduce jobs, in our approach each node executes its own instance of the transformation VM, reusing the standard engine. However our approach computes single transformations in only two MapReduce rounds, while these languages may compile in multi-round MapReduce chains. We also manipulate EMF model elements instead of tool-specific data representations,

hence leaning on a standardized way to represent the data structure. We want to extend this approach, aiming at obtaining similar benefits in data transformation scenarios.

Appendices

I

Model query and transformation benchmark repository

With the growing size and complexity of systems under design, industry needs a generation of Model-Driven Engineering (MDE) tools, especially model query and transformation, with the proven capability to handle large-scale scenarios. While researchers are proposing several technical solutions in this sense, the community lacks a set of shared scalability benchmarks, that would simplify quantitative assessment of advancements and enable cross-evaluation of different proposals. Benchmarks in previous work have been synthesized to stress specific features of model management, lacking both generality and industrial validity. In the context of the MONDO project, we initiate an effort to define a set of shared benchmarks, gathering queries and transformations from real-world MDE case studies, contributed to by academic in industrial partners. We make these case available to community evaluation via a public MDE benchmark repository.

In this chapter we present a set of benchmarks for model transformation and query engines. Our proposal is to select a set of transformations/queries from real-world cases and to make them available to the large public. Two of the four benchmarks included, deal with model transformations, while the other two deal with model queries. Models that feed each one of the benchmarks are of increasing size, also different kinds/extensions. They are either concrete, coming from real projects (i. e., reverse engineered Java project models) or generated using deterministic model instantiators. These instantiators can be easily customized to be able to generate models that suit the benchmarking of a specific feature.

These benchmarks are part of the results of the Scalable Modeling and Model Management on the Cloud (MONDO) [1]research project, that aims at advancing MDE's state of the art to tackle very large models. We plan to involve the community in order to build a larger set of case studies covering additional properties/domains (i.e., verification and simulation of critical systems).

1. http://www.mondo-project.org/

The rest of the Chaper is organized as follows. In Section I.1 we describe benchmarks from the state-of-the-art. In Section I.2 we outline the four different sets of benchmarks provided in the scope of the MONDO project. Finally, Section I.3 describes usage and contribution modalities for the repository.

I.1 Related work

A few works in literature [135, 136, 75, 22] proposed benchmarks to assist developers in selecting the most suitable query/transformation technology for their application scenarios. However only one of the existing case studies is explicitly dedicated to the manipulation of very large models ([75]) and none is based on benchmarks queries and transformations based on real-world applications. In this section we give a short overview of the related works, while in the next section we introduce the real-world benchmarks we proposed.

One of the widely used benchmarks in MDE is Grabats'09 Reverse Engineering [75], where Jouault et al. proposed a case study that consists of the definition of program comprehension operators (i.e., queries over source code) using a graph or model transformation tool. One of the main challenges in the definition of program comprehension operators as transformations is scalability with respect to input size. This case study is divided into two independent tasks (i) a simple filtering query that selects a sub-graph of its input according to a given condition, and (ii) a complex query that computes a control flow graph and a program dependence graph. These queries are performed over the JDTAST metamodel, the Java metamodel used in early versions of MoDisco [27] to discover Java projects. This benchmark comes with 5 different sets of increasing size ranging from 7×10^5 up to 5×10^9.

The experiment in [135] compares the performance of three model transformation engines: ATL, QVT-R, and QVT-O. This comparison is based on two different transformation examples, targeting meta-models with different structural representations: linear representation and tree-like representation. The benchmarking set involves randomly generated input models of increasing numbers of elements (up to hundreds of thousands). Like the previous work [136], the benchmark sets are also tuned according to a particular feature such as the size of input models, their complexity (complex interconnection structure) and transformation strategies. In order to study the impact of different implementation strategies in ATL, the Class2RDBMS transformation was implemented in different programming styles. The first one promotes expressing input models navigation in the in the right-hand side of the bindings, the second use ATL attribute helpers, and third uses the imperative part of ATL.

The work [136] is considered one of the early systematic MDE benchmarks dedicated to Graph Transformations (GT). It proposes a methodology for quantitative benchmarking in order to assess the performance of GT tools. The overall aim is to uniformly measure the performance of a system under a deterministic, parametric, and especially reproducible environment. Four tools participated in the experimentation: AGG, PROGRES, FUJABA and DB. Every benchmarking set is tuned according to some features related on one side to the graph transformation paradigms, and on the other side to the surveyed tools. Thus, a benchmarking set is characterized by turning on/off these features. Bergmann et al. extended this benchmark with incrementality. Two kinds of benchmarks kind were carried

Table I.1 – Summary of the MONDO WP3 benchmarks

Benchmark	**Type**	**Computational complexity**
Train benchmark	query	high
Open-BIM	query/transformation	low
ITM Factory	transformation	high
Transformation zoo	transformation	low

out, simulation and synchronization, for which, a benchmark-specific generators has been provided. The benchmarks were run over different implementations of pattern matchers, VIATRA/LS (Local Search), VIATRA/RETE, and GEN.NET with the distributed mutual exclusion algorithm.

I.2 Benchmark set

The benchmark set has the purpose to evaluate and validate a proposed query and transformation engine. This set of benchmarks is made public, with the intention to also help both research groups and companies to assess their solutions.

In this section we describe the source, context and properties of the benchmarks. The set of benchmarks is designed to cover the main use cases for queries and transformations in model-driven applications. Table I.1 summarizes the characteristics of the four benchmarks in terms of type of computation (query/transformation) and computational complexity (high/low). More information about the train benchmark can be found in the link below [2].

Each one of the benchmarks either includes concrete source models, or a model generator that can be used to produce models of different sizes in a deterministic manner. In the latter case, models of different sizes can be generated, but seeds are provided to drive the deterministic generators in producing the same models for each user.

Each benchmark in the set is given a reference implementation that has to be considered as a specification of the case semantics. Languages and technologies used for each reference implementation may vary, including MDE-specific and general-purpose technologies.

Finally, while each benchmark defines the source/target relation for each query or transformation, other aspects of the transformation runtime semantics are left open. For instance high-complexity benchmarks can be run in batch or incremental mode, to test different execution properties of the tool under study.

2. http://trainbenchmark.inf.mit.bme.hu/

I.2.1 Open-BIM

Context and objectives

This benchmark includes a model validation and a model transformation in the context of construction models. The construction industry has traditionally communicated building construction information (sites, buildings, floors, spaces, and equipment and their attributes) through drawings with notes and specifications. BIM (Building Information Model), a CAD (Computer Aided Design) method, came to automate that process and enhance its operability according to different tools, actors, etc. within the AECO (Architecture, Engineering, Construction, and Operations) industry. A BIM model is a multi-disciplinary data specific model instance which describes all the information pertinent to a building and its components.

A BIM model is described using the IFC (Industry Foundation Classes) specification, a freely available format to describe, exchange, and share information typically used within the building and facility management industry sector. Intrinsically, IFC is expressed using the EXPRESS data definition language, defined as ISO10303-11 by the ISO TC184/SC4 committee. EXPRESS representations are known to be compact and well suited to include data validation rules within the data specification.

Models and metamodel

The repository contains 8 real-world IFC data files with size ranging from 40MB to 1GB. All the files represent real construction projects and were used in production context. They contain a precise and detailed information about actors, approvals, buildings etc. The data files, in EXPRESS format, are translated into EMF models so that they can be used by EMF-based query and transformation tools.

Queries and transformations

The Open-BIM use case includes a query benchmark and a transformation benchmark:

IFC well-formedness rules The IFC format describes, using the EXPRESS language, the set of well-formed IFC models. The EXPRESS notation includes, in a single specification, 1) the set of element types and properties allowed in the data file, 2) the set of well-formedness constraints that have to be globally satisfied. When representing an IFC model in an EMF format these two parts of the specification translate to 1) an Ecore metamodel defining element and property types, 2) a set of constraints encoding the well-formedness rules.

This benchmark involves validating the set of well-formedness rules (2) over a given model, model that conforms to the IFC Ecore metamodel (1). An Ecore metamodel is provided, coming from the open-source BIMServer [3] project. The well-formedness rules are given in EXPRESS format and are meant to be translated to the query technology under evaluation.

3. https://github.com/opensourceBIM/BIMserver

IFC2BIMXML BIMXML [4] is an XML format describing building data in a simplified spatial building model. The BIMXML XML Schema was developed as an alternative to full scale IFC models to simplify data exchanges between various AEC applications and to connect Building Information Models through Web Services. It is currently used by several primary actors in the CAD construction domain, including Onuma System (Onuma, Inc.), DDS Viewer (Data Design System), vROC, Tokmo, BIM Connect, and various plugins for CAD Applications (Revit, SketchUp, ArchiCAD). The BIMXML specification includes an XML Schema [5] and documents the translation rules from the full IFC specification.

This benchmark involves performing the translation of a full IFC model into the BIMXML format. Ecore metamodels for the source and target models are provided.

I.2.2 ITM Factory

Context and objectives

This benchmark contains two transformations and a set of queries, each addressing a different phase in a model-driven reverse engineering process. The use case for this benchmark is taken from the Eclipse MoDisco project.

MoDisco (Model Discovery) is the open-source Model Driven Reverse Engineering project lead by the company Soft-Maint. It uses a two steps approach with a **model discovery** followed by **model understanding**. The initial step consists in obtaining a model representation of a specific view on the legacy system, whereas, the second involves extracting useful information from the discovered model. MoDisco requires high efficiency in handling large models, especially these involved in reverse engineering of legacy systems.

Models and metamodel

Thanks to the MoDisco Java discoverer, we are able to extract Java models up to 1.3GB, that conform to the Java metamodel [96] defined in MoDisco (refinement of the JDTAST metamodel). Those models are the input of the Java2KDM and Java code quality transformations, while, KDM output models are inputs for the KDM2UML transformation. It is also possible to retrieve directly KDM models using MoDisco. Because of confidentiality agreements, Soft-Maint is not able to publish instance models derived from their commercial projects. For this reason we choose to derive instance models from the source code of open-source projects, specifically from the Eclipse JDT plugins (org.eclipse.jdt.*). This does not affect the relevance of the benchmark, as these plugins are written by experienced developers with a quality standard that is comparable to commercial projects.

Table I.2 depicts the different models recovered against the discovered plugins.

4. http://bimxml.org/
5. http://www.bimxml.org/xsd/001/bimxml-001.xsd

Set1	org.eclipse.jdt.apt.pluggable.core
Set2	Set1 + org.eclipse.jdt.apt.pluggable.core
Set3	Set2 + org.eclipse.jdt.core+org.eclipse.jdt.compiler + org.eclipse.jdt.apt.core
Set4	Set3 + org.eclipse.jdt.core.manipulation + org.eclipse.jdt.launching + org.eclipse.jdt.ui + org.eclipse.jdt.debug
Set5	org.eclipse.jdt.* (all jdt plugins)

Table I.2 – Discovered plugins per set

Queries and transformations

Java2KDM This transformation takes place at beginning of almost every modernization process of a Java legacy system, it comes just after the discovery of the Java model from Java projects (plugins) using the MoDisco Java Discoverer. This transformation generates a KDM [109] (Knowledge Discovery Metamodel) model that defines common metadata required for deep semantic integration of application life-cycle management tools. Java2KDM transformation is useful when the only available information on a Java source code is contained in a Java model, even without the source code it is then possible to get a KDM representation. This intermediate model provides useful and precise information that can be used to produce additional types of models.

KDM2UML Based on the previously generated model, this transformation generates a UML diagram in order to allow integrating KDM-compliant tools (i.e., discoverers) with UML-compliant tools (e.g. modelers, model transformation tools, code generators, etc.).

Java code quality This set of code quality transformations identify well-known anti-patterns in Java source code and fix the corresponding issues by a model transformation. The input format of the transformations is a model conforming to the Java metamodel. For a specification for the transformations we refer the reader to the implementations of these fixes in well-known code-analysis tools like CheckStyle and PMD. Table 3 summarizes the references to the documentation for each code fix considered in this benchmark.

I.2.3 ATL Transformation Zoo

Context and objectives

The ATL project maintains a repository of ATL transformations produced in industrial and academic contexts (ATL Transformation Zoo [70]). These transformations are representative of the use of model transformations for low-complexity tasks (i.e., low number of transformation rules, lack of recursion, etc. . .).

In this benchmark we select a subset of the transformations in the ATL Transformation Zoo based on their quality level and usage in real-world applications. We specifically include only transformations that may be used in production environments. We automatize the sequential execution of this subset and the generation of performance analysis data.

Models and metamodels

For the aforementioned transformations, we do not have large enough models that conform to the respective metamodels, and as such we make use of a probabilistic model instantiator. This instantiator takes as parameter a generation configuration specified by the user. A generation configuration holds information such as 1) meta-classes that should (not) be involved in the generation, 2) probability distributions to establish how many instances should be generated for each metaclass, and which values should be assigned to structural features. We provide a default generation configuration, using uniform probability distributions for each meta-class and structural feature. For some transformations we provide ad-hoc probability distributions, exploiting domain knowledge over the instances of the corresponding metamodel.

A generation configuration may come also with a seed that makes the generation deterministic and reproducible. For each one of the built-in generation configurations we provide a seed, producing the exact set of models we used during our experimentation.

Queries and transformations

Ant to Maven Ant [5] is an open source build tool (a tool dedicated to the assembly of the different pieces of a program) from the Apache Software Foundation. Ant is the most commonly used build tool for Java programs. Maven [6] is another build tool created by the Apache Software Foundation. It is an extension of Ant because ant Tasks can be used in Maven. The difference from Ant is that a project can be reusable. This transformation [64] generates a file for the build tool Maven starting from a file corresponding to the build tool Ant.

CPL2SPL CPL (Call Processing Language) is a standard scripting language for the SIP (Session Initiation Protocol) protocol. It offers a limited set of language constructs. CPL is supposed to be simple enough so that it is safe to execute untrusted scripts on public servers [74]. SPL programs are used to control telephony agents (e.g. clients, proxies) implementing the SIP (Session Initiation Protocol) protocol. Whereas, the CPL has an XML-based syntax, the `CPL2SPL` transformation [66], provides an implementation of CPL semantics by translating CPL concepts into their SPL equivalent concepts.

Graphcet2PetriNet This transformation[67] establishes a link between grafcet [43], and petri nets [102]. It provides an overview of the whole transformation sequence that enables to produce an XML petri net representation from a textual definition of a grafcet in a PNML format, and the other way around.

IEEE1471 to MoDAF This transformation example [2] realizes the transformation between IEEE1471-2000 [77] and MoDAF-AV [25]. The IEEE1471 committee prescribes a recommended practice for the design and the analysis of Architecture of Software Intensive Systems. It fixes a terminology for System, Architecture, Architectural Description, Stakeholder, Concerns, View ,and Viewpoints concepts. MoDAF (Ministry of Defense Architecture Framework) is based on the DoDAF (Department of Defense Architecture

Framework). DoDAF is a framework to design C4ISR systems. MoDAF-AV (Architecture View) used several concepts defined in the IEEE1471.

Make2Ant Make (the most common build tool) is based on a particular shell or command interface and is therefore limited to the type of operating systems that use that shell. Ant uses Java classes rather than shell-based commands. Developers use XML to describe the modules in their program build. This benchmark [68] describes a transformation from a Makefile to an Ant file.

MOF2UML The MOF (Meta Object Facility)[110] is an OMG standard enabling the definition of metamodels through common semantics. The UML (Unified Modeling Language) Core standard is the OMG common modeling language. Although, MOF is primarily designed for metamodel definitions and UML Core for the design of models, the two standards define very close notions. This example [69] describes a transformation enabling to pass from the MOF to the UML semantics. The transformation is based on the UML Profile for MOF OMG specification.

OCL2R2ML The OCL to R2ML transformation scenario [98] describes a transformation from OCL (Object Constraint Language) [105] metamodel (with EMOF metamodel) into a R2ML (REWERSE I1 Rule Markup Language) metamodel. The Object Constraint Language (OCL) is a language that enables one to describe expressions and constraints on object-oriented (UML and MOF) models and other object modeling artifacts. An expression is an indication or specification of a value. A constraint is a restriction on one or more values of (part of) an object-oriented model or system. REWERSE I1 Rule Markup Language (R2ML) is a general web rule markup language, which can represent different rule types: integrity, reaction, derivation and production. It is used as pivotal metamodel to enable sharing rules between different rule languages, in this case with the OCL.

UML2OWL This scenario [99] presents an implementation of the OMG's ODM specification. This transformation is used to produce an OWL ontology, and its `OWL Individuals` from an UML Model, and its `UML Instances`.

BibTeXML to DocBook The `BibTeXML to DocBook` example [65] describes a transformation of a BibTeXML [113] model to a DocBook [138] composed document. BibTeXML is an XML-based format for the BibTeX bibliographic tool. DocBook, as for it, is an XML-based format for document composition.

DSL to EMF This example [9] provides a complete overview of a transformation chain example between two technical spaces: Microsoft DSL Tools [97] and EMF. The aim of this example is to demonstrate the possibility to exchange models defined under different technologies. In particular, the described bridges demonstrate that it should be possible to define metamodels and models using both Microsoft DSL Tools and Eclipse EMF technologies. The bridge between MS/DSL and EMF spans two levels: the metamodel and model levels. At the level of metamodels, it allows to transform MS/DSL domain models

to EMF metamodels. At the level of models, the bridge allows transforming MS/DSL models conforming to domain models to EMF models conforming to EMF metamodels. At both levels, the bridge operates in both directions. A chain of ATL-based transformations is used to implement the bridge at these two levels. The benefit of using such a bridge is the ability to transpose MS/DSL work in EMF platform, and inversely.

I.3 The MDE benchmark repository

The `MDE Benchmark repository` is the central storage area where the artifacts of the benchmarks are archived for public access. These artifacts, mainly text files, comprise large models and metamodels – typically represented in their XMI serialization – and model transformations. To increase the visibility of these files we have chosen to make them publicly available through the OpenSourceProjects.eu (OSP) platform. The OSP platform is a software forge dedicated to hosting Open Source projects created within EU research projects.

The OSP platform provides, among other tools, a Git revision control system (RCS). Git repositories hosted in the OSP platform can be easily navigated by a Web interface.

I.3.1 Benchmark structure

The `MDE Benchmark repository` is located at [114]. Inside this repository every top level resource corresponds to a git submodule, each, representing a different case study held in a separate git repository.

Related resources for benchmarking a specific feature of a transformation engine are grouped in `projects`. A `project` is a self-contained entity, and can be considered as the basic benchmarking unit.

`Projects` share a common internal structure that includes a short case description and a set of (optional) folders:

Short case description — A mandatory human-readable file describes the details of the test case, the file and directory structure, and any other important information (e.g., test cases can evolve and additional information not considered at the point of writing this document may be needed for executing the benchmark).

Documentation — This directory stores the documentation about the test case. The documentation of a test case may include, among other information, a detailed description of the test case, the foundations of the feature under testing, the building and execution instructions, etc.

Queries and Transformations — This directory stores the queries and transformations, in source code form, that stress the feature under testing.

Models — This directory contains the model and metamodel descriptions involved in the test transformation(s).

Input data — This directory contains the input data to be used by the test case(s).

Expected data — In this directory we store the files that contain the expected values that must be returned by the transformation. The expected data are compared with

the actual output of the transformation to determine if the test execution has been successful or not.

Source code — In some situations, test cases may require additional code (such as Java code) to be executed. For example, test cases may be automatically launched with the help of third party libraries (such as JUnit), or test cases may execute external code following a black-box scheme. In this situations the additional code should be placed inside the `/src` directory.

Libraries — This directory is used to store any additional third party library (usually a binary file) required by the test case.

Scripts — Build and execution scripts should be placed under the `/build` directory. Examples of such scripts are Ant files [5], Maven files [6], Makefiles [57], bash shell scripts [56].

I.3.2 Submission guidelines

In order to increase the quality and soundness of the test cases available in the `MDE Benchmark repository`, we plan to keep it open to further submissions from the MDE community.

We have defined a simple set of guidelines that must be followed when contributing a new case study to guarantee that the quality of the repository is maintained. Specifically:

— New contributions must include a comprehensive description of the case study. A rationale for its inclusion must be provided, specially focusing on the differential aspects of the proposed case study, compared to the already included benchmarks.

— The sources, models, documentation and utility scripts must be organized as described in Section I.3.1.

— Contributions must be sent to the address `mondo_team@opengroup.org` for their evaluation and approval.

Table 3: List of Java code quality fixes

Rule	Documentation
ConstantName	http://checkstyle.sourceforge.net/config_naming.html#ConstantName
LocalFinalVariableName	http://checkstyle.sourceforge.net/config_naming.html#LocalFinalVariableName
LocalVariableName	http://checkstyle.sourceforge.net/config_naming.html#LocalVariableName
MemberName	http://checkstyle.sourceforge.net/config_naming.html#MemberName
MethodName	http://checkstyle.sourceforge.net/config_naming.html#MethodName
PackageName	http://checkstyle.sourceforge.net/config_naming.html#PackageName
ParameterName	http://checkstyle.sourceforge.net/config_naming.html#ParameterName
StaticVariableName	http://checkstyle.sourceforge.net/config_naming.html#StaticVariableName
TypeName	http://checkstyle.sourceforge.net/config_naming.html#TypeName
AvoidStarImport	http://checkstyle.sourceforge.net/config_imports.html#AvoidStarImport
UnusedImports	http://checkstyle.sourceforge.net/config_imports.html#UnusedImports
RedundantImport	http://checkstyle.sourceforge.net/config_imports.html#RedundantImport
ParameterNumber	http://checkstyle.sourceforge.net/config_sizes.html#ParameterNumber
ModifierOrder	http://checkstyle.sourceforge.net/config_modifier.html#ModifierOrder
RedundantModifier	http://checkstyle.sourceforge.net/config_modifier.html#RedundantModifier
AvoidInlineConditionals	http://checkstyle.sourceforge.net/config_coding.html#AvoidInlineConditionals
EqualsHashCode	http://checkstyle.sourceforge.net/config_coding.html#EqualsHashCode
HiddenField	http://checkstyle.sourceforge.net/config_coding.html#HiddenField
MissingSwitchDefault	http://checkstyle.sourceforge.net/config_coding.html#MissingSwitchDefault
RedundantThrows	http://checkstyle.sourceforge.net/config_coding.html#RedundantThrows
SimplifyBooleanExpression	http://checkstyle.sourceforge.net/config_coding.html#SimplifyBooleanExpression
SimplifyBooleanReturn	http://checkstyle.sourceforge.net/config_coding.html#SimplifyBooleanReturn
FinalClass	http://checkstyle.sourceforge.net/config_design.html#FinalClass
InterfaceIsType	http://checkstyle.sourceforge.net/config_design.html#InterfaceIsType
VisibilityModifier	http://checkstyle.sourceforge.net/config_design.html#VisibilityModifier
FinalParameters	http://checkstyle.sourceforge.net/config_misc.html#FinalParameters
LooseCoupling	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/typeresolution.html#LooseCoupling
SignatureDeclareThrowsException	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/typeresolution.html#SignatureDeclareThrowsException
DefaultLabelNotLastInSwitchStmt	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/design.html#DefaultLabelNotLastInSwitchStmt
EqualsNull	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/design.html#EqualsNull
CompareObjectsWithEquals	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/design.html#CompareObjectsWithEquals
PositionLiteralsFirstInComparisons	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/design.html#PositionLiteralsFirstInComparisons
UseEqualsToCompareStrings	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/strings.html#UseEqualsToCompareStrings
IntegerInstantiation	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#IntegerInstantiation
ByteInstantiation	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#ByteInstantiation
ShortInstantiation	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#ShortInstantiation
LongInstantiation	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#LongInstantiation
BooleanInstantiation	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#BooleanInstantiation
SimplifyStartsWith	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/optimizations.html#SimplifyStartsWith
UnnecessaryReturn	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/unnecessary.html#UnnecessaryReturn
UnconditionalIfStatement	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/basic.html#UnconditionalIfStatement
UnnecessaryFinalModifier	http://pmd.sourceforge.net/pmd-5.1.0/rules/java/unnecessary.html#UnnecessaryFinalModifier

List of Terms

ACID Atomicity, Consistency, Isolation, Durability. 8, 24, 30, 37, 43–47, 69, 75, 83, 84, 102, 120

Application Master Is the entity responsible for negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the containers and their resource consumption . 81, 120

ATL AtlanMod Transformation Language. 5, 11, 36, 120

CAP Consistency, Availability and Partitioning tolerance. 43, 44, 120

CAS Compare And Set. 84, 85, 120

EMF Eclipse Modeling Framework. 3, 5, 92, 103, 120

ETL Epsilon Transformation Language. 30, 36, 104, 120

GPL General-Purpose Language. 4, 5, 13, 29, 120

HDFS Hadoop Distributed File System. 23, 78, 80, 83, 91, 120

MDE Model-Driven Engineering. 3–5, 7, 11–13, 16, 26, 27, 52, 65, 66, 70, 71, 100, 101, 105, 109–111, 118, 120, 123, 127

MQT Model Query and Transformation. 4, 6, 30, 120

Node Manager A per-node agent, and takes care of the individual compute nodesin a Hadoop cluster. 81, 92, 120

OCL Object Constraint Language. 13, 18, 19, 32, 39, 41, 45, 53, 54, 59–61, 74, 81, 85, 95, 104, 116, 120, 123

QVT Query/View/Transformation. 5, 30, 104, 120

RDD Resilient Distributed Datasets. 104, 120

Resource Manager Is a pure scheduler that is strictly limited to arbitrating available resources in the system among the competing applications . 81, 120

UML Unified Modeling Language. 3, 120

VLM Very Large Model. 6–8, 30, 64, 67, 101, 102, 120

XMI XML Metadata Interchange. 13, 74, 78, 88, 92, 120

Contents

List of Tables

List of Figures

Bibliography

[1] CDO Model Repository, 2014. 84

[2] Albin Jossic. ATL Transformation Example: IEEE1471 to MoDAF, 2005. URL: http://www.eclipse.org/atl/atlTransformations/IEEE1471_2_MoDAF/IEEE1471_2_MoDAF.doc. 127

[3] Amazon Web Services, Inc. Amazon EMR FAQs, May, 2016. URL: http://aws.amazon.com/elasticmapreduce/faqs. 98

[4] N. Amálio, J. de Lara, and E. Guerra. Fragmenta: A theory of fragmentation for MDE. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 106–115, Sept 2015. 76

[5] Apache. Apache ant, 2014. URL: http://ant.apache.org/. 127, 130

[6] Apache. Apache maven project, 2014. URL: http://maven.apache.org/. 127, 130

[7] Apache Software Foundation. Apache Hadoop, May, 2016. URL: http://hadoop.apache.org/. 33, 34, 87

[8] Apache Software Foundation. Apache Hadoop Distributed File System (HDFS), May, 2016. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. 33

[9] ATLAS group – LINA & INRIA. The Microsoft DSL to EMF ATL transformation , 2005. URL: http://www.eclipse.org/atl/atlTransformations/DSL2EMF/ExampleDSL2EMF%5Bv00.01%5D.pdf. 128

[10] Aurelius. TitanDB, January, 2016. URL: http://thinkaurelius.github.io/titan/. 84

[11] P. Baker, S. Loh, and F. Weil. Model-driven Engineering in a Large Industrial Context: Motorola Case Study. In *Model Driven Engineering Languages and Systems*, pages 476–491. Springer, 2005. 14

[12] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: Great. *Electronic Communications of the EASST*, 1, 2007. 24

[13] K. Barmpis and D. Kolovos. Hawk: Towards a Scalable Model Indexing Architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 6. ACM, 2013. 42, 61

[14] K. Barmpis and D. S. Kolovos. Comparative analysis of data persistence technologies for large-scale models. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pages 33–38, New York, NY, USA, 2012. ACM. 77

[15] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4EMF, A Scalable Persistence Layer for EMF Models. In *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2014. 18, 78, 97

[16] A. Benelallam, A. Gómez, and M. Tisi. Atl-mr: model transformation on mapreduce. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems*, pages 45–49. ACM, 2015. 18, 85

[17] A. Benelallam, A. Gómez, M. Tisi, and J. Cabot. Distributed Model-to-model Transformation with ATL on MapReduce. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 37–48, New York, NY, USA, 2015. ACM. 18, 74, 97

[18] A. Benelallam, M. Tisi, I. Rath, B. Izso, and D. S. Kolovos. Towards an Open Set of Real-World Benchmarks for Model Queries and Transformations. In C. W. Proceedings, editor, *BigMDE*, York, UK, United Kingdom, 2014. University of York. 97

[19] A. Benelallam, M. Tisi, J. Sánchez Cuadrado, J. de Lara, and J. Cabot. Efficient Model Partitioning for Distributed Model Transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, New York, NY, USA, 2016. ACM. 18, 97

[20] G. Bergmann, A. Hegedüs, and A. Horváth. Integrating efficient model queries in state-of-the-art EMF tools. *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 2012. 42

[21] G. Bergmann, A. Horváth, and I. Ráth. Incremental evaluation of model queries over EMF models. *International Conference on Model Driven Engineering Languages and Systems*, 2010. 42

[22] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In *Graph Transformations*, pages 396–410. Springer, 2008. 122

[23] G. Bergmann, I. Ráth, and D. Varró. Parallelization of graph transformation based on incremental pattern matching. *Electronic Communications of the EASST*, 18, 2009. 41

[24] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013. 75

[25] B. Biggs. Ministry Of Defence Architectural Framework (MODAF). *IET*, 2005. 127

[26] E. Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012. 53

[27] H. Bruneliére, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A Model Driven Reverse Engineering Framework. *Information and Software Technology*, 56(8):1012–1032, 2014. 13, 75, 97, 109, 122

[28] Bryan Hunt. MongoEMF, 2014. URL: https://github.com/BryanHunt/mongo-emf/. 77, 84

[29] L. Burgueño, E. Syriani, M. Wimmer, J. Gray, and A. Moreno Vallecillo. LinTraP: Primitive Operators for the Execution of Model Transformations with LinTra. In *Proceedings of 2nd BigMDE Workshop*, volume 1206. CEUR Workshop Proceedings, 2014. 41

[30] L. Burgueno, J. Troya, M. Wimmer, and A. Vallecillo. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, 2015. 62

[31] L. Burgueño, M. Wimmer, and A. Vallecillo. A linda-based platform for the parallel execution of out-place model transformations. *Information & Software Technology*, 79:17–35, 2016. 41, 43, 44, 112

[32] R. Buyya. High performance cluster computing: programming and applications, vol. 2. *Pre ticeHallPTR, NJ*, 1999. 29

[33] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008. 43, 117

[34] 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 (TOCS)*, 26(2):4, 2008. 34, 35, 143

[35] Y. Chawla and M. Bhonsle. A Study on Scheduling Methods in Cloud Computing. *International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)*, 1(3):12–17, 2012. 59

[36] F. Chen, M. Kodialam, and T. Lakshman. Joint Scheduling of Processing and Shuffle Phases in MapReduce Systems. In *Proceedings of The 31rd Annual IEEE International Conference on Computer Communications*, pages 1143–1151, March 2012. 60, 61

[37] C. Clasen, M. Didonet Del Fabro, and M. Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. In *First International Workshop on Model-Driven Engineering on and for the Cloud*, Copenhagen, Denmark, 2012. Springer. 44

[38] 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 *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 267–270. IEEE, 2002. 23, 24

[39] J. S. Cuadrado, E. Guerra, and J. d. Lara. Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 34–44, Nov 2014. 62, 74, 91

[40] 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. 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 2016), Saint-Malo, France, October 2-7, 2016.*, pages 1–7, 2016. Available Online at `http://tinyurl.com/jhkqoyx`. 88

[41] G. Daniel, G. Sunyé, and J. Cabot. Mogwaï: a Framework to Handle Complex Queries on Large Models. In *Proc of the 10th RCIS Conference (to appear)*. IEEE, 2016. Available Online at `http://tinyurl.com/jgopmvk`. 84, 112

[42] G. Daniel, G. Sunyé, and J. Cabot. PrefetchML: a Framework for Prefetching and Caching Models. In *MODELS 2016*, Saint-Malo, France, Oct. 2016. 110

[43] R. David. Grafcet: A powerful tool for specification of logic controllers. *Control Systems Technology, IEEE Transactions on*, 3(3):253–268, 1995. 127

[44] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Commun. ACM*, volume 51, pages 107–113, NY, USA, 2008. ACM. 15, 31, 32

[45] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *ACM Communication*, volume 51, pages 107–113, NY, USA, 2008. ACM. 59

[46] B. Dominic. Towards Model-Driven Engineering for Big Data Analytics – An Exploratory Analysis of Domain-Specific Languages for Machine Learning, Jan 2014. 14

[47] D. Durisic, M. Staron, M. Tichy, and J. Hansson. Evolution of long-term industrial meta-models – an automotive case study of autosar. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 141–148, Aug 2014. 15

[48] D. Durisic, M. Staron, M. Tichy, and J. Hansson. Quantifying long-term evolution of industrial meta-models-a case study. In *Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2014 Joint Conference of the International Workshop on*, pages 104–113. IEEE, 2014. 15

[49] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006. 40, 62

[50] M. R. Fellows, J. Guo, C. Komusiewicz, R. Niedermeier, and J. Uhlmann. Graph-based Data Clustering with Overlaps. *Discrete Optimization*, 8(1):2–17, 2011. 47

[51] A. Forward and T. C. Lethbridge. Problems and Opportunities for Model-centric Versus Code-centric Software Development: A Survey of Software Professionals. In *Proceedings of the 2008 International Workshop on Models in Software Engineering*, MiSE '08, pages 27–32, New York, NY, USA, 2008. ACM. 14

[52] A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara. EMF Splitter: A Structured Approach to EMF Modularity. *XM@ MoDELS, 3rd Extreme Modeling Workshop*, 1239:22–31, 2014. 76

[53] L. George, A. Wider, and M. Scheidgen. Type-safe model transformation languages as internal dsls in scala. In *International Conference on Theory and Practice of Model Transformations*, pages 160–175. Springer, 2012. 39

[54] H. Giese and S. Hildebrandt. *Efficient Model Synchronization of Large-scale Models*. Universitätsverlag Potsdam, 2009. 42

[55] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8(1):21–43, 2009. 42

[56] GNU. Bourne-Again SHell manual, 2014. URL: `http://www.gnu.org/software/bash/manual/`. 130

[57] GNU. GNU 'make', 2014. URL: `http://www.gnu.org/software/make/manual/`. 130

[58] A. Gómez, A. Benelallam, and M. Tisi. Decentralized Model Persistence for Distributed Computing. In *Proceedings of 3rd BigMDE Workshop*, volume 1406. CEUR Workshop Proceedings, July 2015. 18, 78, 108

[59] A. Gómez, M. Tisi, G. Sunyé, and J. Cabot. Map-based transparent persistence for very large models. In *Fundamental Approaches to Software Engineering (FASE)*, pages 19–34. Springer, 2015. 46, 77, 78, 82, 112

[60] V. Guana and E. Stroulia. Chaintracker, a model-transformation trace analysis tool for code-generation environments. In D. Di Ruscio and D. Varró, editors, *Theory and Practice of Model Transformations: 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, pages 146–153, Cham, 2014. Springer International Publishing. 62

[61] T. Horn. The TTC 2013 Flowgraphs Case. *arXiv preprint arXiv:1312.0341*, 2013. 26, 98

[62] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480. ACM, 2011. 14

[63] G. Imre and G. Mezei. Parallel Graph Transformations on Multicore Systems. In *Multicore Software Engineering, Performance, and Tools*, volume 7303 of *LNCS*, pages 86–89. Springer, 2012. 41, 44

[64] INRIA. ATL Transformation Example: Ant to Maven, 2005. URL: http://www.eclipse.org/atl/atlTransformations/Ant2Maven/ExampleAnt2Maven%5Bv00.01%5D.pdf. 127

[65] INRIA. ATL Transformation Example: BibTeXML to DocBook, 2005. URL: http://www.eclipse.org/atl/atlTransformations/BibTeXML2DocBook/ExampleBibTeXML2DocBook%5Bv00.01%5D.pdf. 128

[66] INRIA. ATL Transformation Example: CPL to SPL, 2005. URL: http://www.eclipse.org/atl/atlTransformations/CPL2SPL/README.txt. 127

[67] INRIA. ATL Transformation Example: Grafcet to Petri Net, 2005. URL: http://www.eclipse.org/atl/atlTransformations/Grafcet2PetriNet/ExampleGrafcet2PetriNet[v00.01].pdf. 127

[68] INRIA. ATL Transformation Example: Make to Ant, 2005. URL: http://www.eclipse.org/atl/atlTransformations/Make2Ant/ExampleMake2Ant[v00.01].pdf. 128

[69] INRIA. ATL Transformation Example: MOF to UML, 2005. URL: http://www.eclipse.org/atl/atlTransformations/MOF2UML/ExampleMOF2UML[v00.01].pdf. 128

[70] Inria. Atl transformation zoo, 2014. URL: http://www.eclipse.org/atl/atlTransformations/. 126

[71] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007. 43

[72] B. Izsó, G. Szárnyas, I. Ráth, and D. Varró. IncQuery-D Incremental Graph Search in the Cloud. In *Proceedings of the Workshop on Scalability in MDE*, BigMDE '13, pages 4:1–4:4, New York, NY, USA, 2013. ACM. 40, 44, 61

[73] C. Jeanneret, M. Glinz, and B. Baudry. Estimating Footprints of Model Operations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 601–610. ACM, 2011. 62

[74] F. Jouault, J. Bézivin, C. Consel, I. Kurtev, F. Latry, et al. Building DSLs with AMMA/ATL, a case study on SPL and CPL telephony languages. In *ECOOP Workshop on Domain-Specific Program Development*, pages 1–4, 2006. URL: https://hal.inria.fr/inria-00353580. 127

[75] F. Jouault, J. Sottet, et al. An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In *5th International Workshop on Graph-Based Tools, Zurich, Switzerland*, pages 123–137, 2009. 23, 75, 109, 122

[76] F. Jouault and M. Tisi. Towards Incremental Execution of ATL Transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 123–137. Springer, 2010. 42

[77] E. Jouenne and V. Normand. Tailoring ieee 1471 for mde support. In *UML Modeling Languages and Applications*, pages 163–174. Springer, 2005. 127

[78] S. Kent. Model driven engineering. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods: Third International Conference, IFM 2002 Turku, Finland, May 15–18, 2002 Proceedings*, pages 286–298, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. 21

[79] A. Khurana. Introduction to HBase Schema Design. *;login: The Usenix Magazine*, 37(5):29–36, 2012. 34

[80] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag. 23, 40

[81] D. S. Kolovos, R. F. Paige, and F. A. Polack. The grand Challenge of Scalability for Model-driven Engineering. In *Models in Software Engineering*, pages 48–53. Springer, 2009. 14

[82] 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, (STAF2015)*, pages 44–53, 2015. 84

[83] D. S. Kolovos, D. D. Ruscio, N. D. Matragkas, J. de Lara, I. Ráth, and M. Tisi, editors. *Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering co-located with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, July 24, 2014*, volume 1206 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014. 84

[84] C. Krause, M. Tichy, and H. Giese. Implementing graph transformations in the bulk synchronous parallel model. In *International Conference on Fundamental Approaches to Software Engineering*, pages 325–339. Springer, 2014. 40, 44, 112, 117

[85] I. Kurtev. Adaptability of model transformations. 2005. 22

[86] M. Laakso and A. Kiviniemi. The ifc standard: A review of history, sevelopment, and standardization, information technology. *ITcon*, 17(9):134–161, 2012. 15

[87] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel Data Processing with MapReduce: A Survey. In *SIGMOD Rec.*, volume 40, pages 11–20, New York, NY, USA, 2012. ACM. 117

[88] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM. 11, 89

[89] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, et al. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003. 31

[90] A. LUMSDAINE, D. GREGOR, B. HENDRICKSON, and J. BERRY. CHALLENGES IN PARALLEL GRAPH PROCESSING. *Parallel Processing Letters*, 17(01):5–20, 2007. 44, 57, 59

[91] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceeding of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, Indianapolis, Indiana, USA, 2010. ACM. 15, 31

[92] MapDB. MapDB, 2016. URL: www.mapdb.org. 85

[93] Markus Scheidgen. EMF fragments, 2014. URL: https://github.com/markus1978/emf-fragments/wiki/. 76

[94] Q. Medini. ikv++ technologies ag, germany. 23

[95] G. Mezei, T. Levendovszky, T. Meszaros, and I. Madari. Towards Truly Parallel Model Transformations: A Distributed Pattern Matching Approach. In *IEEE EUROCON 2009*, pages 403–410. IEEE, 2009. 40, 44

[96] MIA-Software. Modiscojava metamodel (knowledge discovery metamodel) version 1.3, 2012. URL: http://dev.eclipse.org/svnroot/modeling/org.eclipse.mdt.modisco/main/branches/0_11/org.eclipse.gmt.modisco.java/model/java.ecore. 125

[97] Microsoft Corp. The DSL tools, 2014. URL: http://msdn.microsoft.com/vstudio/DSLTools/. 128

[98] Milan Milanovic. ATL Transformation Example: OCL to R2ML, 2005. URL: http://www.eclipse.org/atl/atlTransformations/OCL2R2ML/README.txt. 128

[99] Milan Milanovic. ATL Transformation Example: UML to OWL, 2005. URL: http://www.eclipse.org/atl/atlTransformations/UML2OWL/README.txt. 128

[100] MongoDB Inc. EMF fragments, 2016. URL: https://www.mongodb.com/. 77

[101] J. M. Mottu, S. Sen, M. Tisi, and J. Cabot. Static Analysis of Model Transformations for Effective test Generation. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 291–300. IEEE, 2012. 62

[102] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. 127

[103] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. Acceleo user guide, 2006. 75

[104] U. Nickel, J. Niere, and A. Zündorf. The fujaba environment. In *Proceedings of the 22nd international conference on Software engineering*, pages 742–745. ACM, 2000. 24

[105] Object Management Group. Object Constraint Language, OCL, May, 2016. URL: http://www.omg.org/spec/OCL/. 23, 27, 128

[106] Object Management Group. Query/View/Transformation, QVT, May, 2016. URL: http://www.omg.org/spec/QVT/. 23, 40, 75

[107] Object Management Group. XML Metadata Interchange, May, 2016. URL: http://www.omg.org/spec/XMI/. 23, 88, 102

[108] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008. 43, 117

[109] OMG (Object Management Group). Kdm (knowledge discovery metamodel) version 1.3, 2011. URL: http://www.omg.org/spec/KDM/1.3/. 126

[110] OMG (Object Management Group). Mof (meta object facility) version 2.4, 2011. URL: http://www.omg.org/spec/MOF/2.4. 25, 128, 143

[111] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *14th International Conference on Model Driven Engineering Languages and Systems*, pages 77–92. Springer-Verlag, 2011. 76, 77

[112] J. E. Pagán and J. G. Molina. Querying large models efficiently. *Information and Software Technology*, 56(6):586 – 622, 2014. 76, 77

[113] L. Previtali, B. Lurati, and E. Wilde. Bibtexml: An xml representation of bibtex. In V. Y. Shen, N. Saito, M. R. Lyu, and M. E. Zurko, editors, *WWW Posters*, 2001. 128

[114] M. Project. Transformation benchmarks, 2014. URL: http://opensourceprojects.eu/p/mondo/d31-transformation-benchmarks/. 129

[115] I. Ráth, A. Hegedüs, and D. Varró. Derived features for EMF by integrating advanced model queries. *Modelling Foundations and Applications ECMFA*, 2012. 42

[116] T. Rauber and G. Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media, 2013. 31

[117] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases: New Opportunities for Connected Data*. " O'Reilly Media, Inc.", 2015. 36, 143

[118] C. Sakamoto, T. Miyazaki, M. Kuwayama, K. Saisho, and A. Fukuda. Design and implementation of a parallel pthread library (ppl) with parallelism and portability. *Systems and Computers in Japan*, 29(2):28–35, 1998. 31

[119] M. Scheidgen, A. Zubow, J. Fischer, and T. H. Kolbe. Automated and Transparent Model Fragmentation for Persisting Large Models. In *15th International Conference on Model Driven Engineering Languages and Systems*, pages 102–118. Springer-Verlag, 2012. 76

[120] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012. 59, 61, 71, 74

[121] M. Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010. 33

[122] G. Taentzer. Agg: A tool environment for algebraic graph transformation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 481–488. Springer, 1999. 24

[123] The Apache Software Foundation. Apache ZooKeeper, 2015. 93

[124] The Apache Software Foundation. Apache HBase, May, 2016. URL: http://hbase.apache.org/. 34, 85

[125] The Eclipse Foundation. ATL EMFTVM, May, 2016. URL: https://wiki.eclipse.org/ATL/EMFTVM/. 10, 89

[126] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009. 43, 117

[127] Tinkerpop. Blueprints API, 2016. URL: blueprints.tinkerpop.com. 84

[128] M. Tisi, S. Martínez, and H. Choura. Parallel Execution of ATL Transformation Rules. In *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 656–672. Springer, 2013. 41

[129] M. Tisi, S. Martinez, and H. Choura. Parallel execution of atl transformation rules. In *Model-Driven Engineering Languages and Systems*, pages 656–672. Springer, 2013. 43, 44

[130] M. Tisi, S. Martínez, F. Jouault, and J. Cabot. Lazy execution of model-to-model transformations. *International Conference on Model Driven Engineering Languages and Systems*, 2011. 42

[131] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342. ACM, 2014. 61

[132] L.-D. Tung and Z. Hu. Towards systematic parallelization of graph transformations over pregel. *International Journal of Parallel Programming*, pages 1–20, 2015. 40, 41, 44, 112, 117

[133] Z. Ujhelyi, G. Bergmann, and A. Hegedüs. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, 2014. 42

[134] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. 31

[135] M. Van Amstel, S. Bosems, I. Kurtev, and L. F. Pires. Performance in model transformations: experiments with atl and qvt. In *Theory and Practice of Model Transformations*, pages 198–212. Springer, 2011. 122

[136] G. Varro, A. Schurr, and D. Varro. Benchmarking for graph transformation. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 79–88. IEEE, 2005. 122

[137] R. Volk, J. Stengel, and F. Schultmann. Building Information Modeling (BIM) for Existing Buildings: Literature Review and Future Needs. *Automation in Construction*, 38(0):109–127, 2014. 13, 14

[138] N. Walsh and R. Hamilton. *DocBook 5: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2010. 128

[139] Y. Yao, J. Wang, B. Sheng, J. Lin, and N. Mi. HaSTE: Hadoop YARN Scheduling Based on Task-Dependency and Resource-Demand. In *Proceedings of 7th IEEE International Conference on Cloud Computing (CLOUD)*, pages 184–191, Jun 2014. 60

[140] A. Yie and D. Wagelaar. Advanced Traceability for ATL. In *Proceeding of the 1st International Workshop on Model Transformation with ATL (MtATL)*, pages 78–87, Nantes, France, 2009. 53

[141] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:1–10, 2010. 32

[142] S. E. Zenith. Linda Coordination Language; Subsystem Kernel Architecture (on Transputers). *RR-794, Yale University, New Haven, CT (May 1990)*, 1990. 31

[143] Y. Zhu, Y. Jiang, W. Wu, L. Ding, A. Teredesai, D. Li, and W. Lee. Minimizing Makespan and Total Completion Time in MapReduce-like Systems. In *Proceedings of The 33rd Annual IEEE International Conference on Computer Communications*, pages 2166–2174, Apr 2014. 60

UNIVERSITE BRETAGNE LOIRE

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Amine BENELALLAM

Transformation de modèles sur plates-formes réparties

Persistance décentralisée et traitement distribué

Model transformation on distributed platforms

decentralized persistence and distributed processing

Résumé

Grâce à sa promesse de réduire les efforts de développement et maintenance du logiciel, l'Ingénierie Dirigée par les Modèles (IDM) attire de plus en plus les acteurs industriels. En effet, elle a été adoptée avec succès dans plusieurs domaines tels que le génie civil, l'industrie automobile et la modernisation de logiciels. Toutefois, la taille croissante des modèles utilisés nécessite de concevoir des solutions passant à l'échelle afin de les traiter (transformer), et stocker (persister) de manière efficace. Une façon de pallier cette problématique est d'utiliser les systèmes et les bases de données répartis. D'une part, les paradigmes de programmation distribuée tels que MapReduce et Pregel peuvent simplifier la distribution de transformations des modèles (TM). Et d'autre part, l'avènement des base de données NoSQL permet le stockage efficace des modèles d'une manière distribuée. Dans le cadre de cette thèse, nous proposons une approche pour la transformation ainsi que pour la persistance de grands modèles. Nous nous basons d'un côté, sur le haut niveau d'abstraction fourni par les langages déclaratifs (relationnels) de transformation et d'un autre côté, sur la sémantique bien définie des paradigmes existants de programmation distribués, afin de livrer un moteur distribué de TM. La distribution est implicite et la syntaxe du langage n'est pas modifiée (aucune primitive de parallélisation n'est ajoutée). Nous étendons cette solution avec un algorithme efficace de distribution de modèles qui se base sur l'analyse statique des transformations et sur résultats récents sur le partitionnement équilibré des graphes. Nous avons appliqué notre approche à ATL, un langage relationnel de TM et MapReduce, un paradigme de programmation distribué. Finalement, nous proposons une solution pour stocker des modèles à l'aide de bases de données NoSQL, en particulier au travers d'un cadre d'applications de persistance répartie.

Mots clés

Ingénierie Dirigée par les Modèles, Transformation & Persistance de modèles, Distribution, ATL, MapReduce

Abstract

Model-Driven Engineering (MDE) is gaining ground in industrial environments, thanks to its promise of lowering software development and maintenance effort. It has been adopted with success in producing software for several domains like civil engineering, car manufacturing and modernization of legacy software systems. As the models that need to be handled in model-driven engineering grow in scale, it became necessary to design scalable algorithms for model transformation (MT) as well as well-suitable persistence frameworks. One way to cope with these issues is to exploit the wide availability of distributed clusters in the Cloud for the distributed execution of model transformations and their persistence. On one hand, programming models such as MapReduce and Pregel may simplify the development of distributed model transformations. On the other hand, the availability of different categories of NoSQL databases may help to store efficiently the models. However, because of the dense interconnectivity of models and the complexity of transformation logics, scalability in distributed model processing is challenging. In this thesis, we propose our approach for scalable model transformation and persistence. We exploit the high-level of abstraction of relational MT languages and the well-defined semantics of existing distributed programming models to provide a relational model transformation engine with implicit distributed execution. The syntax of the MT language is not modified and no primitive for distribution is added. Hence developers are not required to have any acquaintance with distributed programming. We extend this approach with an efficient model distribution algorithm, based on the analysis of relational model transformation and recent results on balanced partitioning of streaming graphs. We applied our approach to a popular MT language, ATL, on top of a well-known distributed programming model, MapReduce. Finally, we propose a multi-persistence backend for manipulating and storing models in NoSQL databases according to the modeling scenario. Especially, we focus on decentralized model persistence for distributed model transformations.

Key Words

Model Driven Engineering, Model Transformation & Persistence, Distribution, ATL, MapReduce