Contribution to the analysis of the design-space of a distributed transformation engine

Jolan Philippe

DOCTORAT BRETAGNE LOIRE / MATHSTIC

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

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS-DE-LA-LOIRE - IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Jolan PHILIPPE

Contribution to the Analysis of the Design-Space of a Distributed Transformation Engine

Thèse présentée et soutenue à IMT Atlantique à Nantes, le 19 Décembre 2022
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes
Thèse N° : 2022IMTA0339

Rapporteurs avant soutenance :

Jesus SANCHEZ CUADRADO	Associate professor, Universidad de Murcia, Spain
Matthias TICHY	Professor, University of Ulm, Germany

Composition du Jury :

Président :	Thomas LEDOUX	Professor, Institut Mines-Telecom Atlantique, France
Examinateurs :	Leen LAMBERS	Professor, Brandenburg University of Technology, Germany
	Jesus SANCHEZ CUADRADO	Associate professor, Universidad de Murcia, Spain
	Matthias TICHY	Professor, University of Ulm, Germany
	Antonio VALLECILLO	Professor, University of Málaga, Spain
Dir. de thèse :	Gerson SUNYE	Associate professor, University of Nantes, France
Co-encadrants de thèse :	Massimo TISI	Associate professor, Institut Mines-Telecom Atlantique, France
	Hélène COULLON	Associate professor, Institut Mines-Telecom Atlantique, France

ACKNOWLEDGEMENT

You can do anything
you set your mind to, man

Lose yourself
Eminem

Foremost, I would like to express my sincere gratitude to my advisors Dr. Hélène Coullon, Dr. Massimo Tisi, and Dr. Gerson Sunyé for these three last years of science. None of this journey would have been done without their support, their patience and their advice.

I would like to thank all the members of my jury: Dr. Jesus Sanchez Cuadrado, Pr. Matthias Tichy, Pr. Antonio Vallecillo, Pr. Leen Lambers, and Pr. Thomas Ledoux for reviewing my work, attending my defense, and for the interesting questions and discussions that followed my thesis defense.

I adress a big thank you to the all people I had the chance to work with for all our scientific collaboration. Thank you to Naomod and Stack teams, and all the Lowcomote contributors. I hope there will be future occasion to work together, or at least to meet around a drink. I also want to adress a special thank to my friends Rémi Douence and Théo Le Calvar.

To all my dear A246 office-mates: Thibault, Joachim, Ali and James, I address a big thank you. “Kudos to our stolen sofa”. I did not only shared an office, but also a place. Thanks to all the flatmates I had, who supported me at home.

What would be acknowledgement without thanking all those who really supported me during the writing period: Josselin, Mathieu, Julie, Spencer, Sirine, Hélène, Pierre, Colin. Thank you guys, all these beers helped a lot.

I would like to address a lovely thank you to my family. Words will never be enough to express my gratitude to you, my parents.

Before closing these acknowledgement, I would like to thank Louison Grouard, for her love and support during the final rush of the Ph.D journey. Even at the really end, I thought about quitting, but you kept my head clear, and focused.

And finally, to my two best friends, Anthony and Benoît, who repeated for the 10 last years that one day I will be doctor: an infinite amount of love. I dedicate this thesis to both of you. Thank you my boys.

À Anthony et Benoît

RÉSUMÉ FRANÇAIS

Contexte

L'Ingénierie logicielle basée sur les modèles est devenue une approche populaire. La modélisation est une activité présente dans tous les domaines scientifiques qui vise à donner une vision simplifiée des entités du monde réel telles que la biologie, les mathématiques, le génie civil, les lignes de produits ou les disciplines philosophiques [7, 8, 11, 68, 141]. Plus précisément, le Ingénierie dirigée par les models (IDM) est une approche d'ingénierie logicielle qui met en scène des modèles comme point central pour le développement de logiciels. En génie logiciel, les modèles sont utilisés pour décrire tous les aspects des logiciels et des systèmes, y compris leurs architectures, leurs actions, leurs composants physiques, leurs algorithmes ou leurs communications. Pour des raisons de standardisation, l'Object Management Group (OMG) a défini un langage standard de représentation des modèles nommé Unified Modeling Language (UML) [145]. Compte tenu des nombreux rôles des personnes impliquées dans la conception d'une application, avoir des modèles centraux partagés par les acteurs aide vraiment à la compréhension, et constitue une vue de connaissance forte. Bien sûr, il existe de nombreux types de modèles, avec des objectifs différents, et ils n'ont pas besoin du même niveau d'expertise. Mais les modèles sont souvent considérés comme un pont entre les acteurs autour du logiciel. Les modèles constituent désormais une partie vivante des projets et sont souvent manipulés comme des entités individuelles.

Les deux principales catégories d'outils de gestion de modèle que nous considérons sont la transformation de modèle et la requête de modèle. D'une part, la transformation de modèle est le processus de conversion d'un ou plusieurs modèles d'entrée en modèles de sortie (modèle-vers-modèle) ou en texte (modèle-vers-texte). Une transformation de modèle qui produit un modèle en sortie peut être soit une transformation sur place (c'est-à-dire une modification directe du modèle d'entrée) soit une transformation hors place (c'est-à-dire la production d'un nouveau modèle à partir de celui d'entrée). D'autre part, une requête de modèle analyse les modèles source pour calculer la valeur de données souhaitée. Dans la littérature, il existe une distinction claire entre ces deux opérations, même si les requêtes peuvent s'exprimer sous forme de transformations, et inversement. Nous parlons de

requête lorsqu'il s'agit d'expression atomique, utilisées pour obtenir un résultat unique, tandis qu'une transformation de modèle est l'exécution de règles de transformation, basées sur des requêtes, par un moteur dédié. Notez qu'il existe également des moteurs pour exécuter des requêtes sur les modèles.

Les modèles sont utilisés pour représenter des entités réelles, comme des données. Puisque nous vivons dans un monde entouré de capteurs, nous créons continuellement des informations, augmentant la taille des données produites. Le traitement de cette énorme quantité de données s'appelle Big Data. Dans MDE, nous appelons les modèles contenant beaucoup d'informations des "Very Large Models" (VLMs). Les outils dédiés au traitement des VLMs ont un besoin important d'opérations automatiques, transparentes, efficaces et évolutives, pour manipuler, interroger et analyser des modèles. La plupart des opérations de gestion de modèle sont exécutées au moment de la conception, par exemple pour éditer, valider, transformer le modèle. Le temps nécessaire pour répondre à une commande graphique est un facteur de qualité d'un outil MDE et influe sur le confort du développeur.

Énoncé du problème

L'IDM est une méthode puissante pour organiser et traiter théoriquement tout type de données. Les VLMs posent des défis supplémentaires, en raison de la taille des données qu'ils représentent. Un problème de mise à l'echelle se pose lorsqu'un outil doit manipuler de grands modèles d'instance de données, par exemple, comme cela se produit aujourd'hui dans plusieurs domaines d'ingénierie (automobile, aéronautique, civil). En raison de la taille même du modèle, il est nécessaire de fournir une solution efficace. Pour améliorer l'efficacité et la mise à l'echelle des solutions d'IDM, des recherches récentes sur la gestion de modèles ont étudié la programmation parallèle et concurrente ainsi que des modèles d'exécution spécifiques pour les langages de gestion de modèles. Ces techniques vont de la mise en œuvre d'algorithmes d'exécution spécifiques (par exemple, RETE [74]) à la compilation vers des modèles de programmation distribués (par exemple, MapReduce [65]). La diversité des stratégies employées pose plusieurs défis scientifiques.

La plupart des solutions ont été développées indépendamment, sur différentes technologies et avec des objectifs différents. Les nombreuses solutions n'étant pas formalisées au sein d'une solution unifiée, il n'est pas possible de faire un comparatif clair et précis de ces outils. L'introduction de solutions de parallélisme, qui sont des approches non déterministes, rend encore plus difficile une telle comparaison. Les architectures basées sur le

parallélisme et leurs paradigmes associés (par exemple, le parallélisme des données, le parallélisme des tâches, l'asynchronisme) sont trop différents pour être comparés. Compte tenu de ces critères, la conception et l'évaluation d'une solution distribuée pour les transformations de modèles n'est pas triviale et très difficile.

Ainsi, les challenges adréssés sont les suivants:

1. Une expression dans une règle de transformation peut être évaluée comme une requête sur le modèle d'entrée de la transformation. L'exécution d'une requête sur un modèle dans un moteur distribué dépend vraiment de la stratégie adoptée par l'utilisateur qui l'a définie. Les modèles de programmation basés sur le calcul distribué ne présentent pas les mêmes avantages selon le cas d'utilisation et le modèle d'entrée.
2. Les moteurs de transformation existants pour effectuer des transformations de modèle sont conçus à partir de différents choix en fonction de leur objectif. Il n'est pas possible de comparer différents choix de conception dans les moteurs de transformation mais de le faire en utilisant des moteurs existants. Il y a trop de différences qui ont un impact sur les performances.
3. La configuration d'un moteur de transformation a un impact sur ses performances. Les choix d'ingénierie dans le développement d'un moteur de transformation influencent fortement le temps de calcul nécessaire à l'exécution d'une transformation.

Contribution

Cette thèse contribue à l'analyse des choix de conception dans la conception d'un moteur d'exécution.

Une première partie de cette thèse est centrée sur les modèles de programmation distribués qui peuvent être utilisés pour exécuter des requêtes sur le modèle. Les règles de transformation sont définies à l'aide d'expressions qui exécutent des requêtes sur les modèles pour interroger le modèle d'entrée afin de vérifier une condition ou d'extraire des informations pour créer un contenu de sortie. Dans cette contribution, nous proposons d'abord une implémentation Scala d'OCL, un langage de requête sur les modèles. Puis nous avons proposé des implémentations basées sur des modèles de programmation distribuée : primitives Spark, MapReduce, Pregel et hybrides.

La deuxième contribution de la thèse cible le choix de conception des moteurs de transformation distribués. Nous proposons un raffinement de la spécification CoqTL pour augmenter les possibilités de parallélisme. La spécification affinée est validée en prouvant formelle-

ment en Coq l'équivalence entrée/sortie à la spécification CoqTL standard. Cette nouvelle spécification est implémentée sur Spark et nous évaluons ses performances sur un cluster distribué. Ces performances incluent les avantages de notre nouvelle solution par rapport à une implémentation de la spécification CoqTL standard, une analyse du temps de calcul et une analyse de l'évolutivité potentielle que ce nouveau moteur peut proposer.

Dans la troisième contribution, nous proposons plusieurs configurations pour SparkTE, le moteur de transformation distribué précédemment implémenté à partir de la nouvelle spécification de CoqTL. Les configurations sont basées sur différentes stratégies pour les étapes de calcul de l'opération de transformation du modèle. Nous proposons un modèle de fonctionnalités pour illustrer les nombreuses options dont nous disposons pour l'exécution de SparkTE. Nous comparons l'impact de choix uniques, mais aussi de configurations complètement modifiées, en expérimentant notre moteur configurable sur un cluster de calcul.

A travers la thèse, nous présentons et expérimentons nos travaux en utilisant des cas d'utilisation bien connus sur des modèles conformes à trois méta-modèles. Pour chacun de ces métamodèles, nous expérimentons une ou plusieurs transformations. Les trois principaux cas sont les suivants.

1. La transformation Relational2Class, qui mappe les éléments relationnels, c'est-à-dire les tables, les types et les colonnes, aux éléments de classe, c'est-à-dire les classes, les types de données et les attributs.
2. Un cas de réseau social, de TTC18 [76]. Ce cas d'utilisation vise à extraire le post le plus débattu d'un réseau social, en fonction de son activité (nombre de commentaires, et de likes).
3. Un cas sur Internet Movie Database (IMDb) [92], où le but est de trouver des couples d'acteurs et/ou d'actrices qui ont joué ensemble dans plusieurs films.

De plus, nous avons utilisé deux autres cas comme exemple pour illustrer la variabilité des résultats : (i) une transformation d'identité sur des modèles IMDb [92] et (ii) une requête sur des modèles DBLP pour trouver des auteurs actifs qui ont publié dans des revues [10].

Contexte de la thèse

Les travaux de cette thèse ont été financé par le projet Européen intitulé Lowcomote [1],qui a reçu un financement du programme de recherche et d'innovation Horizon 2020 de l'Union

1. http://www.lowcomote.eu

européenne dans le cadre de la convention de subvention Marie Skldowska-Curie n°813884. Le but de ce projet est de former 15 doctorants autour des plateformes low-code. Les plateformes low-code (LCDP) servent à concevoir des applications directement via des modèles en minimisant ainsi la part de programmation textuelle, et en maximisant à la place la programmation visuelle.

De plus, ce thèse a pris place au sein des équipes NaoMod [2] (anciennement Atlanmod) et Stack [3]. NaoMod et Stack sont deux équipess du Laboratoire des Sciences du Numérique de Nantes [4] (LS2N), localisée sur les campus de l'UFR Sciences et Techniques, et l'IMT Atlantique de Nantes [5]. L'équipe Naomod est spécialisée dans l'ingenierie des models dans la région de Nantes depuis les années 1990, et a proposé de nombreuses technologies, notamment basées sur Eclipse, a destination des développeurs et architectes logiciels, dans l'optique d'améliorer leur productivité, ainsi que la qualité des applications qu'ils développent. L'équipe Stack est un projet initié par l'INRIA, qui traite des défis liés à la gestion et l'utilisation avancées des infrastructures de l'informatique utilitaire. Les activités de l'équipe se concentrent sur la définition d'abstractions et de mécanismes permettant d'opérer les futures infrastructures massivement géo-réparties (Fog/Edge).

2. https://naomod.github.io/
3. https://stack-research-group.gitlabpages.inria.fr/web/
4. https://www.ls2n.fr/
5. https://www.imt-atlantique.fr

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ATL	ATLAS Transformation Language
BMF	Bird-Meertens Formalism
CoC	Calculus of Constructions
CPU	Central Processing Unit
EMF	Eclipse Modeling Framework
ETL	Epsilon Transformation Language
IMDb	Internet Movie Database
JVM	Java Virtual Machine
LHS	Left Hand Size
MBSE	Model-Based Software Engineering
MDE	Model Driven Engineering
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
MOF	Meta-Object Facility
MPI	Message Passing Interface
MQ	Model Query
MR	Model Transformation
OCL	Object Constraints Language
OMG	Object Management Group
OOP	Objct-Oriented Programming
PaaS	Platform as a Service
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QVT	Query View Transformation
RDD	Resilient Distributed Dataset
RHS	Right Hand Side
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SSSP	Single Source Shortest Path
UML	Unified Modeling Language
VLM	Very Large Model

CHAPTER 1

CONTEXT

Contents

1.1 Introduction

Model-Based Software Engineering (MBSE) has become a popular approach. Modeling is an activity present in all scientific areas which aims at giving a simplified view of real-world entities such as biology, mathematics, civil engineering, product lines, or philosophic disciplines [7, 8, 11, 68, 141]. More precisely, Model Driven Engineering (MDE) is a software engineering approach that stages models as a central point for developing software. In software engineering, models are used to describe all the aspects of software and systems including their architectures, actions, physical components, algorithms, or communications. For standardization reasons, the Object Management Group (OMG) defined a standard language for representing software systems named Unified Modeling Language (UML) [145]. Considering the many roles of people involved in the conception of an application, having central models shared by the actors helps comprehension, and constitutes a strong knowledge view. For sure, there exist many types of models, with different purposes, and they do not need the same level of expertise. But models are often considered as a bridge between the actors around the software. Models now constitute a living part of projects and are often manipulated as individual entities.

In the context of MDE, model management tools are software applications that help organizations manage the development, deployment, and maintenance of models that are

used in the MDE process. These tools typically provide features such as version control, collaboration, and tracking of model performance. By using model management tools, organizations can improve the efficiency and effectiveness of their MDE processes. Examples of model management tools for MDE include tools such as the Eclipse Modeling Framework (EMF) [153] and the Meta Object Facility (MOF) [157].

The two main categories of model-management tools we consider are model transformation (MT) and model query (MQ). On the one hand, model transformation is the conversion process of one or more input models to output models (model-to-model) or text (model-to-text). A model transformation that produces a model as output can be either an in-place (i. e., direct modification of the input model) or an out-place transformation (i. e. production of a new model from the input one). On the other hand, a model query analyzes source models to compute the desired data value. In the literature, there is a clear distinction between these two, even if queries can be expressed as transformations, and vice versa. We name query and atomic expression, used to obtain a single result, while a model transformation is the full execution of rules, based on queries, by a dedicated engine. Note that there also exist engines to run single queries.

Models are used to represent real-life entities, like data. Since we live in a world surrounded by sensors, we create information continuously, increasing the size of the produced data. The processing of this huge amount of data is called Big Data. In MDE, we refer to models containing a lot of information as Very Large Models (VLMs). Tools dedicated to the treatment of VLMs have a significant need for automatic and transparent efficient and scalable operations, for manipulating, querying, and analyzing models. Most of the model-management operations are executed at design time, e.g., for editing, validating, and transforming the model. The required time for responding to a graphical command is a quality factor of a MDE tool and influences the developer's comfort.

1.2 Problem Statement

MDE is a powerful method for organizing and theoretically treating every kind of data. VLMs pose additional challenges, due to the size of the data they represent. A scalability issue arises when a tool needs to manipulate large instance models of data, e.g., as it happens today in several (automotive, aeronautics, civil) engineering domains. Because of the sheer size of the model, providing an efficient solution is necessary. A second scalability issue arises when there is a need to run a big number of operations in parallel for many users. In

the context of a Platform as a Service (PaaS), numerous customers may query models. Hence, efficient concurrent execution of model management operations is necessary. To improve efficiency and scalability, recent research on model-management studied parallel and concurrent programming as well as specific execution models for model management languages. These techniques range from implementing specific execution algorithms (e.g., RETE [74]) to compiling toward distributed programming models (e.g., MapReduce [65]). The diversity of strategies that have been employed poses several scientific challenges.

Most of the solutions have been independently developed, on top of different technologies, and with different purposes. Since the numerous solutions are not formalized within a unified solution, it is not possible to give a clear, and precise, comparison of these tools. The introduction of parallel solutions, which are non-deterministic approaches, makes even harder such a comparison. The architectures based on parallelism, and their associated paradigms (e.g, data-parallelism, task-parallelism, asynchronism), are too different to be compared. Considering these criteria, designing and evaluating a distributed solution for model transformations is not trivial and very challenging.

In this thesis, we adress the following challenges:

1. An expression in a transformation rule can be evaluated as a query on the input model of the transformation. The execution of a query on a model in a distributed engine depends on the strategy adopted by the user who defined it. The programming models based on distributed computing do not show the same benefits according to the use case and the input model.
2. The existing engines for performing model transformations are designed from different choices according to their purpose. Comparing different design choices in transformation engines but making it using existing engines is not possible. There are too many differences that impact performance.
3. The configuration of a transformation engine has an impact on its performance. Engineering choices in the development of a transformation engine deeply influence the computation time needed for running a transformation.

1.3 Contributions

This thesis contributes to the analysis of design choices in the design of an execution engine.

The first part of this thesis is focused on the distributed-based programming models that can be used to run queries on the model. Transformation rules are defined using expressions, that run queries on the models for interrogating the input model to either check a condition or extract information for creating output content. In this contribution, we first propose a Scala implementation of OCL, a query language on models. Then we proposed implementations based on distributed programming models: Spark primitives, MapReduce, Pregel, and hybrids.

The second contribution of the thesis targets the design choice of distributed transformation engines. We propose a refinement of the CoqTL specification to increase parallelism opportunities. The refined specification is validated by formally proving in Coq the input/output equivalence to the standard CoqTL specification. This new specification is implemented on top of Spark and we evaluate its performance on a distributed cluster. These performances include the benefits of our new solution compared to an implementation of the standard CoqTL specification, an analysis of computation time, and an analysis of the potential scalability this new engine can propose.

In the third contribution, we propose multiple configurations for SparkTE, the distributed transformation engine previously implemented from the new specification of CoqTL. The configurations are based on different strategies for the computational steps of the model transformation operation. We propose a feature model to illustrate the numerous options we have for the execution of SparkTE. We compare the impact of single choices, but also of completely changed configurations, by experimenting with our configurable engine on a computational cluster.

Through the thesis, we present, and experiment, with our work using well-know use cases on models that conform to three meta-models. For each of these metamodels, we experiment with one or several transformations. The three main cases are the following.

1. The Relational2Class transformation, which maps relational elements, i.e., tables, types, and columns, to class elements, i.e., classes, datatypes, and attributes.
2. A Social Network case, from TTC18 [76]. This use case aims at extracting the most debated post in a social network, based on its activity (number of comments, and likes).
3. A case on the Internet Movie Database (IMDb) [92], where the goal is to find couples of actors and/or actresses who used to play together in several movies.

In addition, we used two other cases as examples to illustrate the variability of results: (i) an identity transformation on IMDb models [92] and (ii) a query on DBLP models to find active

authors who published in specific journals [10].

1.4 Outline of the thesis

This thesis is organized as follows. **Chapter 2** presents some background material and key concepts for the comprehension of the rest of the document. **Chapter 3** presents the state of the art that already targets the problem we are trying to face in this thesis. In **Chapter 4**, we present a comparison of model queries, based on several programming models. **Chapter 5** proposes two main contributions: a refinement of CoqTL to increase the parallelism opportunities (Sect. 5.4), and SparkTE, its implementation on top of Spark (Sect. 5.3). In **Chapter 6**, we enrich SparkTE with a configuration aspect, to select the different strategies to run transformations on the engine. We finally conclude in **Chapter 7** by giving a global summary of the contributions and their limits and proposing future work as perspectives. Two appendices are presented at the end of the document. (i) **Appendix A** illustrates how we extended NeoEMF-IO for loading serializable models distributable on Spark, and **Appendix B** which presents an excerpt of benchmarking library for multi-parameter applications.

1.5 Scientific productions

During this thesis, we produced 2 articles: 1 international conference, and 1 international workshop.

- International conference
  - Jolan Philippe, Massimo Tisi, Hélène Coullon, Gerson Sunyé. Executing Certified Model Transformations on Apache Spark. SLE 2021: 14th ACM SIGPLAN International Conference on Software Language Engineering, Oct 2021, Chicago IL, United States. `https://hal.archives-ouvertes.fr/hal-03343942`
- International workshop
  - Jolan Philippe, Hélène Coullon, Massimo Tisi, Gerson Sunyé. Towards Transparent Combination of Model Management Execution Strategies for Low-Code Development Platforms. 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, Oct 2020, Montreal (Virtually), Canada. `https://hal.archives-ouvertes.fr/hal-02952952`

CHAPTER 2

PRELIMINARIES

Contents

In this chapter, we present various conceptual and software tools to ease the reading of the contribution chapters. First, we begin to present what is Model Driven Engineering (Section 2.1). We give general concepts and then define what is model transformation and how it can be defined. Section 2.2 introduce distributed computation and more specially Spark, a unified engine for data-distributed computation. Finally, an overview of what is correctness of programs is given in Sect. 2.3. In this last section, we introduce Coq, a proof assistant designed for interactive theorem proving, and Coq-based language for defining model transformations: CoqTL.

2.1 Model transformations

In the 2000s, the Object Management Group (OMG) introduced the concept of Model-Driven Architecture (MDA). The main goal of MDA is to provide tools for solving issues

raised by complex systems. The approach separates the functionalities of a system from its implementation. It allows users to emancipate the functionalities from the target platform by supplying 'vendor-neutral interoperability specifications". MDA promotes the usage of Platform-Independent Models (PIM) as primary artifacts to design systems and software architectures. These models can then be adapted to a specific platform (i.e., Platform-Specific Model (PSM)) using successive transformations, refinements, and finally, code generation [104, 149]. We call this process forward engineering. On the contrary, PIM can be reconstructed by analyzing PSM. This new PIM can then be used as a basis for code modernization, maintenance and, enhancement. We call this process reverse engineering [33]. These concepts can be generalized under model transformation, a concept introduced in Sect. 2.1.2. Model Driven Engineering (MDE) is a more general approach that emerged from MDA. MDE is not limited to architecture, but also processes, data representation, and analysis.

In the rest of the section, we first introduce general concepts of MDE about modeling (Section 2.1.1). Then, in Section 2.1.2, we introduce transformation rules and engines to run through. Finally, we give an overview of ATL, a language used for expressing model transformations in Sect. 2.1.3.

2.1.1 Modeling concepts

MDE is a software engineering approach born from the Object Oriented Programming (OOP) paradigm. OOP is based on the concept of "objects", which are atomic abstractions that encapsulate data to describe and substitute real entities. In the Java language, or in its functional support Scala, the API defines an object with a name and a set of attributes, as an instance of class. A class gives a more general definition of how objects are structured with identifiers: a classname, a set of behavior methods, and a set of fields. A class instance, that is an object, inherits all the features described in the class definition.

In a MDE approach, a set of object definitions is named a `model`. There exist about ten of definitions for what is a model [129]. In this thesis, we refer to OMG's definition: "*A model of a system is a description or specification of that system and its environment for some certain purpose.*". In other words, a model describes data, and its environment, in a specific context. In an object-oriented context, a class can be instantiated into an object. In MDE, the definition of the structure of a model is named a `metamodel`. A metamodel is an explicit specification of an abstraction [128, 28]. It is used to define a list of concepts (e.g., structure, classes), and how these concepts interact with each other. All the metamodels in this the-

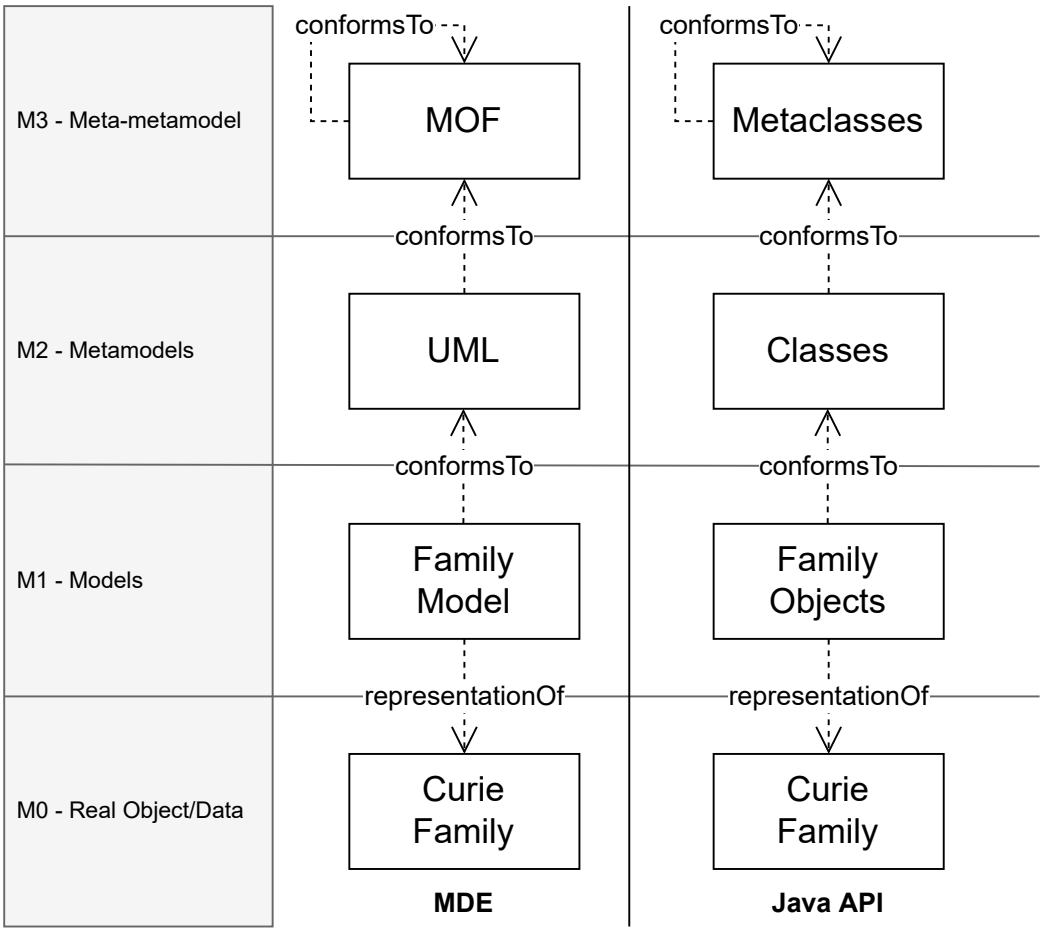


Figure 2.1 – Modelisation hierarchy from real-world things to meta-metamodels

sis follow the OMG's Meta-Object Facility (MOF) [1]. Then the metamodel can be instantiated into a model, that conforms to the metamodel abstract syntax and satisfies the metamodel semantics. The syntax and the semantics of a metamodel definition can itself be viewed as metamodel, namely `meta-metamodel`. For instance, UML, which stands for Unified Modeling Language, was introduced in the late 90s as a standard language for modeling object-oriented systems. In UML, the metamodels all conform to the Meta-Object Facility (MOF) meta-metamodel. This concept can be associated of metaclasses from programming languages, that are used in reflexive programming.

Figure 2.1 from [95] summarizes the four-level of a metamodeling architecture, with the example of modeling families:

— The real-world thing is a concrete family, with family members (layer $M0$). For in-

1. https://www.omg.org/mof/

stance, the Curie family is composed of Marie, Pierre, and their two daughters: Irène and Ève. They all were existing people, who lived in the real world.

— The $M1$ layer groups the family models, which represent the data of the $M0$ layer. In the family case, the model states the name and the place of the family members.
— The $M2$ layer contains metamodels whose models of $M1$ conform to. For example, the family model conforms to a UML metamodel describing how a family is constructed.
— Finally, the meta-model from the $M2$ layer conforms to a meta-metamodel of the layer $M3$. For example, a UML metamodel conforms to the MOF meta-metamodel.

2.1.2 Model transformation

Model transformation (MT) is certainly one of the most used concepts in MDE. It is the conversion process of one or more input models to output models (model-to-model) or text (model-to-text). A model transformation that produces a model as output can be either an in-place (i.e., direct modification of the input model) or an out-place transformation (i.e. production of a new model from the input one). To perform transformations, input content must be analyzed. Model queries (MQ) analyze source models to compute desired data values. A query can be used to express a condition on an input element or evaluate content for the output model.

Even if these transformations can be expressed in any language, transformation engines mostly use dedicated tools and languages. These tools take advantage of the previously introduced 4-layers. The languages are used to specify the equivalence between a source and a target metamodel. The transformation can then be applied to a source model, which conforms to the source metamodel of the transformation. As output, the target model will conform to the target metamodel. Figure 2.2 presents the structure of a model transformation from a source model to a target model.

As a concrete example of transformation, we can consider a change in the representation of members of a family. In a `Family2Person` transformation, we transform all `Member` of a `Family` into `Person`, either a `Male` or `Female`.

2.1.3 ATL: an example of transformation language

Model transformation languages have been designed to help users specify model transformations. Among the numerous proposals [103], the ATLAS Transformation Language (ATL) [98, 99] is a rule-based transformation language. The ATL language specifies transformations in

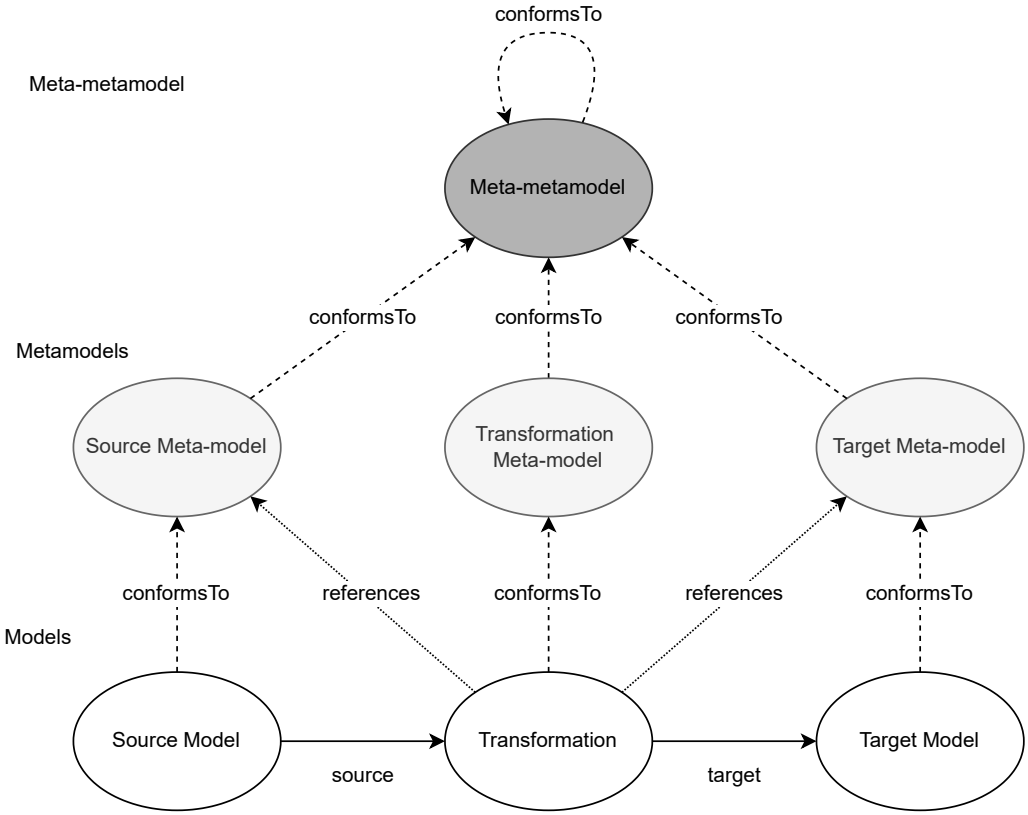


Figure 2.2 – Structure of a model transformation

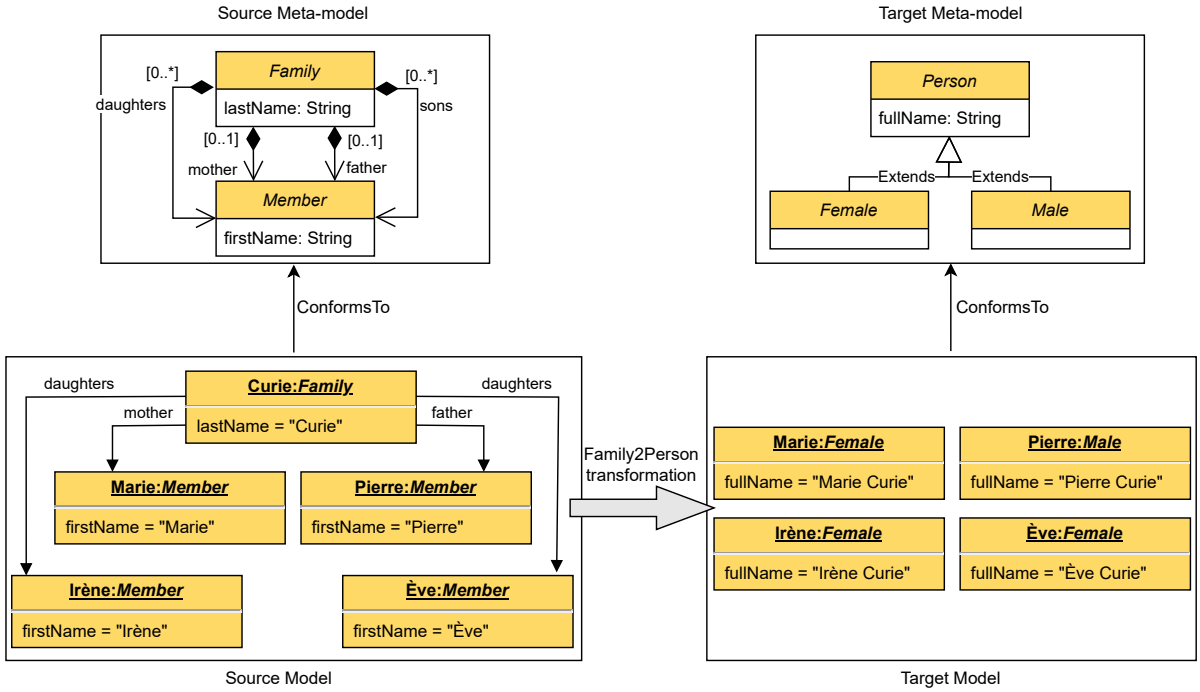


Figure 2.3 – A model conforms to the Family metamodel transformed into a model conforms to the Person metamodel

modules, from a read-only input model. The user defines rules in two parts. First, an input pattern is defined in the first section of the rule (**from** clause), as a set of typed elements from the input model, optionally with a guard condition. Second, an output pattern is declared in the second section of the rule (**to** clause). The bindings of values for output elements are calculated here, using queries on the input model. The query language used in ATL is OMG's Object Constraints Language (OCL) [84].

Assuming there exist a helper `isFemale` that retuns **true** if a `Member` is involved in a `mother` or `daughters` relationship, and second helper `familyName` giving the family name of a `Member`, the `Family2Person` transformation shown in Figure 2.3 can be defined as two rules as illustrated in Listing 2.1. The first rule `Member2Female` instantiates a `Female` from a match `Member s`, with `s.isFemale()` evaluated to **true**. The `fullName` attribute of the output `Female` is binded to a string value calculated from the `firstName` and the `familyName` of the matched input pattern. The second rule `Member2Male` follows the same approach.

In ATL, rule matching and rule application are separated into two phases. In the first phase, all patterns of the rules are matched against the source model(s). For every match, the target elements are created. Traceability links are also created in this phase. In the second

```
1  module Family2Person;
2  create OUT : Person from IN : Family;
3
4  helper context Family!Member def: familyName : String = ...
5  helper context Family!Member def: isFemale() : Boolean = ...
6
7  rule Member2Female {
8      from
9          s : Family!Member (s.isFemale())
10     to
11         t : Person!Female (
12             fullName <- s.firstName + ' ' + s.familyName
13         )
14 }
15
16 rule Member2Male {
17     from
18         s : Family!Member (not s.isFemale())
19     to
20         t : Person!Male (
21             fullName <- s.firstName + ' ' + s.familyName
22         )
23 }
```

Listing 2.1 – ATL definition of the Family2Person transformation

phase, all the bindings for the created target elements are executed.

Several engines exist for performing transformations specified in ATL. For instance, in [112] Le Calvar et. al. proposed an incremental execution of ATL transformations, while Tisi et. al. proposed a "call-by-need" approach for evaluating ATL rules in [166]. Each of the solutions is a different implementation, with its purpose, developed thanks to the ATL toolkit. Additional implementations for running ATL transformations, based on parallelism, are discussed in Chapter 3.

2.2 Distributed computation

2.2.1 Distributed computation and data-distributed programs

In the modern computer science world, system architectures are described using Flynn's taxonomy [73]. It represents computer architectures with four approaches:

— **Single Instruction Single Data (SISD)** corresponds to a uniprocessor model. The data are treated one by one, in sequential order. This architecture is also known as the Von Neumann architecture [155].

— In a **Single Instruction Multiple Data (SIMD)** architecture, one operation is applied to several data. Most of modern processors use SIMD as a vectorial approach. **Single Program Multiple Data (SPMD)** is a variant of this approach. It is very similar to these two approaches that are not mutually exclusive. SPMD is a much higher level of abstraction. The processors operate on different subsets of the data, but different operations may be applied at the same time.

— **Multiple Instruction Single Data (MISD)** applies successive treatments to data. This category is usually associated with pipelines and numerical filtering.

— **Multiple Instruction Multiple Data (MIMD)** is the most used parallel architecture. It is composed of calculation units with their data to treat. In other words, treatments are entirely independent. This architecture can be used with two types of memory:

  — Different programs at the same time can access the shared memory. Languages provide libraries to do shared memory parallelism, such as the Pthreads library in C [130]. This architecture has, however, limitations. All the Central Processing Unit (CPU) cores access the memory with a shared bus, which represents an obvious concurrency issue. Critical sections must be defined, and memory accesses must be restricted.

— With distributed memory, each processor has its private memory and is the only which can access its data. Concrete communications must operate the exchange of information between processors (e.g., using Message Passing Interface (MPI) [148]).

Modern computers contain all of these architectures but at different levels of their overall architecture. In a data-parallel approach, data is split and distributed across several computation units. Then, the same piece of program (from a single basic operation to a complex function) is applied simultaneously on each part of the data by each processing unit without synchronization. From the user's point of view, it follows a SPMD approach. This computation strategy is the one followed by the parallel algorithmic skeletons [27] on data structures [63, 30]. Most of the recent frameworks designed for distributed computation on data are based on a MIMD approach at the framework level, where data is distributed and independently receives a single instruction to compute on their owned chunks. In this thesis, we mostly use Apache Spark [176], which is a framework built with this approach, as a solution for distributed computation. It is defined as a unified multi-language engine for executing data engineering on single-node machines or clusters. Each computation node receives a chunk of data and independently processes an instruction on it. Since the global computation is driven by a master node, the same instruction is replicated on each node. More details about Spark architecture is given in Section 2.2.2. Furthermore, additional synchronizations and communications may be needed between processing units to correctly compute the overall result. For instance, data may need to be merged into a single result. Additional details about how merging results is processed in Spark are given in Section 2.2.3. This computation strategy is the one followed by the parallel algorithmic skeletons [52] on data structures [137, 55]. *MapReduce* [65] is an example of a programming model, designed for parallelism, that takes advantage of this strategy. However, MapReduce is mainly adapted and implemented for distributed arrays or lists, and the approach is not directly suitable for all types of data structures.

For instance, *Pregel* [122] is a strategy that aims at easing parallel computations on graphs by using a vertex-centric approach [125]. In Pregel, graphs are specified by their vertices, each of them embedding information on their incoming and/or outgoing edges. A Pregel program is iterative, and is decomposed into three main phases: (i) a computation on top of a vertex value, (ii) a generation and the sending of output messages through the edges of the vertex, (iii) and the receipt and merge of incoming messages. This process is simultaneously applied to each vertex of a graph (such as a `map` in the MapReduce model). For example, solving the Single Source Shortest Path (SSSP) problem could be expressed as il-

lustrated in Listing 2.2. The computation is expressed as follows. First, an initial message is sent to all (line 2) nodes. Then, the vertex program (line 3) checks if the message received contains a distance small than the current assigned one. The new value of the vertex is the minimum between the previously assigned shortest distance, and the one received. Note that this program treats a message that is already a merge of all incoming messages. Line 4 to 10 defines the logic for sending messages along outgoing edges. It is defined as a function taking a triplet as argument, which contains the source vertex, a target, and the distance between the source and the target vertices. The function tests if the target vertex is assigned to a distance that is smaller than the distance assigned to the source plus the distance from the source to the target vertices. If no, a message is sent with the new distance (line 6). Otherwise, no message is sent, and the source vertex is halted until the moment the vertex receives a new message from another vertex.

Listing 2.2 – GraphX code for solving the SSSP problem

```
1  sssp = graph.pregel
2  (Double.PositiveInfinity) // Initial message
3  ( (id, curr, received) => math.min(curr, received), // Vertex Program
4    triplet => { // Send Message
5      if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
6        Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
7      } else {
8        Iterator.empty // No message
9      }
10   },
11   (m1, m2) => math.min(m1, m2) // Merge Message
12 )
```

Data-parallelism is adapted and adopted in the case of large datasets. Indeed, to make profitable the parallel execution of a single computation on data, the data chunks must be large enough, otherwise, an overhead has to be paid without much benefits from the parallelization effort [3, 53].

Task-distribution Data-distribution is not the only approach for distributing computation. A *task-parallel* program focuses on the distribution of tasks instead of data. According to [144], “a task is a basic unit of programming that an operating system controls” within a job. This concept is often associated with multi-threading. The grain size of tasks depends

on the context of the execution. At the operating system level, tasks may be entire programs while at the program level, they may be a single request or a single operation. Because of concurrency, and the limited number of processing units, task executions must be ordered by considering both priorities, and dependencies across tasks. Ordering tasks in parallel are similar to the workflow concept. Task-parallelism will be preferred to data-parallelism when tasks are complex enough, or when the number of tasks is large enough to exploit parallelism capacities of the underlying parallel architecture (i.e., hardware).

Asynchronism Both data-parallelism and task-parallelism can be defined as synchronous strategies where synchronizations are explicitly performed through communication patterns or task dependencies. *Asynchronism* is another way of programming parallelism where synchronism is not explicitly coded but implicitly handled by an additional mechanism between processing units. For example, the Linda approach [44], is based on the treatment of asynchronous tasks or data, shared in a common knowledge base, the "blackboard" [38]. More specifically, in Linda several processes access a shared tuple space representing the shared knowledge of a system. The processing units interact with the shared space by reading, and/or removing tuples.

2.2.2 Architecture of a Spark cluster

A Spark cluster is composed of a set of machines with specific roles.

— a `master` node hosts a `driver` program, that aims at driving the full distribution of data and jobs to do among the other computational nodes. The `driver` program coordinates the sets of processes with a `SparkContext`, a Java object which contains all the information about the rest of the cluster (e.g., computational nodes and their amount of allocated resources, a job name). The main purpose of the `driver` is to schedule the tasks that will be submitted to the cluster.

— a set of `workers`, that are connected to the `driver`, which concretely compute the tasks submitted to the `driver`. Each `worker` has its number of working threads, private memory, and cache. All workers can be independently configured. It is preferable to keep the `driver` and the `workers` in the same cluster for limiting network issues.

Outside the master and worker nodes, an external `bf`cluster manager can be deployed. Several technologies are supported: Apache Mesos [75] (deprecated), Hadoop YARN [172] or Kubernetes [37]. By default, in a standalone mode, the `master` program is in charge of managing the rest of the cluster. Note that a single machine can assume several roles. In that case,

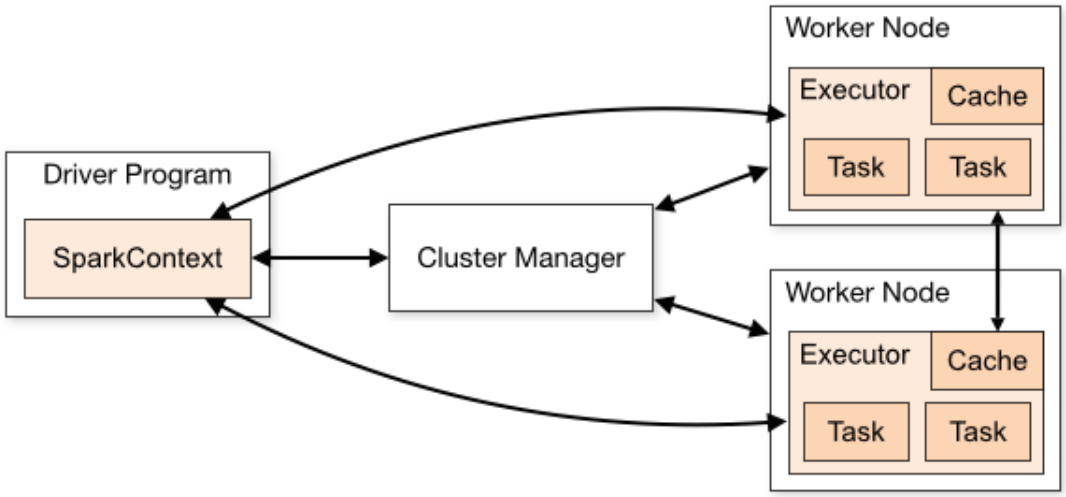


Figure 2.4 – Overview of a Spark cluster architecture [2]

different Java Virtual Machines (JVM), that is executing environments for running compiled Java code with a framed amount of resources (memory, threads), is instantiated: one for each hosted node. The role of the cluster manager is to allocate the asked amount of resources to each of the nodes of the cluster, i.e., the `driver` and `worker` nodes. A global view of a Spark cluster is built is illustrated in Fig. 2.4. Additional details about how tasks and jobs are managed in Spark are given in Sect. 2.2.3.

2.2.3 Jobs in Spark

A user program, i.e., the `driver` program, referred as an `application`, is built and submitted to the `master` node upon the cluster. It is written as a simple Java or Scala program (both working on the JVM), with a `main()` functions driving the full computation. Thanks to the Spark environment, it is possible to distribute the computation on a data-structures using `Resilient Distributed Datasets (RDDs)`. RDDs are parallelized collections, create from a collection of data (e.g., a Scala `Seq`), providing a SPMD view to the user. Indeed, the use of methods to manipulate a RDD is similar to manipulating a sequential collection of data, but it will be processed in parallel. There exists two kinds of methods for RDDs:

— A **transformation** is a function that builds a tree of computation for the data and produces a new RDD from the existing one. Transformations are lazy, that is they do not

2. `https://spark.apache.org/docs/latest/cluster-overview.html`, Copyright © 2018 The Apache Software Foundation, Licensed under the Apache License, Version 2.0

execute any action until we explicitly ask for it. It allows the optimization of the composition of functions. Either by transforming a chain of computation into the simplest one following transformation rules [116] or by optimizing the needed communications between the nodes during the computation.

— A concrete **action** triggers the computation on a RDD. Contrary to transformations, actions do not create new RDDs, but return a value, that will be gathered in the `master` node.

Considering `sc`, the `SparkContext` of an application, an example of distributing a list into a RDD in Scala, and processing a simple calculation of size, can be done as follows.

```
1 val data = List(1, 2, 3, 4, 5)
2 val length = sc.parallelize(data).count
```

Contrary to other data-distributed frameworks, such as Hadoop MapReduce [66], Spark is designed to handle real-time data efficiently. Besides submitting applications to a cluster, it is possible to process data interactively. While Hadoop's model proposes an execution based on lifecycles, with a reading-writing at each lap of map-reduce operation, Spark processes all its computation in memory. In other words, Spark stored all intermediate data in memory, improving the speed of processing. The main drawback of this approach is the requirement for a lot of memory, increasing the cluster size.

2.2.4 Spark libraries

Spark is defined as a "Unified engine for large-scale data analytics" [3]. It provides support for several data-distributed approaches, with different purposes (e.g., distributed operations, data analytics, machine learning) on several languages: Python, SQL, Scala, Java or, R. Since most the Spark technologies only exist on Scala [133], we only used this language for running Spark applications.

Scala library. Scala combines object-oriented and functional programming in one concise, high-level language. In Scala, RDDs are instantiated objects on which methods can be called. As explained before, the methods are separated into two categories: transformations and actions. In this thesis, we only use a subset of transformations that are the following:

— The `map` function takes a function as an argument, and iterates over every element of a RDD to apply this function.

3. `https://spark.apache.org/`

— Spark RDD `flatMap` is the composition of a `map` operation called with a function that produces a collection, and a flatten operation.
— A `filter` call on a RDD removes all elements that do not validate a predicate passed as an argument.
— While the `union` operation concatenates two RDDs into a new RDD containing the elements of both input RDDs, the `intersect` operation only keeps elements that are present in the two RDDs.
— The `distinct` function returns a new dataset that contains the distinct elements of the source. In other words, `distinct` removes duplicated elements in a RDD.
— `groupByKey` is an operation aiming at grouping elements according to a key value. It transforms a RDD of pair of 'key'-'value', into a RDD of 'key'-'list of values'.

In the rest of the thesis, we also use the following triggering actions:

— `collect` is a simple action that triggers the computation and gathers the resulting RDD into a Scala `Array` on the driver program.
— The `count` operation simply counts the number of elements contained in a RDD.
— The `reduce` function takes an associative operation as an argument and uses it to reduce the elements into a single value. The `fold` function is the same operation but considers an initial value to initiate the reduction with the first element of the RDD

An example of how these transformations and actions can be used in Scala to perform the computation of the variance value of a set of doubles can be expressed as follows.

```
1 def variance(values: List[Double], sc: SparkContext): Double = {
2     val X = sc.parallelize(values)
3     val length = X.count
4     val avg = X.fold(0)((a, b) => a + b).collect / length
5     val variance = X.map(x => Math.pow((x - avg), 2)).reduce((a, b) => a +
          b) / length
6     return variance
7 }
```

GraphX. Besides the core library of Spark, there exist several additional libraries with a more specific purpose. Among them, GraphX [80] aims at processing large-scale graphs with vertex-centric and triplet-view approaches. The former is, as described above, a synchronous approach where vertex program kernels are executed iteratively for a certain number of rounds. Contrary to a conventional SIMD data-distributed approach, vertices only have a

view of their data. Messages are expressly defined to make them communicate information. GraphX also proposes a global view of the distributed data through triplets. Triplets in GraphX are represented as a RDD of tuples, where each triplet describes an edge of the graph as a tuple of the source id, the source data, the target id, the target data, and the data associated with the edge (e.g., weight or label). An example of how a GraphX can be used to represent the Family model (left) of Fig. 2.3 is presented in Listing 2.3.

Listing 2.3 – Example of distributed Family model on GraphX

```
1  val Curie = new Family("Curie")
2  val Marie = new Member("Marie")
3  val Pierre = new Member("Pierre")
4  val Irene = new Member("Irene")
5  val Eve = new Member("Eve")
6  val sc: SparkContext = ...
7  val vertices: RDD[(VertexId, String)] =
8      sc.parallelize(Seq((0L, Curie),(1L, Marie),(2L, Pierre),(3L,
           Irene),(4L, Eve)))
9  val edges: RDD[Edge(String)] =
10     sc.parallelize(Seq(Edge(0L, 1L, "mother"), Edge(0L, 2L, "father"),
11                    Edge(0L, 3L, "daughters"), Edge(0L, 4L, "daughters")))
```

2.3 Interactive Theorem Proving

2.3.1 Correction of programs

In modern software developments, different methods are used to get quality-controlled applications. They are classified as Agile methods and are based on a systems development life cycle. The work in Agile methods is based on iterations of small increments that minimize the amount of planning and design devices of an application. Contrary to waterfall models, where the build phase and the testing phase are separated, the development testing is completed in the same iteration as programming. That is, an increment is both a software component and related tests. More specifically, test-driven development consists of writing the tests focused on requirements before writing the code. Another Agile approach relative to tests is continuous integration. With this practice, all the tests are run at each modification of the source code to check the absence of regression. However, The main cons of tests are

their specificity. On one hand, it is straightforward and natural to write and execute tests. On the other hand, it only verifies the quality of the program for specific cases. In other words, tests do not cover all the possibilities of execution. According to Edsger W. Dijkstra [142],

"Program testing can be used to show the presence of bugs,
but never to show their absence."

Formal methods will be preferred to verify the correctness of a program for all possible inputs and execution.

By construction In correctness-by-construction, the specification is written first and then transformed step-by-step into an efficient executable program. Each transformation is proved correct. In other words, the previous and the new models are shown as equivalent. For example, the Bird-Meertens Formalism (BMF) (also called Squiggol) [12, 30, 79, 126] is a calculus that provides rules of equivalence between standard primitives on data structures to get a more efficient program. Program calculation, in particular of functional programs [29, 78], is a style of reasoning of correctness-by-construction. Proofs assistants [150] are well suited to conduct program calculation reasoning on functional languages [159]. SparkTE, which is presented in this thesis, is a model transformation engine that has been designed using a correctness-by-construction approach. There also exist similar methods for imperative languages such as the B-method [43], which models the abstract specification of a program to obtain a concrete C or Ada executable program.

A posteri verification In *a posteri* verification, the specification and the program are written independently. When they are both finished, a proof of correspondence is made to ensure the correctness of the program. The usual approach for doing this kind of verification is by using a deductive system, also called deductive inference. It consists of the use of axioms, or inference rules, defined in a semantic [131] to prove properties at a specific moment of the execution of a program. The most used formal system is the Hoare logic [91], inspired by Floyd's works on flowcharts [72]. Hoare logic is based on Hoare triples describing a state of the computation. This logic can be naturally applied to most sequential imperative programs. There exist tools to support this kind of verification for the languages, such as Frama-C for sequential C Programs [57]. CoqTL [163] proofs are conducted following this approach. On one side, the user can define a transformation using the internal DSL of CoqTL and express properties on it. The properties are then proved using the Coq proving mechanism.

2.3.2 The Coq proof assistant

The proof assistant Coq is based on the mathematical theory CoC (Calculus of Constructions). Coq is divided into three sublanguages:

— `Gallina` to write Coq terms (functions, types, axioms, etc.). Its syntax is very similar to OCaml's;

— `Vernacular` to control the behavior of the proof assistant;

— `LTac`, a language of tactics allowing to construct proofs interactively.

Every term of Gallina has a type, and the types are also terms of the language.

Type definitions Every object of Coq has a type, including the types themselves. The types are defining ordered sorts with `Set` as the bottom of the hierarchy. The following inclusions hold.

$$Set \leq Type_0 \leq Type_1 \leq Type_2 \leq \ldots$$

For any $i < j$, a Typei is typed by $Type_j$. In implies that the particular case of Set is typed by $Type_i$ with any i. Since `Set` is the type of the "small" datatypes and function types, cannot directly or indirectly involve other types [97]. As indicated in its original name, Coq is based on the calculus of construction theory. A definition is made using `Definition` and is constructed as follows.

```
Definition name : type := definition.
```

The pure type system from the CoC has been extended with inductive definitions from the Calculus of inductive Constructions (CiC). It is possible to write an inductive definition using the keyword `Inductive`. For example, natural numbers are defined in the standard library by:

```
Inductive nat: Set :=
| O: nat
| S: nat → nat.
```

Another type can parametrize the definition of a type. It is very convenient to use parameters for polymorphic structures. For example, the lists are defined in Coq by:

```
Inductive list (A: Type): Type :=
| nil: list A
| cons: A → list A → list A.
```

Instead of `nil` and `cons h t`, we can respectively use `[]` and `h :: t`.

Functions Coq provides a pattern-matching mechanism for defining functions. By filtering cases characterized by a pattern, different behaviors can be defined. However, it is important to notice that all the functions in Coq must be total. In other words, a function must determine behavior for every constructor of inputs. For example, the functions pred that returns the predecessor of a natural number can be defined by:

```
Definition pred (n: nat): nat :=
 match n with
  | O ⇒ O
  | S m ⇒ m
 end.
```

Besides, to set a recursive function, the keyword `Definition` must be replaced by `Fixpoint`. Note that Coq only allows definitions of functions that terminate. A recursive function must have a decreasing argument. The map function on lists can be described as follows.

```
Fixpoint map (A B: Type) (f: A→ B) (l: list A): list B :=
 match l with
  | [] ⇒ []
  | h :: t ⇒ (f h) :: (map A B f t)
 end.
```

Proofs in Coq From definitions, it is possible to define lemmas, properties, or theorems in Coq with related proof. For example, from the function length on lists, that the number of elements in the structure, and map defined previously, we define the following property.

```
Lemma map_length: ∀ A B (f: A→ B) (l: list A),
 length (map f l) = length l.
```

Proof of this lemma can be written using LTac, the tactic language.

```
Proof.
 intros A B f l.
 induction l as [| x xs Hx].
 + simpl. reflexivity.
 + simpl. rewrite Hx; reflexivity.
Qed.
```

Let us analyze this step by step. First, we start the proof using the keyword `Proof`. The environment of Coq returns a response indicating that there is still one subgoal to prove.

```
1 subgoal
______________________________________
∀ (A B: Type)(f: A → B)(l: list A), length (map A B f l) = length l
```

To start the proof, we need to introduce the variables we will use: `intros A B f l`.

```
A: Type
B: Type
f: A → B
l: list A
____________________________________
length (map A B f l) = length l
```

In our lemma, `l` is a list, and then its definition is made by induction. We need to do an induction on the structure of `l` for solving this proof: `induction l as [| x xs Hx]`. This tactic can be understood such there are two cases separated by `|`. The first case is the situation of `l` is `nil`. There is nothing to define here. Otherwise, `l` is `cons x xs`, and we name the inductive hypothesis with `Hx`. The answer of Coq shows two subgoals: one for each possible constructor of `l`.

```
A: Type
B: Type
f: A → B
____________________________________
length (map A B f []) = length []

subgoal 2 is: length (map A B f (x :: xs)) = length (x :: xs)
```

According to the definition of `length`, `length [] = 0`. Since `map A B f [] = []`, `length (map A B f [])` can be simplified by `0` with the tactic `simpl`. The two expressions will be simplified, and the environment returns:

```
A: Type
B: Type
f: A → B
____________________________________
0 = 0
```

Because equality is reflexive, the resolution can be finished by `reflexivity`. There is still the second subgoal to solve.

```
A: Type
B: Type
f: A → B
x: A
xs: list A
Hx: length (map A B f xs) = length xs
____________________________________
```

```
length (map A B f (x :: xs)) = length (x :: xs)
```

Using the simplification with the tactic `simpl` the Coq response is the following.

```
A: Type
B: Type
f: A → B
x: A
xs: list A
Hx: length (map A B f xs) = length xs
------------------------------------
S (length (map A B f xs)) = S (length xs)
```

The resolution can be finished by using the inductive hypothesis and `reflexivity`. We process these two operations using a semi-column by `rewrite → Hx; reflexivity`.

CoqTL CoqTL [163] is an internal language in Coq, for writing rule-based model-transformations. Besides, the language is associated with a library to simplify proving transformation correctness in Coq. The rule definition syntax in CoqTL is inspired from ATL: a `from` section describes the input pattern with a guard condition (`where` clause), while the `to` section describes the output to create from the matched pattern. The `Family2Person` transformation, expressed with ATL in Listing 2.1, can also be written using CoqTL. Considering the `getFirstName`, `getFamilyName`, and `isFemale` helpers and two constructors, `BuildFemale` and `BuildMale`, respectively to build a `Female` and a `Family2Person`, the `Family2Person` transformation can be written as defined in Listing 2.4.

```
Definition Family2Person :=
transformation from FamilyMetamodel to PersonMetamodel with fm as FamilyMetamodel :=

rule Member2Female
    from element m class Member
    where (isFemale fm m)
    to [
        output "female"
            element f class Female :=
            BuildFemale ((getFirstName m) + (getFamilyName fm m))
    ];

rule Member2Male
    from element m class Member
    where notb (isFemale fm m)
```

```
to [
    output "male"
        element f class Male :=
        BuildMale ((getFirstName m) + (getFamilyName fm m))
]
```

Listing 2.4 – CoqTL definition of Family2Person transformation

In CoqTL, the semantic of model transformation is designed in order to help users to prove properties on transformation. CoqTL does not impose a specific schema for writing theorems, but it is recommended to express them in Hoare-style as follows.

— a **transformation**, with an input and an output model;

— a **post-condition** on the source model;

— a **pre-condition** on the target model.

```
Theorem th_tranformation:
    ∀ (sm: SouceModel) (tm: TargetModel), execute tranformation sm = tm →
    precondition sm → postcondition tm.
```

Listing 2.5 – Transformation theorem in CoqTL

A simple theorem to prove for the `Family2Person` transformation is: if all family members have a first name that is not empty, then all output objects in the `PersonModel` has a full name that is not empty. This theorem is expressed as follows.

```
Theorem theorem_tranformation:
    ∀ (sm: FamilyModel) (tm: PersonModel),
    (* tranformation *) execute Family2Person sm = tm →
    (* precondition *) (∀ (m: Member), In m (allModelElements sm)
                        → (getFirstName m <> ""%string)) →
    (* postcondition *) (∀ (p: PersonMetamodel_Object), In p (allModelElements tm)
                        → (getFullName p <> ""%string)).
```

Listing 2.6 – Family2Person theorem in CoqTL

Proving properties in CoqTL [49, 48] is eased by an extension of the LTac language of Coq. Additional tactics and proved general lemmas make automatic certain part of the proof.

CHAPTER 3

STATE OF THE ART

Contents

3.1 Efficiency in model transformation

In this section, we outline the execution strategies that are commonly used to enhance the efficiency of model-management. The below presented strategies have been identified with their use in MDE. In this section, we only focus on the strategies, regardless of the chosen language for their implementations. We also give an overview of the existing applications of these strategies in model-management tools.

Parallelizing computations

Parallelism designates the use of several processing units in order to achieve a global operation. There exist many kinds of parallel architectures, from multi-cores to clusters of GPUs. In this section, we focus on the parallelism strategies that are used to take advantage of parallel architectures.

In Table 3.1, we classify how parallelism has been applied to model management in literature, by the following columns:

— **MQ** or **MT** if the whole model query or transformation is parallelized;

— **Matching** if the work only parallelizes the matching phase of the model query/transformation;

— **Performance** whether the work pays particular attention to the impact of data distribution or task distribution on performance.

We classify the strategies into three categories: data-parallelism (Section 3.1.1); task-parallelism (Section 3.1.2), both of them being synchronous strategies; and one example of asynchronous strategy (Section 3.1.3).

Table 3.1 – Parallelism for model-management in literature.

	MQ	**MT**	**Matching**	**Perf.**
Task-parallelism	[119, 169]	[94, 165]	[127]	
Data-parallelism	[118]	[14, 107, 168, 56]	[107]	[15]
Asynchronism		[34, 35, 36]		[35]

3.1.1 Data-Parallelism

In [118], Madani et al. propose a concurrent version of EVL (Epsilon Validation Language) to validate model properties. This new proposal for EVL can be executed both on parallel and distributed architectures. In parallel-EVL constraints to validate are set in a pool of threads, and executed independently. Besides, tasks are decomposed and distributed in a data-parallel maneer among computational cores.

Benelallam et al. [14] use data-parallelism for distributing models among computational cores to reduce computation time in the ATL model transformation engine. The MapReduce version of ATL makes independent transformations of sub-parts of the model by using a local “match-apply” function. Then, the reduction aims at resolving dependencies between map outputs. The proposed approach guarantees better performance on basic cases such as the transformation of a class diagram to a relational schema. In a more recent work [15], the same authors highlight the good impact of their strategy for data partitioning. Instead of randomly distributing the same number of elements among the processors, they use a strategy based on the connectivity of models. In [56], Cuadrado et al. propose A2L, a compiler for the parallel execution of ATL model transformations, which produces efficient code that

can use existing multicore computer architectures, and applies effective optimizations at the transformation level using static analysis. The execution of the obtained code is in average 22.42x faster than the current ATL implementation.

Distributed computation was also applied to efficiently implement parts of the rule evaluation. For example, the Forgy's RETE algorithm [74] for pattern matching, presented in [171], that constructs a network to specify patterns and, at runtime, tracks matched patterns, has been implemented as a parallel solution in [18]. The proposal harness multi-core architectures by, on one side, enabling concurrent execution of pattern matching, and on the other side, by parallelizing the pattern matching algorithm itself.

[100] illustrates how a model can be considered as a typed graph with inheritance and containment. Considering a model as a graph data-structure, graph technologies can directly be applied to models. For instance, Imre et al. efficiently use a parallel graph transformation algorithm on real-world industrial-sized models for model transformation [94]. In [127], Mezei et al. use graph rewriting operations based on task-parallelism to distribute matching operations in large models in their transformation tool Visual Modeling and Model Transformation (VMTS). The Henshin framework [9] proposes to extract the matching part of its transformation rules into vertex-centric code (i.e., Pregel) [107]. Another possibility to use Pregel in model transformation is by using a DSL, such as [168] for graph transformation. The proposed compiler transforms the code written with the DSL into an executable Pregel code. Finally, MapReduce is used in [69] for finding inexact patterns in graphs. The approach targets graph but it can be easily applied in a MDE context for model validation.

3.1.2 Task-Parallelism

[169] proposes a formal description of parallelism opportunities in OCL. Two main kinds of operation are targeted: the binary operations that can have their operands evaluated simultaneously, and the iterative processes of independent treatments. In [119], Madani et al. use multi-threading for "select-based" operations in EOL, the OCL-like language of the Epsilon framework, for querying models. The extension of the language with parallel features for selective operations have shown a non-negligible speed-up (up to 6x with 16 cores) in their evaluations on a model conform to the Internet Movie Database (IMDb) metamodel [1].

Next to query evaluation, multi-threading is also used for model transformation. In [165], Tisi et al. present a prototype of an automatic parallelization for the ATL transformation en-

1. http://www.imdb.com/interfaces

gine, based on task-parallelism. To do so, they just use a different thread for each transformation rule application, and each match, without taking into account concurrency concerns (e.g., race conditions).

3.1.3 Asynchronism

LinTra is a Linda-based platform for model management and has several types of implementation. First, on a shared-memory architecture (i.e., the same shared memory between processors, typically multi-threading solutions), LinTra proposes parallel in-place transformations [36] and parallel out-place transformations [34]. Both strategies have significant gains in performance, compared to sequential solutions. Nonetheless, shared memory architectures are fine for not too big models. Indeed, since the memory is not distributed, a too big model could lead to an out-of-memory errors. This phenomenon happens more concretely in an out-place transformation since two models are involved during the operation. The first prototype of distributed out-place transformations in LinTra, is presented in [34], and works with sockets for communicating the machines. This first proposal remains naive. That is why Burgueno et al. propose a more realistic prototype for transformations on distributed architecture [35]. But the use of a distributed architecture raises new questions: how to distribute data and, how to distribute tasks? They applied different strategies mixing both the evaluation of tasks on a single or on multiple machines, and storing the source and target models on the same, or on different machines. The study was conducted for the specific IMDb test case only, and then does not provide a general conclusion about the benefits of a such solution.

One can note from Table 3.1 that only two papers of the related work on parallelism in MDE offer detailed performance analysis according to the data or task distribution. However, both these papers clearly show that many factors can influence performance such as the size of models, their reading/writing modes (e.g., in-place), the distribution of the models and the distribution of the operations to perform on them and so on.

Avoiding computations

Incrementality and *laziness*, or *incremental* and *on-demand* computation, are the main strategies used in MDE for minimizing the sequence of basic operations needed to perform a query or transformation. They have been classified as strategies for reactive execution in [124], since they foster a model of computation where the model-management system

reacts to update and request events, (note that the term is only inspired by the reactive programming paradigm in the sense of [85], that we will not discuss here).

We classify existing applications of these strategies to model-management tools in the columns of Table 3.2, depending on their scope:

— **MQ** or **MT** if the strategy is applied to the whole model query or transformation;

— **Matching** if the strategy is only applied to the matching phase (the subgraph isomorphism of the pattern to query/transform, over the full model) of the model query/-transformation;

— **Collections** if the strategy is only applied to the computation of collections during the query/transformation.

Table 3.2 – Reactive strategies for model-management in literature.

	MQ	**MT**	**Matching**	**Collection**
Incrementality	[41]	[112]	[20, 171]	
Laziness	[164]	[166]		[31, 174]

Incrementality

Incrementality is an event-based pattern, whose goal is to reduce the number of needed operations when a change happens within the input model. Instead of applying from scratch the whole set of operations on the new input model, incrementality allows the system to apply only the operations impacted by updates. Since the system needs to apply a subset of operations, a trace to relate the output pieces to input elements is necessary. The approach leads then to an additional memory cost, with a good trade-off only if changes occur often enough.

To achieve incremental execution of transformation rules, Calvar et al. designed a compiler to transform a code written with ATL [112]. The output program takes advantage of active operations of the language. The active mechanism works as an observer pattern: the values are defined as mutable, and changes are notified to an external observer. From there, it is easy to isolate what part of the model has been changed, and then to deduce what rules must be operated again. To illustrate their proposal, they applied their evaluation to two cases including social media models to illustrate the efficiency of the strategy for querying models that have strong user activity. This is not the single attempt of integrating incremental aspects in ATL.

In [41], Cabot et al. present an incremental evaluation of OCL expressions that are used to specify elements of a model in ATL. They used such an approach to state the integrity preservation of models at runtime. Instead of testing the whole integrity of a model every time it is changed, the proposed system is able to determine when, and how, each constraint must be verified.

For example, the Forgy's RETE algorithm [74] for pattern matching, presented in [171], constructs a network to specify patterns and, at runtime, tracks matched patterns. Instead of matching a whole pattern, the RETE algorithm will match the subparts of the pattern until getting a full match.

Research efforts have used incrementality to update the incomplete patterns in the use of the RETE algorithm, without fully recalculating the matching for all the present candidates. In MDE, the Eclipse VIATRA framework has an implementation of the RETE algorithm to achieve an incremental pattern matching [21, 86, 20]. The choice of using an incremental algorithm is due to the focus of the tool. Indeed, the VIATRA platform focuses on event-driven and reactive transformations thus an efficient solution, for handling multiple changes, has been chosen.

Laziness

Laziness is also commonly used by model management tools. In general, laziness reduces computations by removing the ones that are not needed to answer the user requests. Indeed, by using laziness, pieces of output are calculated only when they are needed by the user. This "call-by-need" approach is mainly used on big models, known as Very Large Models (VLMs). Since users may want to get only a part of the output, computing the whole query/transformation is unnecessary.

In [166], Tisi et al. extended the model transformation mechanism of ATL with laziness. Elements of the target model are firstly initialized, but their content is generated only when a user tries to access it. To do so, the model navigation mechanism has a tracking system, which provides, for a target element, the rules that must be executed. In addition, the tracking system keeps the information about already executed rules to avoid recomputation. Other engines, such as ETL (Epsilon Transformation Language), from the Epsilon framework, implements a similar approach [2].

Besides model transformation, laziness is also used in model querying. In [164], Tisi et al. redefine OCL features with laziness aspects. For instance, operations of the language

2. https://www.eclipse.org/epsilon/doc/etl/

are redefined to be evaluated with a lazy strategy. Also, the work proposes lazy collections that respect the OCL specification. The latter is similar to the collections proposed by Willink in [174]. The OCL collections are implemented as generic Java classes, with lazy operators. These approaches aim at tackling OCL related efficiency issues. For example, because of the OCL collections are immutable, the successive addition of elements in a collection would create intermediate data structures. More generally, the composition of operation calls would cause an evaluation of a cascade of operations. The proposed implementation of a lazy evaluation optimizes such common cases.

3.2 Semantics and correction in MDE

Reasoning on programs is an active field of research. On one hand, work has been done for reasoning on transformation semantics, either automatically or interactively. On the other side, research efforts have been dedicated to proving properties on parallel programs. In this section, we both investigate approaches that have been designed for reasoning on model transformations(Section 3.2.1), and in a more general context, on distributed computing (Section 3.2.2). We want to highlight that the goal of our work is to bridge the gap between certified model transformations and data analytics frameworks, such as Spark.

3.2.1 Correction for model transformations

Automatic proving approach In [40], Buttner et al. provide an automatic translation of ATL transformations into OCL as a first-order semantics for model transformations. Using these semantics, transformation correctness can be automatically verified [39] with respect to non-trivial OCL pre- and postconditions by using SMT solvers (e.g, Z3). This is not the only attempt of automatic proofs of ATL transformations based on solvers. In [46], Cheng et al. present a translation validation approach to encode a sound execution semantics for the ATL specification. Similarly to Buttner approach, they verify an ATL specification against the specified OCL contracts. To demonstrate their approach, they have developed the VeriATL verification system using the Boogie2 intermediate verification language, which in turn provides access to the Z3 theorem prover. They extend their work by developing a formalization for EMFTVM [47], the research VM included in ATL, bytecode. This work target MT language not having an implementation but having a well-defined execution semantics. Finally, Oakes et al. propose a method for verifying ATL model transformations by translating them

into DSLTrans [132], a transformation language with limited expressiveness. Pre- postcondition contracts are then verified on the resulting DSLTrans specification using a symbolic-execution property prover.

Interactive theorem proving The techniques that are presented above are based on an automatic proving mechanism. But proving is not a trivial task, and might require human expertise that cannot be automated. In [42], Calegari et al. encode ATL model transformations and OCL contracts into Coq types, propositions and functions to interactively verify whether the transformation is able to produce target models that satisfy the given OCL contracts. Stenzel et al. propose a Hoare-style calculus, developed in the KIV prover, to analyze transformations expressed in (a subset of) QVT Operational [154]. UML-RSDS is a tool-set for developing correct-by-construction model transformations [111]. It chooses well-accepted concepts in MDE to make their approach more accessible to model transformation developers. Once the development is achieved, transformations are verified against contracts by translating both into interactive theorem provers. Poernomo et al. use Coq to specify model transformations as proofs and take advantage of the Curry-Howard isomorphism to synthesize provably correct transformations from those proofs [140]. Their approach is further extended by Fernández and Terrell, who use co-inductive types to encode bi-directional or circular references [71]. None of these research efforts addresses proving the equivalence of the sequential and the distributed executions of a transformation.

3.2.2 Proving parallel programs

In this thesis, Chapter 6 proposes a correct-by-construction distributed transformation engine on top of Spark. The extraction of the parallelizable semantic into executable Scala code is proceeded by hand. One can argue that we could have chosen any other back-end language or framework instead of Spark. In particular, some back-ends automatically extracted from proof assistant, such as Coq, which would enhance the automatic certification of our pipeline. For instance, we could have chosen to extract Haskell code from CoqTL and then use the *Haskell distributed parallel Haskell* (HdpH) language to introduce parallelism and distribution. Similarly, we could have extracted OCaml code from the CoqTL specifications and then use the BSML [115] library for parallelization. Note that the optimizations for enhancing parallelism introduced in this thesis, and proven equivalent to the initial Coq specification, would be useful for any back-end that introduces parallelism and distribution. Another approach for obtaining a certified distributed engine could be to specify the engine

using the formalism of an existing distributed solution. Several works are based on a deep embedding of parallel languages or libraries: the syntax and semantics of such languages are modeled using a proof assistant. If it is convenient to have such a formalization to reason about meta properties of the considered language, it is less convenient to write programs than using a shallow embedding as it is done in SyDPaCC [114]. For example, Grégoire and Chlipala provide a small parallel language and its semantics and prove correct optimizations of stencil based computations [81]. A subset of Data Parallel C has been formalized using the Isabelle/HOL proof assistant [64]. The tool generates Isabelle/HOL expressions that represent the parallel program rather than actual compilable code. The dependent type language Agda is used by Swierstra to formalize mutable explicitly distributed arrays. He uses this formalization to write and reason about algorithms on distributed arrays: a distributed map, and a distributed sum. It is, of course, possible to reason about distributed collections, such as RDDs, and consider their distribution using the formalization of BSML in Coq. SyDPaCC however allows for the extraction of parallel code, but it does not support mutable data structure.

3.3 Multi-parameter and benchmarking

The configuration of applications is designed to enhance the performances of applications. In a MDE context, it is mostly used to choose the relevant strategy according to a use case. As illustrated in Sect 3.1, there exist many strategies for computing queries and transformations. While most of the solutions only implement one single strategy, without any configurable aspects, there exist analyses of model management solutions, based on properties, to either choose the right software solution, or the right algorithm within a single application.

3.3.1 Feature models in MDE

In [156], Tamura et al. propose a comparison of taxonomies for MT languages based on classification schemes and highlighted the need of having formalism. Existing work followed this approach and proposed different attempts of formal classifications for model-management operation and more specifically for model transformation. In [5], Amrani et al. attempt to build a catalog of model transformation intents that describes common uses of model transformations in MDE and the properties they must or may possess to face case-

related challenges. Previous work more focused on the characterization of model transformation languages. Czarnecki et al. proposed feature diagrams to describe the taxonomy of model transformation approaches [59, 58]. These work aims at explaining the different design choices of model transformations to propose categories for classifying use-cases. In [101, 102] Kahaniet al. classifies MT tools over 6 clear categories of model-to-text and model-to-model transformations: general, model-level, transformation style, user experience, collaboration support, and runtime requirements.

Studies have shown that the performance of transformations is relevant but there is a lack of support for transformation developers without detailed knowledge of the engine to solve performance issues [83]. In [82], Groner et al. propose a study based on performance following some main factors: the execution time, the size of the models used, the relevance of whether a certain execution time is not exceeded in the average case, and the knowledge of how a transformation engine executes a transformation.

The work mentioned before are very generic, and encompass all kind of model transformations. Other approaches use a different granularity to focus on a more specific category of model transformation engine. For instance, [143] only focuses on languages for model-to-text transformations. In [89], Hidaka et al. clarify and visualize the space of design choices for bidirectional transformations, in the form of a feature model. Finally, [173] aims at generalizing compositions of transformation using two internal composition mechanisms for rule-based transformation languages: module import and rule inheritance.

3.3.2 Multi-strategy based MDE tools

Bergmann et al. have developed the EMF-IncQuery Framework [22], an industrial tool to compute declarative queries over EMF models, as a part of the model transformation framework VIATRA. The inputs of rules are obtained from a query evaluation that finds matched patterns within a given model. In [19], they propose two strategies for evaluating pattern matching. The first is local-based graph pattern matching which starts the matching process from a single node and extends it step-by-step by neighboring nodes and edges. The second is an incremental solution: patterns are explicitly stored and incrementally maintained upon model manipulation. The second solution provides significant memory but increases memory cost. In [93], Horvath et al. extend initial measurements carried out in [19] to assess the effects of combining local search-based and incremental pattern matching strategies.

CHAPTER 4

PROGRAMMING MODELS FOR EXECUTING DISTRIBUTED MODEL QUERIES

Contents

Several choices are made in the design of an execution engine. The programming model of expressions is one of the most important, since often most of the computation of a rule is used for expressions. In this chapter we illustrate the variability of existing programming models, and emphasize the need for a multi-strategy vision for model-management where strategies can be automatically switched and combined to efficiently address the given model-management scenario. Furthermore, we stress the need for automatic choice and configuration of strategies to enhance performance of LCDPs. We outline code-related challenges such an approach and provide hints for technical solutions to these problems.

4.1 Expressions in Model Transformations

A model transformation (MT) definition is composed by rules. Each of them is composed by two distinct parts: the left-hand side (LHS) defines what must be matched, and the right-hand side (RHS) what to produce considering the former. This very generic definition can be found in several level of programming. For instance, the Bird-Meertens formalism [30, 126, 79], proposing equivalence rules between expressions is a basis of term rewriting used in the optimization of programs [116]. In a MDE context, rules are used to modify an instance of a model. The LHS of a MT rule is more than a syntactic condition, as it can be in terms rewriting. Indeed, it matches a set of elements, along side a condition on these elements and the model itself. This expression, that can be considered as a model query, and must be computed by the engine, and be evaluated into a boolean value. In the other side, the RHS describes how are defined a set of new elements. Expressions are used for constructing each of these new elements.

The adopted strategy for running expressions follow one or several programming models. While most model-management languages implement a single execution strategy for evaluating an expression, with specific strengths and weaknesses depending on the use case, the diversity of strategies that have been employed poses several scientific challenges. These techniques range from implementing specific execution algorithms (e. g., RETE [171]) to compiling toward distributed programming models [107] (e. g., MapReduce [65]). These programming models are sometimes qualified as *paradigms* in the literature, but this term may lead to confusion with programming paradigms (functional, logic, etc.). Some existing solutions in MDE offer more than a single execution strategy but the choice is left to the user which requires expertise. Moreover, it appears that performance for some use cases could be improved by the combination of different strategies, e. g., after decomposing the model-management operation. Furthermore, evaluating expressions on very-large models (VLMs) is challenging. To improve efficiency and scalability, recent research on model-management studied parallel and concurrent programming as well as specific execution models for model-management languages. As explained in [136], developing distributed program is challenging, because of the non-deterministic aspects of the computation. The consequence of the difficulty to write parallel programs is a lack of parallel programmers and programs remain error-prone. To tackle these difficulties, there exist programming models to help developers to write expression that will be evaluated in parallel. Apache Spark proposes several distibuted programming models including MapReduce [65], and its vertex-centric based ap-

proach named Pregel [122]. In this chapter, we illustrate the variability of existing distributed programming models, and emphasize the need for a multi-strategy vision for model-management, where strategies can be switched and combined to efficiently address the given model-management scenario. Furthermore, we stress the need for automatic choice and configuration of strategies to enhance performance. We outline code-related and DevOps challenges of a such approach and provide hints for technical solutions to these problems.

4.2 Motivating Example

Social network vendors often provide specific development platforms, used by developers to build apps that extend the functionality of the social network. Some networks are associated with marketplaces where developers can publish such apps, and end-users can buy them. Development platforms typically include APIs that allow analyzing and updating the social network graph.

As a running example for this chapter, we consider a scenario where a vendor adds a low code development platform (LCDP) to allow end-users (also called *citizen developers* in the LCDP jargon) to implement their own apps. Such LCDP could include a WYSIWG editor for the app user-interface, and a visual workflow for the behavioral part. In particular, the LCDPs would need to provide mechanisms, at the highest possible level of abstraction, to express expressions for updates on the social graph.

In Fig. 4.1 we show the simple metamodel for the social graph that we will use in the chapter. The metamodel has been originally proposed at the Transformation Tool Contest (TTC) 2018 [76], and used to express benchmarks for model query and transformation tools. In this metamodel, two main entities belong to a `SocialNetwork`. First, the `Posts` and the `Comments` that represent the `Submissions`, and second, the `Users`. Each `Comment` is written by a `User`, and is necessarily attached to a `Submission` (either a `Post` or another `Comment`). Besides commenting, the `Users` can also like `Submissions`.

As an example, in this chapter we focus on one particular query, also defined in TTC2018: the extraction of the three most debated posts in the social network. To measure how debated is the post, we associate it with a numeric score. The LCDP will have to provide simple and efficient means to define and compute this score. In a real-world context, finding the most active and debated posts help identifying trends (e. g., a Twitter [1] key concept), understand the audience, and the evolution of a social network.

1. `http://www.twitter.com`

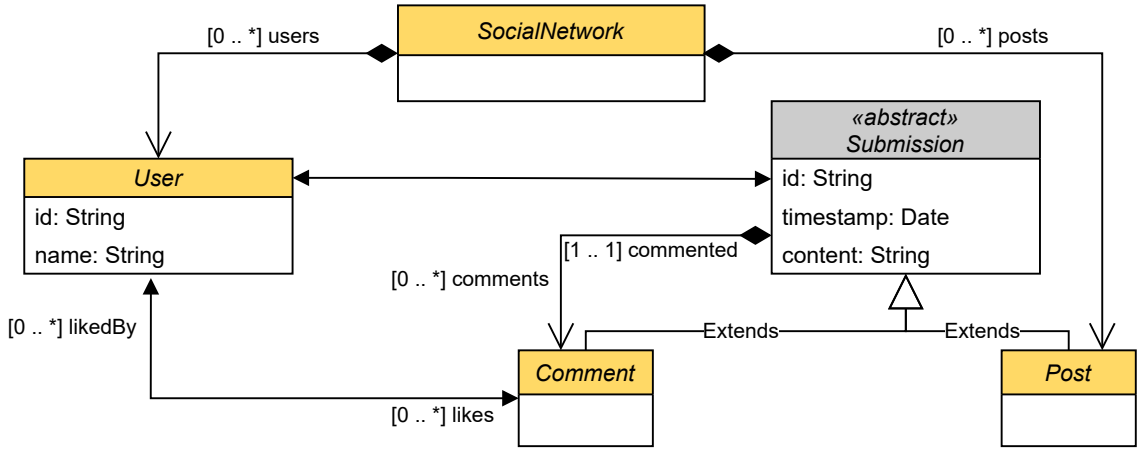


Figure 4.1 – The metamodel of a social network (TTC 2018)

We suppose the vendor to include a declarative query language for expressing such computation on the social graph, and storing scores as a derived properties of the graph (i.e. new properties of the social graph that are computed on demand from other information in the graph).

In Listing 4.1 we implement the query to get the top-three debated posts in a model conforming to the presented metamodel, using the formula defined in TTC2018. The query is written in OCL, the most used declarative query language in MDE. In particular we use the ATL flavor of OCL.

In this code, a score of 10 is assigned to the post for each comment that belongs to it. Comments belong to a post in a recursive manner: a comment belongs to a post if it is attached either to the post itself, or to a comment that already belongs to the post. Then, a score of 1 is also added every time a belonging comment is liked.

The query is defined using three (attribute) helpers, that can be seen as derived properties. The first helper, `allComments` (line 7 to 11), collects recursively all the comments of a `Submission`. The second helper, `countLikes` counts how many times a comment that belongs to the given post has been liked. Then, the `score` of a `Post` is calculated by summing the result of `countLike` and the number of its belonging comments multiplied by ten. Finally the top three posts are obtained by the query `topPosts` sorting the posts by decreasing score, and selecting the first three.

The simple declarative query in listing has not been defined with efficiency concerns in mind. Indeed, since we cannot make assumptions on the background of citizen developers, our LCDP cannot presume that they will structure the query for satisfying any performance

Listing 4.1 – An OCL query for the first task of the TTC 2018.

```
1  query topPosts =
2      SN!Post.allInstances()
3            →sortedBy(e | -e.score)
4            →subSequence(1, 3);
5
6  helper context SN!Submission def : allComments =
7      self.comments
8          →union(self.comments
9                     →collect(e | e.allComments)
10                    →flatten());
11
12 helper context SN!Post def : countLikes =
13     self.allComments
14         →collect(e| e.likedBy.size())
15         →sum();
16
17 helper context SN!Post def : score =
18     10*self.allComments→size() + self.countLikes;
```

requirement. As a result, when the number of users increases, soon the size of the social graph makes the computation of this query challenging. First of all, the list `Post.allInstances()` (line 2) becomes too large to manipulate. Especially the full sorting of posts (line 3) seems prohibitive. Without an efficient mechanism, the naive recomputation of `allComments` each time it is called, is a further performance waste. If we consider the typical frequency of updates for social network graphs, keeping the list of top posts up-to-date by fully recomputing this query at each update could consume a significant amount of infrastructure resources.

Moreover, the most efficient way to execute the query does not depend only on the given query definition and metamodel structure, but on several characteristics of the usage scenario. A technique to optimize a particular use case typically has significant overhead in other use cases. Factors that can influence this choice in our example can be related to the model size (e. g. order of magnitude for the number of `Users`), frequency of updates (e. g. of new `Submissions`), average model metrics (e. g. average number of `Comments` per `Post`), acceptable response time for the final query (`topPosts`), infrastructure constraints and resources (e.g. available memory, CPUs) and so on. In some cases techniques can be combined, further complexifying the search for the optimal solution.

Finally, while in this chapter we will focus exclusively on this example, it is not difficult

OCL	Spark
`constant`	`constant`
`variable`	`variable`
`e op e`	`e op e`
`e.field`	`e.field`
`c.foo(e,...,e)`	`c.foo(e,...,e)`
`head::tail`	`head::tail`
`if cd then e1 else e2 endif`	`if (cd){ e1 } else { e2 }`
`let v1 = e1:T1 in e`	`val v1:T1 = e1 ; e`
`e | foo(e)`	`e => foo(e)`

Table 4.1 – Equivalence between OCL and Spark expressions

to identify similar issues for update operations (e. g. removal of all information for an unsubscribing user) or transformation (e. g. for storing the graph in a particular persistence format).

4.3 OCL expressions in Spark

The basics of the OCL expressions can be directly expressed using simple Scala and Spark features. In this section, we present Socle, an equivalence between the OCL expression language and its Spark direct translation. The full work made on Socle can be publicly found online [2]. About types, primitive types are a copy paste from OCL definitions, using Scala types from its standard library. Collections can either be expressed as Scala collection, implementing the semantics of specific collections on top of Scala sequence, or using Spark RDDs. While the former solution proposes several semantic on the top of the same Scala structure, allowing operations on a mutable collection, Spark RDDs are immutable and do not give control on how the operation are computed. In this section, we describe how Scala `Seq` and Spark `RDDs` can be used as OCL collections, both as OCL `Bag`, that is a not-ordered collection allowing duplicates. However, an attempt of proposing the semantics of all the specific OCL collections also exists in Socle.

In the following, we consider models being either (i) sequential, defined as a EMF model, or (ii) distributed, defined with a Spark graph as couple of a RDD of vertices, and a RDD of edges.

2. https://github.com/JolanPhilippe/Socle

OCL	Scala	Spark
`c.collect(expr)`	`l.map(expr)`	`rdd.map(expr)`
`c.sortedBy(expr)`	`l.sortBy(expr)`	`rdd.sortBy(expr)`
`c.subSequence(i, j)`	`l.slice(i, j)`	undefined
`c1.union(c2)`	`l1 ++ l2`	`rdd1.union(rdd2)`
`c.flatten()`	`l.flatten`	`rdd.flatMap(identity)`
`c.size()`	`l.size`	`rdd.size`

Table 4.2 – Equivalence between OCL and Spark expressions

OCL syntax As illustrated in 4.1, the OCL statements are very similar to the Scala ones. Expressing basic OCL expressions is then very fluent in Spark.

OCL primitve types Scala primitive types are largely enough for naturally use OCL types. For all these types, there exist a transparent equivalence between the used functions in OCL and their call in Scala.

— `Integer` with `scala.Int`;
— `Real` with `scala.Double`;
— `Boolean` with `scala.Boolean`;
— `String` with `scala.String`;
— `Enumeration` with Scala `enum` types;
— `TupleType` with the set of Scala classes for tupling: `Tuple2`, `Tuple3` and so on.

OCL collections OCL proposes different collections for storing elements with different semantics. For instance, OCL `Set` is a not ordered collection without duplication while a `Sequence` is ordered and allow duplication. In this part of the work, we consider the used structures as `Set`. To do so, we give a Spark equivalence for a subset of OCL collections in 4.2, where `c` represents a OCL collection, `l` a Scala `List` and `rdd` a Spark RDD. The table only gives an equivalence for the OCL operations used in 4.1.

Additional expressions OCL proposes additional functions to handle a model that we need equivalences on our model implementation:

— a direct access to elements of a given type (with `allInstances`);
— an access to the target of a relation using it source and the type of relation.

In a Spark graph, the former is defined with a filtering function on the `vertices`, while the second is a filter on the graph `triplets`, i. e. the set of edges defined as tuples composed

by the source, the target and a label.

TTC18 OCL query With the all proposed equivalences, a proposition of a direct implementation of the OCL helpers (Listing 4.1) is presented in Listing 4.2. Thanks to the transparency between `List` and `RDD`, the same definition can be operated on either a Scala or a Spark model. The main difference is how elements and relations are accessed, depending on their type. However, Spark does not allow nested recursive definition. That is why, a function `traversal` must be defined to recursvively obtained all the comments of a given submission.

Listing 4.2 – Spark implementation of OCL query for score.

```
def allComments(p: Post): List[Comment] = {
    def traversal(c: Submission): List[Comment] = {
        var res: List[Comment] =
            c match {
                case comment: Comment => List(comment)
                case _ => List()
            }
        for (comment <- getComments(c, model)) {
            res = res ++ traversal(comment)
        }
        res
    }
    traversal(p)
}

def countLikes(p: Post): Long = {
    allComments(p).map(comment => getLikedBy(comment, model).size).sum
}

def score(post: Post): Long = {
    10 * allComments(post).size + countLikes(post)
}
```

4.4 Multi-Strategy Model Management

Each of the research efforts presented in Tables 3.2 and 3.1 exploit a single strategy for optimizing model-management operations. Typically, the strategy is applied in an additional implementation layer for the model-management language, e.g. an interpreter or compiler.

We say that a query or transformation engine performs **multi-strategy model management** if it automatically considers different strategies, instead of a single one, in order to manipulate models in an efficient way. According to Chap. 3 and to the best of our knowledge, such approach does not exist in the literature.

In this section, we exemplify the multi-strategy approach by implementing the OCL query of Listing 4.1 in different ways, using different strategies of parallelism. The goal of this Section is not to provide the most efficient solutions for solving the given problem. Instead, it aims at illustrating the diversity of solutions, that each have its own advantages depending on the use cases. To do so, we implemented several solutions using different parallel strategies and compared them. Also, this section only illustrates the variability of single solution, and not their possible combination.

Our prototype is built on top of Spark [3], an engine designed for big data processing. In addition to parallel features of Spark on data structures, called Resilient Distributed Datasets (RDDs), the Scala implementation of Spark proposes several APIs including a MapReduce-style one, an API for manipulating graphs (GraphX [80] that embeds the possibility to define Pregel programs), and a SQL interface to query data-structures. Because the framework proposes different parallel execution strategies, we only focused on parallel approaches to illustrate the need of a multi-strategy approach. Comparing solutions that include laziness and incrementality aspects is a part of our future works. In our implementation example, we use GraphX, in addition to its provided Pregel function, and MapReduce features. We represent instances of `SocialNetwork` as a GraphX graph where each vertex is a couple of a unique identifier and an instance of either a `User` or a `Submission` (`Comment` or `Post`). Edges represent the links of elements of a model conforming the meta-model presented in Figure 4.1, labeled by a String name. We keep exactly the same labels from the meta-model for [0..1] or [1..1] relations but we use singular names for [0..$*$] relations (e. g., one edge "like" for each element of the "likes" relationship). For the rest of this section, we consider `sn` a GraphX representation of a `SocialNetwork`.

Considering that there exists an implementation for the function `score`, that will be de-

3. `https://spark.apache.org/`

tailed later in this section, the OCL query `topPosts` of Listing 4.1 can be rewritten using Spark, as presented in Listing 4.3.

Listing 4.3 – Spark implementation of a query from TTC 2018.

```
1 sn.vertices.filter(v => v.isInstanceOf[Post])
2     .sortBy(score(_._2), ascending=false)
3     .collect.take(3)
```

First, the `SN!Post.allInstances()` statement of the OCL specification is translated into the application of a filtering function on the vertices of the graph `sn` (line 1). A sorting with a decreasing order is then applied to the score values (computed by the `score` function) of each vertex. The projection `_._2` returns the second element of the vertex values, that is an instance of `Post`, while `_._1` would have returned its identifier within the graph. At the end of line 2, the current structure is still a RDD. Because of the small number of values we aim at finally obtaining, the structure is converted into a sequential array of values (function `collect`), from which we get the first three values. We can notice the similar structure between the Spark and OCL queries. Hence, the global query can almost be directly translated from one language to the other. However, the scoring function can be implemented in many different ways with many different strategies. We illustrate this through three implementations in the rest of this section: *direct-naive*, *pregel*, and *highly-parallel*. Then we discuss these three implementations and open to the multi-strategy approach.

4.4.1 Direct naive implementation

The first implementation, namely *direct-naive*, shown in Listing 4.4, directly follows the OCL helpers from Listing 4.1. The first auxiliary function `countLikes`, corresponding to the homonym helper, sums the number of `"like"` relations for each comment of a given post (lines 17 to 21). The second auxiliary function `score` (lines 23 and 24) is also a direct Spark translation from the OCL query. It uses parallelism, coupled with the lazy evaluation provided by Spark. Indeed, the execution of operations on RDDs is not started until an action is triggered. In our example, `collect` and `count` are these actions. Finally, the `allComments` function is defined recursively using GraphX features. The direct-naive implementation of `score` uses three functions that are inspired by functional languages: `filter` which removes all the elements of a list that do not respect a given predicate; `map` that applies a function to

Listing 4.4 – Direct implementation of score.

```
1
2  def allComments (p : Post) = {
3    // recursive function
4    def getComments (co : Comment) : List[Comment] =
5      List(co).union(sn.triplets
6          .filter(t => t.srcAttr == co
7                       & t.attr == "comment")
8          .flatMap(_.dstAttr compose getComments).collect)
9
10   sn.triplets
11       .filter(t => t.srcAttr == p & t.attr == "comment")
12       .flatMap(_.dstAttr compose getComments).collect
13 }
14
15 def countLikes (p: Post) =
16   allComments(p)
17     .map(c => sn.triplets.filter
18                 (t => t.attr == "like" & t.dstAttr == c)
19         .count).sum
20
21 def score (p : Post) =
22     10 * allComments(p).size + countLikes(p)
```

every element; and `flatMap` which is a composition of `map` and `flatten`. The latter is equivalent to `flatten` from Listing 4.1. The implementation first gets the direct comments of a post (lines 10 and 11), and, using an auxiliary function `getComments`, recursively gets all the belonging comments (lines 13 and 14). The method `flatMap` of lines 8 and 13 transforms the list of lists, into a list of comments.

4.4.2 Pregel implementation

The second solution, namely *pregel*, proposed in Listing 4.5, is a Pregel-based implementation. The main idea of this solution is, starting from a `Post`, counting the number of comments and the number of likes for these comments by propagating messages through edges of the graph by using Pregel. To do so, we declare two variables, `nbComments`, and `nbLikes`, that can be seen as aggregators, i.e., global accumulator of values. The propagation is processed using the Pregel support of GraphX that works as follows. At each iteration, the function $mergeMsg$ accumulates into a single value the incoming messages (line 20), that are

stored in a iterable structure, from the previous iteration (with an initial message defined for the first iteration). This value is used by $vprog$ with the previous vertex v_n to generate the new vertex data v_{n+1}. With this value, messages are generated with $sendMsg$ and sent to vertices through edges for the next iteration. Because the stucture which stores the message must be iterable, all the messages must be of type Iterator. An empty message is then produced by `Iterator.empty`. The program stops when no message is produced for the next iteration. In our implementation, messages are tuples of two values. The first one is an integer value for specifying which vertices should compute and send messages. Besides, if the integer is negative, then all the vertices should compute. The second one aims at precising what value must be incremented (either the number of comments (`false`), or likes (`true`)). The initial step of the execution questions the model to get the id of the vertex containing the `Post` we want to score (line 3 and 4). This identifier is added to every vertex to provide them a global view on the computation status (line 6). Then, messages are propagated through the edges to belonging comments of computed vertices, or to the users who likes the scoped comment (line 13 to 19). At the message reception, the computation will increment the aggregator according to the second value of the message (line 9 to 11). After the execution of the `pregel` function, a score value is calculated using `nbComments` and `nbLikes`.

4.4.3 MapReduce implementation

Listing 4.6 illustrates a solution with a higher level of parallelism, namely *highly-parallel*, that uses a MapReduce approach. The purpose of this third solution is to process as much as possible operations in parallel in a first time, and then go through the graph to reduce these values. The first step counts the number of direct sub-comments, and the number of likes, for each element of the model, using a map and reduce-by-key composition (line 3 to 7). Because the number of likes has not the same importance than the number of belonging comments in the score calculation, two keys are created for a single element: one for counting each property (i.e., number of comments and number of likes). Then a graph-traversal operation calculates the total number of belonging comments and likes for a given post. However, the keys are only created if a comment, or a like, exists. Then, to initialize values, we use a composition of `find` that returns an option, and `getOrElse` in the case of the absence of the key. The latter returns the value of the option if it exists, and a default value otherwise We do not expect to gain performances with this approach because the operations are not costly enough. However, having a highly parallel approach largely increase the scalability of the program.

Listing 4.5 – Pregel implementation of score.

```
def score(p: Post) = {
  var nbComment, nbLike = 0L // Aggregators
  val fstId = sn.vertices
                .filter(v => v._2 == p).first._1

  sn.mapVertices((_, v) => (fstId, v)).pregel
    (initialMsg = (fstId, false))
    (vprog = (id, value, merged) =>
      if (merged_msg._1 == id || merged_msg._1 < 0)
        if (merged_msg._2) nbLike += 1L
        else nbComment += 1L
      (merged_msg._1, value._2),
    sendMsg = t =>
      if (t.srcId == fstId | t.srcAttr._1 == -1L)
        if (t.attr == "comment")
          Iterator((triplet.dstId, (-1L, false)))
        if (t.attr == "likedBy")
          Iterator((t.dstId, (-1L, true)))
      Iterator.empty,
    mergeMsg = (m, _) => m)

  10 * nbComment + nbLike
}
```

4.4.4 Discussion on multi-strategy

First, the complexity of the solutions *direct-naive* and *pregel* can be compared. On the one hand, the complexity on time of the direct implementation of the OCL query, can be given as the sum of the complexity of `allComments` and `countLikes`. Considering n the number of nodes, these two complexities are defined as follows. First, `allComments` is a depth-first search of complexity $O(n+m)$ with m the number of `comment` edges (i.e., the depth of belonging comments). Second, `countLikes` is composed by a depth-first search, and the `map` of a function whose complexity is $O(n)$. Then, the complexity of the mapping part is given by $O(n^2)$. Since the complexity of the `sum` operation is negligible, we do not consider it in the calculation of the global complexity. By summing these values, we obtain a complexity of $O(n^2+m)$ for the direct implementation of the scoring function. On the other hand, the Pregel implementation complexity is bounded by $O(n^2)$, in the case of all comments are all belonging to the same post. Naturally, the second solution will be preferred since its complexity is lower. However, if the model has a small depth of belonging comments (i.e., a small value for m), the two solutions are not significantly different.

The Pregel solution has nonetheless an important weakness. Indeed, for optimization reasons, $vprog$ is only applied to vertices that have received messages from the previous step. Then, considering the case where the comments are all commented once, the $vprog$ function will be applied to only one vertex. Hence, the parallelism level strongly depends on the number of siblings of each comment. With Pregel, only active vertices, i.e., vertices which received a message from the previous iteration, compute the `vprog` function. Thus, the number of operations concurrently executed in Pregel varies from the less to the most commented and liked element. On the contrary, the highly parallel implementation executes the processing operations on every elements of the model. In the latter, the parallelism level of graph-traversal has the same limitation than the Pregel implementation, but always process a less complex operation (i.e., a reduction as a sum of integer values).

The three above parallel approaches can solve the same problem, but their efficiency depends on external parameters. For executing the `topPosts` query, a multi-strategy engine would compile it to:

— the direct-naive implementation if the depth of belonging comments is small;
— the pregel solution if the environnement has few resources for parallelism;
— the highly-parallel solution if the score computation needs big calculation on the vertices themselves.

As mentioned at the beginning of the Section, our proposed solutions do not claim to

Listing 4.6 – Highly parallel implementation of score.

```
def score(p: Post) = {
  // number of likes and comments per element
  val scores = sn.triplets
      .filter(t => t.attr == "likedBy"
                 | t.attr == "comment")
      .map(t => ((t.attr, t.srcAttr), 1L))
      .reduceByKey((a,b) => a + b).collect

  def getScore(s: Submission) = {
    val default = ((_,_), 0L)
    var nbLike = // 0 if s is not liked
        scores.find(e => e._1._2 == s
                       & e._1._1 == "likedBy")
            .getOrElse(default)._2
    var nbComment = // 0 if s is not commented
        scores.find(e => e._1._2 == s
                      & e._1._1 == "comment")
            .getOrElse(default)._2
    // recursive call
    val subScores = sn.triplets
        .filter(t => t.srcAttr == s
             & (t.attr == "likedBy"
               |t.attr == "comment"))
        .map(_.dstAttr compose getScore).collect
    // sum of all score from belonging comments
    for (score <- subScores) {
        nbLike += score._1
        nbComment += score._2
    }
    (nbLike, nbComment)
  }
  val score_p = getScore(p)
  10L * score._1 + score._2
}
```

be the most efficient ones. They are based on three parallelism strategies to illustrate the variability of possible solutions for a given problem. Considering the all presented strategies of Section 3.1.3, a more robust solution could include reactive aspects. For this particular example, mixing incrementality and parallelism would avoid useless calculations when the score of a single post has changed. For instance, the independent scores could be calculated once using parallelism, and, when a change occur, use incrementality to avoid the recomputation of unchanged elements. Considering a possible deletion of a part of the model (e.g., deletion of a user, and then of all his posts, and comments), laziness could be incorporated to the solution, to only recompute potential new most-debated posts.

4.5 Challenges in Multi-Strategy Model Management

In the perspective of low-code platforms, a multi-strategy engine should be fully automated, from the automatic strategy selection to the automatic configuration and deployment on distributed infrastructures. Our approach is different from the multi-strategy approach proposed in [4] which is focused on languages and their salient features. The conception of a multi-strategy engine leads to many scientific challenges that we divide in two parts in the rest of this section: the challenges related to the code and the challenges related to DevOps.

4.5.1 Code-related challenges

A first scientific challenge that arises from the multi-strategy approach is the automatic and transparent selection of the most adapted strategy for a given model-management operation. The motivating example of Section 4.2 shows the large variability to take into account to make the right choice. We divide this variability in different properties that should be considered:

— properties on the input model: size, meta-model, topology, etc.;
— properties on the operation to perform: update, launch, request, read or write, the frequency of the operation, etc.;
— properties on the available infrastructures: type of frameworks compatible or already deployed on the infrastructure;

This variability results in a combinatorial choice that could be solved by using constraint programming, or by leveraging machine learning techniques to automatically learn how to

associate these properties together.

The second challenge related to the code aspect is that an initial code written with a MDE solution may need to be rewritten to follow the chosen execution strategy, while guaranteeing the same expected output. In other words, a code rewriting or code generation challenge is raised by the multi-strategy approach. In particular, formal semantics for the model-management engine may be of high importance to guarantee a correct output code [163].

As example, Listings 4.4, 4.5, and 4.6 described in Section 4.4 all present a different way of writing a code from the initial OCL solution shown in Listing 4.1. The complexity and the level of parallelism of the solutions have been discussed in Section 4.4.

4.5.2 DevOps-related challenges

In the context of low-code platforms, automatically handling the strategy selection and the code generation is not the only concern. Once generated, the code must be deployed and run on complex distributed infrastructures. These tasks should be as transparent as the previous code-related challenges. Hence, a first challenge is automatically handling the deployment of a set of model operations that potentially use different strategies, onto the associated infrastructures that could themselves be very heterogeneous (e. g. different public Cloud solutions such as AWS, or private Clouds, hybrid Clouds, etc.). This complexity should be handled at the LCDP level, which requires safe and efficient deployments [45, 54].

Furthermore, choosing a given strategy, often involves deploying code on existing frameworks or platforms that implement that strategy. For instance, when choosing the MapReduce (respectively Pregel) strategy, Hadoop [4] (resp. Giraph [5] or Spark [6]) should be used to benefit from efficient implementation. All these frameworks are highly configurable, e. g., MapReduce has more than one hundred parameters [109]). Because of their large number of parameters, finding their optimal configuration is a difficult problem. This additional layer of configuration represents an additional combinatorial challenge. Several solutions could provide a good trade-off. For instance, instead of providing a full configuration for the tools, which is very costly, a performance prediction built from configuration samples could be used. This solution has been adopted by Pereira et al. [2]. Another approach would be to make a full cost estimation but only considering critical parameters. For example, give the right level of parallelism by providing an approximation of the optimal number of mapper

4. `http://hadoop.apache.org/`
5. `https://giraph.apache.org/`
6. `http://spark.apache.org/`

	Dataset			
	#users	#posts	#comments	#likes
1	889	1064	118	24
2	1845	2315	190	66
3	2270	5056	204	129
4	5518	9220	394	572
5	10929	18872	595	1598
6	18083	39212	781	4770

Table 4.3 – Description of used datasets from TTC18

and reducer in a MapReduce job [7]. More formal approaches can also be used to estimate the cost of parallel programs (e.g., cost model for GraphX [108]), and compare the different solution using additional parameters such as hardware configuration (e.g., the bridging Bulk Synchronous Parallel cost model [170]).

Using formal approaches to estimate the cost of a parallel program such as BSP cost model [170] or Pregel cost estimation [108].

Finally, as for the combinatorial problem of choosing the right strategy, machine learning techniques could be adopted [123].

4.6 Evaluation

This section gives an overview of the different strategies we use to implement the query to solve the problem presented in Section 4.2. We experiment our five parallel implementations of the TTC18 query. The experiments have been conducted on a shared memory machine with a Intel Core i7-8650U having 8 cores at 1.90GHz and a memory of 32GB. The machine was running Ubuntu 16.04 LTS. We use Java 8, Scala 2.12 with Spark 3.1.0. The used model and theirs respective sizes is described in Table 4.3. The used models are described in 4.3: The smallest model is composed 2071 elements and 2163 links while the biggest is made with 58076 elements and 111197 elements. Models have been loaded using the NeoEMF I/O module, as presented in Appendix A. Each speed-up on Table 4.4 is the mean of a series of 30 measures.

The first observation we can made is a Spark solution does not suit with small models. The Spark overhead is constant for a given executing environment and the time for memory allocation grows lower than the time to compute on bigger datasets. Thus dealing with small

7. `http://wiki.apache.org/hadoop/HowManyMapsAndReduces`

Dataset	OCL (Scala)	OCL (Spark)	Pregel	MapReduce	OCL + Pregel	MapReduce + Pregel
1	**1x**	0.39x	0.36x	0.46x	0.44x	0.46x
2	**1x**	0.51x	0.68x	0.85x	0.66x	0.71x
3	**1x**	0.27x	0.35x	2.34x	0.15x	2.96x
4	**1x**	4.25x	5.21x	4.17x	4.68x	4.03x
5	**1x**	4.68x	2.83x	2.39x	1.97x	3.91x
6	**1x**	4.07x	4.12x	4.58x	5.17x	3.27x

Table 4.4 – Comparison of speed-ups of 5 Scala + Spark implementations with a direct Scala implementation based on different strategies for the TTC18 query

model does not necessitate a long enough computation for making worth the use of Spark (e.g., on dataset 1 and 2). Because shared chunks among nodes by Spark have a constant size, additional experiments have shown that running expression evaluation on models with very few elements can lead to an approximate speed up of 1 for the OCL Spark implementation. Since these results are not certain, and obtain results not constant enough, we have decided to not show them here.

Secondly, we see that there is no single strategy outperforming the others. Evaluating the expressions on the 3rd dataset highlights MapReduce-based solutions as the best. Moreover, they are the only ones which lead to a speedup while the other largely increase the computation time. In another case, like for the 4th and 6th datasets, the evaluated performances are closed to each other, showing that, for these cases, all solutions show equivalent speedup. Finally, using the basic Spark implementation of the OCL query on the 5th dataset is the best solution.

4.7 Conclusion

In this chapter, we made an overview of what, and how, execution strategies can be used in expressions. In the context of developing low-code platforms for managing models, these strategies might be used for optimizing performances. However, a wrong use of a programming model can have a bad impact on calculation efficiency. The motivating example presented in Section 4.2 and the implementations of Section 4.4 illustrate that by using different strategies and different combinations of computational models for a given input model, different advantages could be observed, such as complexity, and parallelism level. Different computational models may be chosen, according to different properties: the type of input

model, its size, its topology, the type of computation to perform, and the available infrastructure. The future goal of a such prototype is to drive a complete study of how expressions can be used and combined, and to classify them depending on use cases.

CHAPTER 5

SEMANTICS FOR EXECUTING CERTIFIED MODEL TRANSFORMATIONS ON APACHE SPARK

Contents

The existing engines for performing model transformations are designed from different choices according to their purpose. We want to compare different design choices in transformation engines but making it using existing engines is not possible. There are too many differences that can impact performance. Hence we have developed a new engine, namely,

SparkTE for CoqTL, a transformation language defined with Coq. In this chapter we present the architecture of the engine and evaluate three design choices.

5.1 Introduction

In model-driven engineering, model management frameworks propose dedicated languages to transform models, like the AtlanMod Transformation Language [99] (ATL) or the Epsilon Transformation Language [106] (ETL). Good scalability and facilities for formal reasoning are among the intended benefits for users of model-transformation languages. Researchers have proposed transformation engines designed to effectively perform computationally or memory-intensive transformations [105]. For instance, transformation languages have been equipped with implicit parallel/distributed modes of execution to automatically multiply the number of resources allocated to a transformation [16, 106]. To achieve this, some engines have been built on top of distributed programming frameworks (e. g. Apache Hadoop in [16]). However, comparing these different solutions does not make sense. First of all, the main goal is not the same. One can address scalability while the other is focused on how deal with frequent modifications. Also, the used technology differs (the choice of language, or running environment), making biased comparisons. Moreover, the community has extensively worked at formal reasoning and verification tools for model transformation languages. Among these solutions, the CoqTL language [163] allows users to write transformation rules, define contracts and certify the transformation against them, within the Coq proof assistant [158]. Coq programs are not designed for being executed. They are only based on formalism. They represent a good starting point for comparing different execution models. These executions are specified with different semantics, independently of any technologies. This pure proposal allows a comparison between semantics themselves. Indeed, independently of the execution environment, proposing different pure semantics that proven equivalent, allow a comparison and highlights the impacts of how are defined a transformation engine.

In this chapter, we introduce a transformation engine that addresses at the same time distribution and certification, with different execution semantics. This is not trivial. Distributed programming typically involves non-deterministic execution order, complicating the proof of properties on parallel programs. Interactive theorem proving is already a very costly activity on sequential model transformations. Certifying transformations by formal reasoning on the complexities of modern distributed frameworks would require unmanageable proofs.

We propose a solution to allow users to certify transformations in Coq and execute them over a modern data-analytics framework. The core of our proposal is SparkTE, a distributed implementation of the CoqTL specification, built on top of Apache Spark, a widely-deployed framework. Besides scalability, Spark provides an ecosystem of libraries, e. g. for connection to heterogenous data sources. Importantly, the Spark programming interface mirrors a functional/higher-order model of programming. This is a key property for our solution, since it enables a straightforward and high-confidence extraction of parallel code from functional Coq specifications.

Notice that we designed the parallelizable specification for a specific data analytic framework, and targeting another back-end solution would necessitate additional manual changes.

This chapter presentes two main contributions:

— A refinement of the standard CoqTL specification, including three major optimizations to increase parallelism opportunities. The refined specification, here named Parallelizable CoqTL, is written in Coq, and is validated by formally proving in Coq the input/output equivalence to the standard CoqTL specification.

— A transformation engine that implements Parallelizable CoqTL on top of Apache Spark, named SparkTE. By evaluating the performance of a simple case study, we assess the speedup that SparkTE can reach.

5.2 Motivation and Background

5.2.1 Running Example

We choose a simple transformation as a running case to illustrate the approach and perform preliminary performance assessments. Listing 5.1 shows an excerpt of the code of a CoqTL transformation named `Relational2Class`, which ideally reverse-engineers class diagrams from given relational schemas, and Figure 5.1 shows its source and the target metamodels. The transformation is a simplified inverse of the well-known Class2Relational transformation [165], often used in the community for exemplifying new contributions. We choose the inverse direction because it is representative of reverse-engineering transformations, where scalability problems frequently arise [15].

The transformation is written in CoqTL, an internal DSL for model transformation within the Coq theorem prover. The transformation primitives are newly-defined keywords (by the notation definition mechanism of Coq), while all expressions are written in Gallina, the func-

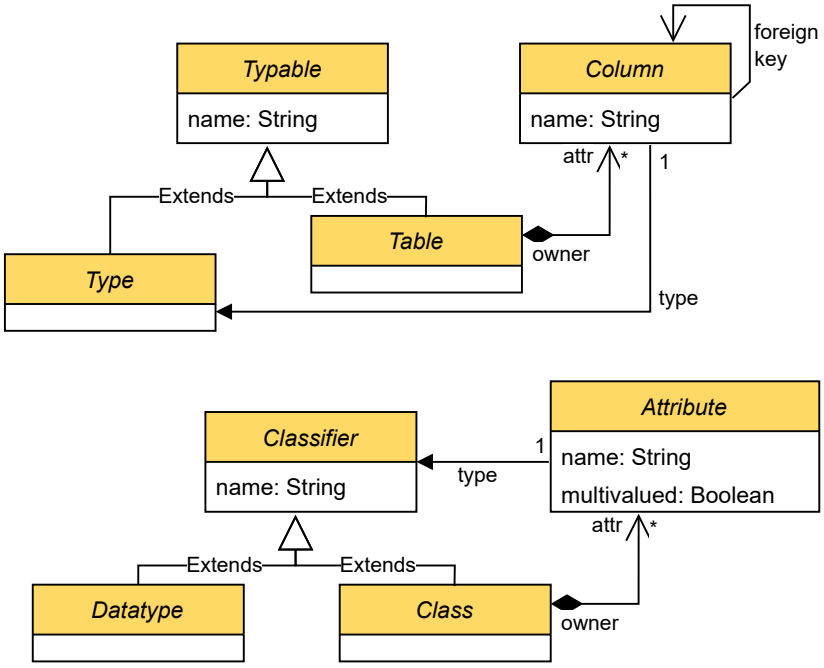


Figure 5.1 – Relational and Class diagram metamodels

tional language used in Coq. The CoqTL semantics is heavily influenced by ATL [99] (notably in the distinction between a match/instantiate and an apply function), and its original design choices focus on simplifying proof development.

In Listing 5.1, the `Relational2Class` transformation is defined via four rules, composed of two parts: (i) a matching section (type and guard condition) and (ii) an output section, which contains a definition for created target elements and optional references. To keep a trace of which expression is used for mapping a source to a target element, each element of the output section of the rules is named.

The first rule (lines 5–11) maps all relational `Types` to class-diagram `Datatypes`. We specify that a datatype is constructed using `BuildDataType` with the same `id` and the same `name` as the matched type (line 10). The `Table2Class` rule (lines 5–11) translates all tables to a corresponding class, with the exception of tables that persist multivalued attributes. To filter these tables, a guard condition, introduced by the **`when`** keyword, calls a user-defined `isClassTable` function (line 14), that we will discuss later. The created target class is constructed from the `id` and the `name` of a matched table (line 18). The third rule (lines 21–40) generates `Attributes` and links them to `Classifiers`. `Columns` are transformed into single-valued `Attributes` by a call to `BuildAttribute` with the last parameter (`multivalued`)

```
Definition Relation2Class :=
transformation from RelationalMetamodel to
  ClassMetamodel with m as RelationalModel :=

  rule Type2Datatype
    from element t class Type
    to [
      output "type"
        element dt class Datatype :=
        BuildDataType (getTypeId t) (getTypeName t)
    ];

  rule Table2Class
    from element t class Table when isClassTable t
    to [
      output "class"
        element c class Class :=
          BuildClass (getTableId t) (getTableName t)
    ];

  rule Column2Attribute
    from element c class Column when
      isClassTable (getOwner c m)
    to [
      output "svattr"
        element a class Attribute := BuildAttribute
            (getColumnId c) (getColumnName c) false ;
        links [
          reference AttributeType :=
            ty <- getColumnType c m;
            dt <- resolve Relational2Class m "type"
                   DataType [[ ty ]];
            return BuildAttributeType a dt;
          reference AttributeClass :=
            ta <- getColumnTable c m;
            cl <- resolve Relational2Class m "class"
                   Class [[ ta ]];
            return BuildAttributeClass a cl
        ]
    ];

  rule MVAttributeTable2Attribute
    from
      element t class Table; element o class Table
      when isMVAttributeTable t o
    to [
      output "mvattr"
        element a class Attribute := ...
    ]
```

Listing 5.1 – Excerpt of the Relational2Class Transformation in CoqTL

```
Lemma tr_r2c_inverse: ∀ (m : ClassModel),
execute Relational2Class (execute Class2Relational m)
  = m.
```

Listing 5.2 – The tr_tracePattern_source lemma

equal to `false` (line 27). Again, columns that are contained by tables that persist multi-valued attributes are not transformed, thanks to a guard `isClassTable` (line 14). Additionally, two links are defined for the generated `Attribute`. First, a link from the target `Attribute` to its `type` (lines 29–33). The corresponding `type` is defined in `BuildAttributeType` by resolving the output of the `Type` of the source `Column` (line 32). The second reference, from the `Attribute` to its `owner` (lines 34–38), is defined in the same manner (i. e., by resolving the output of the owner `Table` of the source `Column`) (line 37). `MVAttributeTable2Attribute` (line 42–49), that is the last rule of our transformation, matches two tables: a table `t` generated from a multi-valued attribute and the table `o` that corresponds to its owner. The two references are built as in the `Column2Attribute` rule: one reference to the type of the attribute and one to its owner.

Once the transformation has been defined in CoqTL, it is possible to prove that it has a given property, e. g. it respects a given contract. For instance, CoqTL users can interactively prove properties of `Relational2Class` and `Class2Relational`, by proving the lemma in Listing 5.2 using Coq tactics. Several strategies are possible to distinguish tables that were generated from classes, from tables that correspond to multivalued attributes. These strategies would correspond to different implementations of the functions `isClassTable` and `isMVAttributeTable`. In our simple example, we assume that *Class2Relational* translates multivalued attributes into tables with a specific name pattern (e.g., name of the owner class followed by an underscore and the name of the multivalued attribute). Under this assumption `isClassTable` and `isMVAttributeTable` just contain a string check, with constant complexity. In the experimentation (Section 5.5) we will introduce more computational time in these functions to simulate increasing complexity, to better estimate the potential parallelization of our solution.

5.2.2 Objective

CoqTL includes an executable semantics for the transformation engine. An implementation of the engine in OCaml or Haskell can be automatically obtained by the standard extraction mechanism of Coq. However, since the executable semantics is designed to be rea-

soned about in proof terms, it is kept very simple (the core semantics is formalized in 196 Coq LOC) and does not include any efficiency optimization. This has a significant impact on the performance of the extracted engine for CoqTL. For instance , the execution of the Relational2Class transformation on a sequential version of the CoqTL engine for a model of a thousand of model elements takes more than 4 hours on a recent laptop (Intel Core i7-8650U CPU @ 1.90 GHz, 32 GB of RAM).

Considering the size of realistic code-bases for reverse-engineering projects, we aim for a solution that allows users to deploy and run the transformation on a state-of-the-art distributed data-analytics framework. Such frameworks are often already deployed in companies that perform large computations, or in their cloud-based services portfolio.

Apache Spark is a data-analytics framework aiming at querying or manipulating large-scale data. In a local mode, the computation is run using a single machine, while in a cluster mode a master dispatches some partitions and tasks to several other additional machines: the workers. Such architecture allows to use more resources (memory, number of processors) which lead to the opportunity of running computations on larger datasets. The data-parallel approach promoted by Spark consists in using a specific data structure called RDDs (Resilient Distributed Dataset) that is partitioned by the master. Partitions are distributed among the workers as well as tasks to execute on RDDs. A task in Spark is a set of operations performed on a RDD. Spark supports several programming paradigms. The use of these paradigm for model-management operations has been investigated in [139]. In this chapter, we use Spark with the MapReduce paradigm [110].

We aim at designing a solution that executes on Spark a CoqTL transformation (like `Relational2Class`), certified against a contract. On the performance side, we aim at improving two kinds of scalability: 1) The capacity of scaling with additional computational resources, referred to as vertical scalability; 2) The capacity of dealing with increasing datasets, referred to as horizontal scalability. On the reliability side, while the whole execution stack can not be certified end-to-end (e. g. the formal semantics of Apache Spark is not available), we aim nonetheless to produce a solution that gives users high confidence that the proved contract will hold on the distributed transformation.

5.3 Approach Overview

Figure 5.2 shows the global workflow of our approach, separating the formal part, written in Coq (left side), from the executable engine in Scala and Spark (right side). Starting from

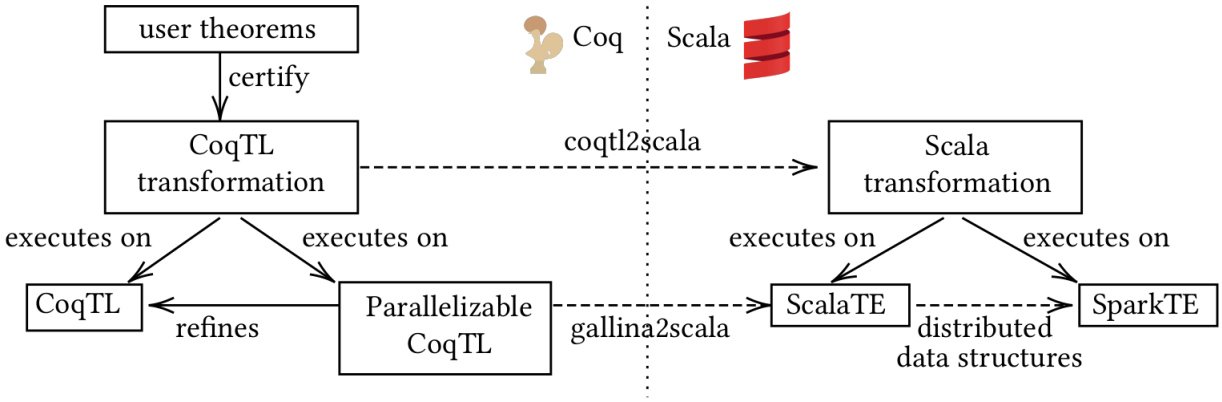


Figure 5.2 – Global overview of our workflow to execute certified model transformations on Apache Spark.

the Coq part, the workflow must be interpreted as follows.

The CoqTL language allows users to define model transformations, theorems on their behavior and machine-checked proofs of these theorems in Coq. These transformations can be executed directly on the executable CoqTL specification. We introduce *Parallelizable CoqTL*, a refinement of the executable CoqTL specification that increases its parallelization opportunities. We use the Coq theorem prover to prove that this new specification is a refinement of the original one (see Section 5.4.2). This entails that, for a given source model, any transformation produces the same output model when it runs on standard CoqTL or Parallelizable CoqTL. The Parallelizable CoqTL specification is written in Gallina, the functional language used in Coq.

Then we provide an implementation of Parallelizable CoqTL in Scala. To obtain the Scala implementation, that we name ScalaTE, we manually extract the Gallina functions into corresponding Scala functions. Section 5.3.1 details this extraction.

ScalaTE uses data structures that are the direct correspondent of Gallina data structures (e. g. Scala `List` for Gallina `list`). In a final step we replace these data structures with distributed data structures (RDDs) from the Spark library, as described in Section 5.3.2. We name the resulting distributed engine SparkTE.

To run a transformation on SparkTE, the CoqTL transformation rules (e. g., Listing 5.1) have to be translated into their corresponding Scala version. We currently perform this step manually, but its automatization is possible and left for future work. Since Spark does not change the functional interface of standard Scala, the obtained Scala transformation can run on both ScalaTE and SparkTE.

```
Fixpoint tuples_of_length_n {A :Type} (s1: list A)
(n : nat): list (list A) :=
match n with
| 0 ⇒ nil::nil
| S n ⇒ prod_cons s1 (tuples_of_length_n s1 n)
end.
```

Listing 5.3 – CoqTL definition for generating all combination of elements

Listing 5.4 – Extracted Scala code from the Coq function `tuples_of_length_n`

```
1 def tuples_of_length_n[A] (s1: List[A], n: Int)
2 : List[List[A]] =
3 n match {
4     case 0 =>
5         List(List())
6     case n1 =>
7         prod_cons(s1, tuples_of_length_n(s1, n1-1))
8 }
```

5.3.1 Coq to Scala

The Coq environment includes an extraction mechanism targeting ML languages: OCaml, Haskell, or Scheme. Although an automatic extractor to Scala is available [67], it supports only a subset of Gallina on an outdated version of Coq. Hence, we opted to perform the extraction manually. We perform manual extraction at two levels: first to create the core engine (*gallina2scala* in Fig. 5.2), then to obtain Scala rules representing a CoqTL transformation (*coqtl2scala*).

Gallina to Scala The executable CoqTL specification can be seen as a functional program that interprets the transformation code. We produce a literal translation of this interpreter in Scala.

For extracting Scala code from Parallelizable CoqTL we translate Gallina types and (pure) functions into their correspondent types and pure functions in Scala. Listings 5.3 and 5.4 show an example of CoqTL executable specification (in Gallina) and its implementation in SparkTE (in Scala). The example well illustrates the one-to-one translation among types and functions that we adopt for all the extraction.

We implicitly assume that Scala functions (e. g., `flatMap` for Scala `List`) implement the same semantics than their Gallina correspondent (e. g., `flat_map` for Gallina `list`).

Listing 5.5 – Scala implementation for the Table2Class rule

```
new RuleImpl (
  name = "Table2Class",
  types = List(RelationalMetamodel.TABLE),
  from = (m, l) => {
    val t = l.head
    Some(t.isClassTable)
  },
  to = List (
    new OutputPatternElementImpl(
      name = "class",
      elementExpr = (i, m, l) =>
        if (l.isEmpty) None else {
         val t = l.head.asInstanceOf[RelationalTable]
         Some(new ClassClass(t.getId, t.getName,
           multivalued=false))}
  )))
```

CoqTL to Scala The CoqTL parser translates the concrete syntax of the transformation (e. g., from Listing 5.1) into Coq code to construct an abstract syntax tree. Obtaining the same transformation in Scala requires constructing the same abstract syntax tree as Scala objects. Note that Scala constructors for the abstract syntax are the literal translation of the corresponding Gallina constructors in CoqTL.

Listing 5.5 shows the translation of the *Table2Class* rule from Listing 5.1. The rule constructor (`RuleImpl`) requires a name, a list of types for an input pattern (the `types` argument), a guard condition (the `from` function), a list of output pattern elements (the `to` argument). Each output pattern element has a name, and a function for creating output elements from input pattern elements. While not shown in this example, it can be also accompanied by a list of functions for creating links in the output model.

As illustrated by the example, the only non-trivial part of this extraction is the translation of the body of expressions for guards and output element creation (e. g. the body of the anonymous functions in Listing 5.5). An automatic compiler from CoqTL rules to Scala is a part of future work (Section 5.6).

5.3.2 Distributed Data Structures

Spark RDDs are data structures that are automatically partitioned and resulting in the distribution of the computation operations on a Scala sequence (e. g., `List`) of serializable

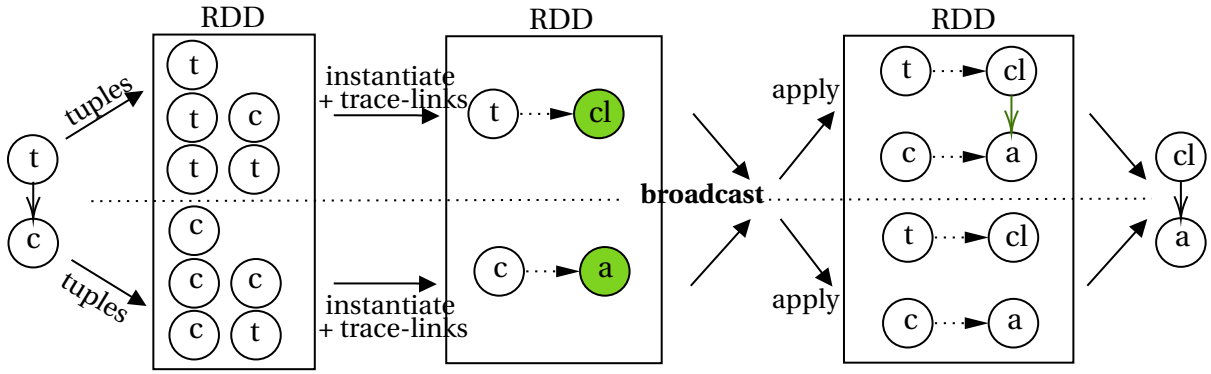


Figure 5.3 – Distributed computation of a model transformation on SparkTL. *c* denotes a column, *t* a table, *a* an attribute, and *cl* a class. Green elements and links illustrate the entities created by the transformation in the output.

elements. From a user point of view, RDDs can be manipulated as lists, using the same primitive functions, and parallelism is implicit. The advantage of using such abstraction for parallelism is the semantics preservation of the operations on the distributed structures. Because of the popularity of Spark and its support, we assume the correctness of parallel operations on the data.

An efficient use of RDDs requires an effective partitioning of data. For instance, to take advantage of the internal multi-threading mechanism, it is typically recommended in Spark to assign four data partitions to each core. Each independent computation of a partition is referred as a task. The Spark task scheduler makes the distribution following a round-robin approach, optimizing load-balancing: once a task is ended on a core, a new one can be assigned from the waiting list. Section 5.4 gives more detail about how we use RDDs in SparkTE.

5.4 Parallelizable Semantics for CoqTL

The `execute` function shown in Listing 5.6 is the entry point of the transformation execution in the standard CoqTL semantics.

First, the `allTuples` function (line 3) produces all the tuples of elements that can be possibly matched by the rules. `allTuples` computes a list of $\sum_{a=0}^{A} n^a$ tuples, with n the number of elements in the input model, and A the maximum arity of the transformation rules.

Then, the `instantiatePattern` function (line 5) tests each tuple to find if it matches with any rule and for each match it constructs the corresponding output pattern elements.

```
Definition execute (tr: Transformation) (sm: SourceModel)
    : TargetModel :=
    let tuples := allTuples tr sm in
    let elements :=
      flat_map (instantiatePattern tr sm) tuples in
    let links := flat_map (applyPattern tr sm) tuples in
    Build_Model elements links.
```

Listing 5.6 – `execute` function in the base CoqTL specification

```
Definition execute (tr: Transformation) (sm : SourceModel)
    (sm : SourceMetamodel) : TargetModel :=
    let tuples := allTuplesByRule tr sm mm in
    let (elements, tls) :=
      flat_map (tracePattern tr sm mm) tuples in
    let links :=
      flat_map (fun sp ⇒ applyPatternTraces tr sm sp tls)
        (allSourcesPattern tls) in
    Build_Model elements links.
```

Listing 5.7 – `execute` function in the Parallelizable CoqTL specification

Internally it iterates on each rule, executes their guard function and if the result is positive, executes the element creation function for each output pattern element of that rule. The resulting elements are gathered by the `flat_map` in a single output list.

Finally, the `applyPattern` (line 6) function is executed on each tuple to create target links. Similarly to the function `instantiatePattern`, the function internally iterates on all rules and checks if the rule matches the given pattern. In a positive case, the element creation functions for that rule are executed and then the link creation functions. The resulting links are gathered by the `flat_map` in a single output list.

5.4.1 Parallelizable CoqTL

Parallelizable CoqTL contains three optimizations to the base CoqTL specification:

— To increase parallelization, the algorithm is split into two consecutive phases, *instantiate* and *apply*, that are built on parallelizable functional patterns (`flat_map`);

— To improve load balancing of the instantiate phase, only possibly useful tuples are generated and then distributed;

— To improve load balancing of the apply phase, a set of trace links is produced by the instantiate phase and the apply phase iterates only on those trace links.

Note that similar optimizations (among others) are already implemented in well-known transformation engines, like ATL [99] or ETL [106]. Differently from previous work, in this chapter we formalize the optimizations, interactively prove that they do not affect the transformation output and assess their impact on distributed execution.

The entry point of Parallelizable CoqTL, is presented in Listing 5.7. Figure 5.3 illustrates the global behavior on a minimal example with one table and one column.

Optimization 1: Two phases In standard CoqTL the `applyPattern` function performs all the computation of the links generated by a matched input pattern, by the rule that matches it. However the computation of the links of a rule is not independent from the computation of other rules. This dependency is caused by the `resolve` function (e. g., lines 32 and 37 in Listing 5.1) that searches for the output of another rule in order to set the target of the created link. In general, because of this dependency, two executions of the apply function can not be run in parallel, without replicating some matching and instantiation within each call to `resolve`. [1]

We refactor the computation to split it in two phases, similarly to ATL [99]. This is visible in Listing 5.7. In the first phase (lines 5) we compute the tuples and we run the matching and instantiation by a new function named `tracePattern`. The first phase produces the list of generated elements, and trace links connecting them to their corresponding source patterns. Differently from Listing 5.6, here the second phase (line 8)) can only start computing output links after the full first phase has finished computing the trace links, since the `flat_map` expects the `tls` structure as parameter.

In Listing 5.7 every execution of `tracePattern` can be run in parallel. When the first phase is over, every execution of `applyPatternTraces` can be run in parallel too, since the calls to resolve can be computed immediately on the trace-link structure. This greatly improves the parallelization of the algorithm.

Optimization 2: Tuple generation by rule Matching a pattern to a rule happens in two consecutive steps. First, the types of the pattern are checked against the types expected by the rule. Then, if the types are correct, a guard condition is evaluated. The type checking is very fast, hence it acts as a first filtering. Instead the evaluation of the guard condition can

1. This is exactly how the standard CoqTL specification computes resolve. This simple solution keeps the specification compact and has no negative impact on proofs. However it has a big impact on performance and parallelization.

Table 5.1 – Size (in LOC) of new specification and certification proofs added for each optimization, with proof effort (in man-days).

Optimization	Spec. size	Cert. size	Proof effort
twoPhases	69	484	10
byRule	42	487	7
iterateTraces	69	520	4

potentially be very long, or navigate large parts of the model. So it is executed only for the few tuples that pass the type check.

Since the tuples that require an evaluation of the guard condition are a small subset of all the possible tuples, arbitrarily distributing all tuples among the cores can potentially lead to imbalanced partitions. In particular it would not be uncommon to have partitions that do not require any guard evaluation, opposed to partitions that need to evaluate several expensive guards. In such cases, idle workers would wait for the synchronization barrier of Spark to start new computations. The imbalance impacts the scalability of the program.

To limit imbalance, in the initial sequential tuple generation phase, we generate only tuples whose type matches with at least one rule of the transformation. This is shown in Listing 5.7 by the use of the `allTuplesByRule` for tuple generation (line 2). `allTuplesByRule` iterates on rules and produces only combinations of elements of the types listed in the rule input pattern. This improves load balancing of the first phase since all the produced tuples require a guard evaluation.

Optimization 3: Apply iterates on traces Executing the apply phase on the tuples generated by `allTuplesByRule` would cause an imbalance among partitions, similar to the one discussed in Optimization 2. Indeed, among these tuples only very few have passed the guard condition in the first step. A partitioning of `allTuplesByRule` would produce partitions that do not require any computation, together with partitions that need to evaluate several expensive link creation functions. In this optimization we make the apply phase iterate only over the source patterns that passed the guard evaluation in the instantiate phase. We retrieve these patterns by collecting them from the list of trace links. This is performed by the function `allSourcePatterns` at line 8 of Listing 5.7.

5.4.2 Refinement Proof

```
Lemma exe_preserv :
∀ (tr: Transformation) (sm : SourceModel),
  twophases.execute tr sm = execute tr sm.

Lemma In_by_rule :
∀ (sp: list SourceModelElement) (tr: Transformation)
  (sm: SourceModel),
  In sp (allTuplesByRule tr sm)
    → In sp (allTuples tr sm).

Lemma In_by_rule_instantiate :
∀ (sp: list SourceModelElement) (tr: Transformation)
  (sm: SourceModel),
    In sp (allTuples tr sm)
      ∧ instantiatePattern tr sm sp <> nil
        → In sp (allTuplesByRule tr sm).
Lemma In_by_rule_apply : ...

Lemma tr_tracePattern_source:
∀ (tr: Transformation) (sm : SourceModel)
  (tl : TraceLink) (sp: list SourceModelElement),
  In tl (tracePattern tr sm sp)
    → sp = TraceLink_getSourcePattern tl.
```

Listing 5.8 – Certifying optimizations lemmas

Besides the executable functional specification, CoqTL is also described by an axiomatic specification. Certifying against the axiomatic specification involves providing 10 types, 27 semantics functions and proving 15 theorems. The specification is fully illustrated in [49], together with a proof that engines implementing the executable specification certify against the axiomatic one.

We prove that engines implementing Parallelizable CoqTL certify the axiomatic specification, too. For this step, we naturally reuse the types, functions and certification proofs of the base executable specification that are not impacted by the optimization. Each optimization is proved independently. Table 5.1 shows the size (in lines of code) of new specifications and proofs required for describing and certifying each optimization, plus the human effort (in man-days) to complete the proofs. The refined specifications and their proofs are available online [2]. All the discussed lemmas below can be found in Listing 5.8.

The key step for certifying the *twoPhases* optimization, is proving that it does not change the output of the full transformation (`exe_preserv`). This proof has two parts: 1) for the instantiation phase, we prove that the additional computation of the trace, does not have any effect on the computation of the instantiated elements; 2) for the apply phase we need to prove that the new apply function is equal to the old one. Coq is able to prove automatically the second step, by full unfolding and simplification of the old and new apply functions. Note that in this proof we use the axiom of functional extensionality (two functions are equal if their values are equal at every argument), notably to compare the body of inner anonymous functions.

The *byRule* optimization changes the order of the element and link creation in the transformation output, hence a lemma similar to `exe_preserv` would not hold. We prove its equivalence in two steps: 1) we prove that tuples computed by rule are a subset of the all tuples (`In_by_rule`), 2) we prove that tuples computed by rule include all tuples that produce elements or links (`In_by_rule_instantiate`). The second step is the most challenging and is performed by case analysis on matchPattern: given any pattern, if the pattern does not match then it cannot produce anything, if it matches then it is one of the patterns generated by the rule `allTuplesByRule`.

The *iterateTraces* optimization does not change any function in the instantiate phase, hence we can easily prove a similar theorem to (`exe_preserv`), but limited to the result of instantiate. The apply phase does not change the computation of a single link, but skips some source patterns, and produces links in a different order w.r.t. the base version. To prove

2. `https://github.com/atlanmod/SparkTE_public/`

that the same links are generated, we proceed by case analysis on the trace generated for a given source pattern: 1) if the instantiate phase did not generate any trace for that pattern, then the standard apply phase would not have produced any corresponding link; 2) if there is a trace, then the apply phase applies the same function to the same source pattern, hence producing the same output (`tr_tracePattern_source`).

5.4.3 Implementation on Spark

Distributed computation of our Spark implementation revolves around the use of RDDs. First, the input patterns, represented as a list of tuples, are distributed with an RDD among cores. Each core independently applies the instantiate function to every tuple of its partition. Implicit communications are operated by Spark to scatter the tuples from the master process to the workers. After the computation, the resulting elements and their trace-links are all gathered to the master process.

For the second phase, the trace-links are distributed with an RDD. Each core is in charge of generating the links for a partition of output elements and their associated applied rule. Since this second phase needs a global knowledge of the trace-links to resolve output elements, we used a broadcast communication to share the whole set of trace-links to all cores.

A global view of where RDDs are used and what communications are operated is illustrated in Figure 5.3.

5.4.4 Limitations

SparkTE implements the complete CoqTL specification but some limitations remain. First, all the translations from Coq to Scala are manual. In particular, the transformation rules need to be manually translated by the user. However the correspondence between the abstract syntax in CoqTL and ScalaTE/SparkTE makes the translation of the rule structure trivial. Hence, our solution reduces the complex problem of translating to Scala a CoqTL transformation, into a simpler problem: translating to Scala separately each side-effect-free expression for guards, output pattern elements creation, and output pattern links creation. While simpler, this task is still tedious and error prone, hence we aim at making the translation fully automatic in future work.

Table 5.2 – Hardware setup of the two clusters used during experiments. These clusters are part of the Grid'5000 experimental platform for distributed computing.

Cluster	CPU	RAM	Netw.
Rennes *paravance*	2× 8 cores/CPU *Intel Xeon E5-2620*	128GB	2× 10Gbps
Nancy *gros*	2× 18 cores/CPU *Intel Xeon Gold 5220*	96 GB	2× 25Gbps

Table 5.3 – Description of the three datasets used in the experiments with the number of elements and links.

Dataset	D1	D2	D3	D4	D5
model type	Relational			IMDb	DBLP
elements	150	300	600	440	700
links	290	580	1060	1968	1886

5.5 Experiments

In the previous sections, we have presented the three optimizations adopted for Parallelizable CoqTL to increase the efficiency and propensity to parallelization of the extracted engine. In this section, we evaluate the performance of SparkTE and the quality of its speedup compared to the ideal speedup that divides the execution time by the number of cores (relative to a reference version). As none of the contributions of the related work have studied the use of Spark for parallel transformations, we restrict our comparisons to this theoretical ideal speedup. All jobs are run with Spark in a standalone mode.

The first subsection presents the results we get from the use-case previously described in Section 5.2.1, i. e. the *Relational2Class* transformation, and similar experiments on two additional cases.

We show that the three optimizations introduced in Section 5.4 improve significantly the performance of the transformation in Spark. The results also show a relatively poor speedup compared to the ideal one, but that can be improved when having more complex operations within the transformation. To investigate this result, the second subsection illustrates that a speedup close to the ideal can be observed with a high number of cores if the computation time and the size of the dataset is big enough to offset the overheads of Spark. Finally, a performance analysis by phase is presented to show the very good speedup obtained by the

parallelized steps of SparkTE.

All results presented in this section have been executed ten times on the same hardware configuration, and an average is presented. The standard deviation of our measurements are never higher than 10 % of the total execution time, thus guaranteeing our results to be stable, and the average to be meaningful. Furthermore, all our experiments have been conducted on the French experimental platform for distributed computing Grid'5000. Grid'5000 is made of geographically distributed clusters of machines in France, each one with its specific hardware setup. This platform also facilitates the reproducibility of the experiments by offering a way to build the same environment for any researcher. However, Grid'5000 makes us dependent on available machines (i. e. nodes) for provisioning which is why we use two different clusters of Grid'5000 in our experiments. For each experiment the number of machines and the number of cores are specified. One can note that the number of machines that is specified correspond to the number of workers instantiated in Spark and that one additional dedicated machine is provisioned to host the master of Spark. All our codes, raw results, and analysis scripts are publicly available online. [3]

5.5.1 Evaluation of SparkTE on Use-cases

In this section we apply our running example to a first model, denoted D1 in Table 5.3. We also consider two additional transformations: the IMDb case [92], aiming at finding couples of actors who recurrently played together, and queries on a DBLP model to find active authors who published in specific journals [10]. We run the two additional transformations on model instances from the IMDb and DBLP metamodel, respectively denoted D4 and D5 in Table 5.3.

The transformations specified in CoqTL, subsequently translated to Scala and Spark, are always correctly computed by SparkTE, i. e. with the expected output of the transformation, for all our experiments.

We have used from 1 to 2 machines of the cluster *paravance* detailed in Table 5.2. Table 5.4 presents the execution times and speedups of the three transformations from 1 to 8 cores with 1 and 2 machines. Notice that proofs of lemmas are Coq programs executed on the specification only and are not a part of the target computation, and are not considered in the evaluation performance. The results show a relatively poor speedup. It is caused by the lack of complexity in the operations used in the guard condition, the instantiate function

3. `https://github.com/atlanmod/SparkTE_public`

and the apply function.

This phenomenon will be confirmed in the performance analysis of the next subsection.

Spark's overheads are more particularly observable with the poor speedup measured with 2 cores. Indeed, in this case the gain of parallelizing with Spark is completely swallowed by the overheads, but then an improvement is obtained with 4 cores. Overall the obtained speedup is not satisfying as the speedup for 8 cores is almost the same as with 4 cores, and far from the expected ideal speedup of 8. We will show in the next subsection that Spark's overheads can be almost fully compensated by increasing the computation time of the transformation and/or the size of the dataset. The additional cases show that the performance of the approach depends on the computation time of the rules. In the IMDb case, the better growing speedup is caused both by the rules themselves that have a high-level of complexity, and the high connectivity of the input model. On the contrary, in the case of the DBLP transformation, the rules with a low computational cost lead to a poor speed-up. We will show in the next subsection that Spark's overhead is fully compensated by high computation time of the transformation and/or large size of the dataset.

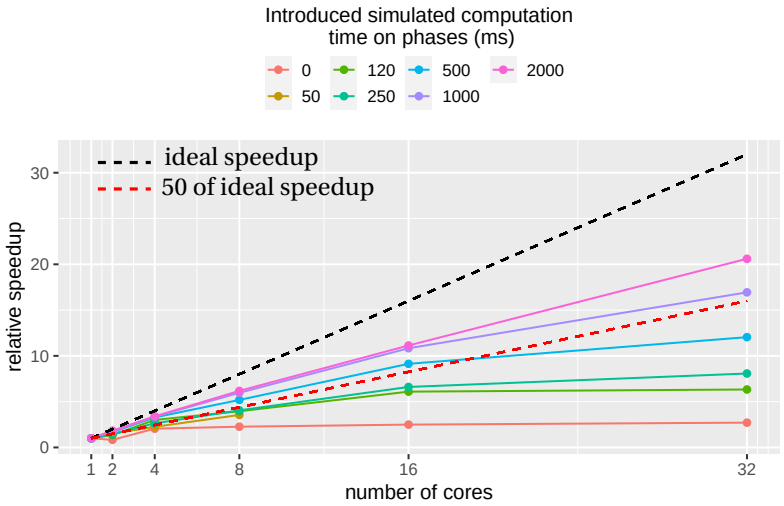
One problem with computation on large models is the lack of memory. Distributed solutions, such as the one we propose here, allow users to increase the resources allocated to a transformation. For instance, a fixed allocated memory for one core is not enough for processing the DBLP example, causing disk operations which drastically slow the computation. This phenomenon disappears with 2 cores, and we observe a hyper speedup.

Finally, we want to show the performance gain obtained thanks to our three optimizations. To this purpose, we execute the same *naive* transformation on the direct implementation of the CoqTL specification (without optimizations). On 1 core *naive* is computed in 27 s by SparkTE and in 52 s by the CoqTL implementation.

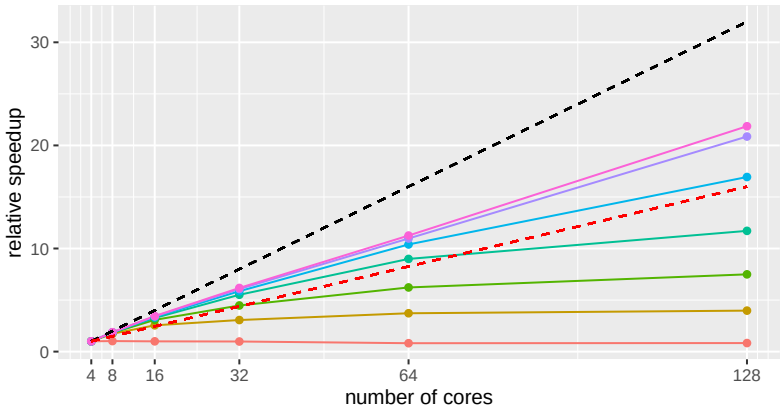
5.5.2 Performance Analysis by Complexity and Datasets

In the previous subsection, we have shown that the performance of SparkTE is enhanced by our three optimizations in terms of execution time and speedup. However, compared to the ideal speedup our use-cases are disappointing. In this subsection, we demonstrate that these results are due to the small computation time in the transformation and the small size of the dataset.

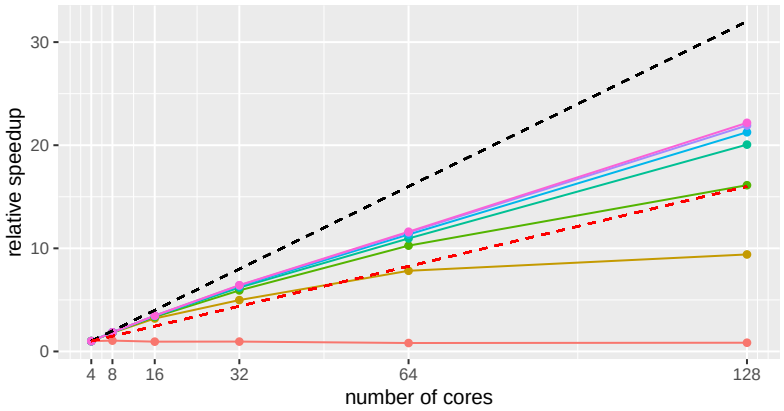
Experimental setup. To this purpose, we have built a version of the transformation with additional fictitious processing time in the different phases and steps. More precisely, we incorporate sleeping functions to different parts of the rule evaluation: the guard condition to



(a) Relative speedups of SparkTE for D1 up to 32 cores on 4 machines



(b) Relative speedups of SparkTE for D2 up to 128 cores on 8 machines



(c) Relative speedups of SparkTE for D3 up to 128 cores on 8 machines

Figure 5.4 – Relative speedups of SparkTE with sleeping times

Table 5.4 – Execution times and speedups (relative to 1 core) for the executions with SparkTE of *Relational2Class* (R2C), the IMDb findcouple transformation, and the DBLP case. Experiments respectively conducted with the dataset D1, D4, and D5, on the cluster *paravance*.

	♯ cores (♯ machines)	1 (1)	2 (1)	4 (2)	8 (2)
R2C	time (s)	27.02	32.50	13.17	11.91
	speedup	1.00	0.84	2.13	2.31
IMDb	time (s)	38.35	22.01	18.33	11.61
	speedup	1.00	1.74	2.09	3.30
DBLP	time (s)	0.83	0.35	0.56	0.84
	speedup	1.00	2.37	1.49	0.99

Table 5.5 – The set of experiments described in Section 5.5.2 with different fictitious processing time in the transformation, and different sizes of datasets. The number of cores used by each benchmark is also indicated, as well as the number of cores per node (i. e. worker). Finally, the Grid'5000 cluster used for the benchmark is given in the last column.

Benchmark	**B1**	**B2**	**B3**
dataset	D1	D2	D3
sleeping	$\{0, 50, 120, 250, 500, 1000, 2000\}$	same as B1	same as B1
♯ cores	$\{1, 2, 4, 8, 16, 32\}$	$\{4, 8, 16, 32, 64, 128\}$	same as B2
♯ machines	$\{1, 2, 4\}$	$\{1, 2, 4, 8\}$	same as B2
♯ cores per node	4 to 8	4 to 16	same as B2
cluster	*paravance*	*gros*	*gros*

simulate complex checking functions; the instantiate part to simulate complex instantiation of element; and the apply function to simulate a long resolution for links. As we discussed in Section 5.2, user-defined functions can have several implementations with different complexities. Proposing an evaluation based on the computation time of these functions offers an infinite set of benchmarks to accurately estimate the speedup of SparkTE. Furthermore, we additionally vary the size of the dataset in the obtained benchmarks.

Table 5.5 summarizes the set of benchmarks explained hereafter. The first benchmark (B1) uses the same data introduced in the previous subsection on (D1) with 1, 2, and 4 machines, and up to 8 cores per machine on the *paravance* cluster of Grid'5000. We compare the computation time of the transformation on 2, 4, 8, 16, and 32 cores relative to the execution time on 1 core. The second and third benchmarks, (B2) and (B3), illustrate both the horizontal scalability and the vertical scalability of SparkTE by increasing the size of the dataset

with (D2) and (D3) detailed in Table 5.3, and the number of machines (i. e. nodes, workers) and cores detailed in Table 5.5. The two benchmarks are evaluated on 1 to 4 machines, up to 16 cores per machine. As increasing the size of the dataset and the fictitious sleeping time also increases the total execution time of experiments, we compute our speedup relative to 4 cores instead of 1. As a result, the ideal speedup for 16 cores, for instance, is 4. Experiments on (B2) and (B3) are conducted on the *gros* cluster of Grid'5000. Furthermore, as indicated in Table 5.5, the three benchmarks are evaluated on the following sleeping times: 0, 50, 100, 250, 500, 1000, and 2000 ms. Finally, for the three benchmarks the overall execution time is measured, thus including some small sequential parts of the code, and the broadcast phase.

Results. Figures 5.4a, 5.4b, and 5.4c show the speedups observed for each benchmark according to the sleeping time and compared to the theoretical ideal speedup. When increasing the sleeping time, i. e. the execution time of the transformation, the speedup is enhanced and is closer to the ideal one. By increasing the size of the dataset, one can note a slight increase of the speedup by comparing B2 and B3 results respectively in Figure 5.4b and Figure 5.4c. Indeed, in Figure 5.4b, at 128 cores, more than half of the points are below the 50 % *optimal* value, while in Figure 5.4c only two points are below this theoretical value. In other words, at 128 cores, (D2) needs a sleeping time of 500 ms to reach 50 % from optimal speedup while (D3) only needs a sleeping time of 120 ms.

5.5.3 Performance Analysis by Phase

This third subsection aims at analyzing the impact of the two phases on the global speedup of the program. To do so, we processed the same benchmark as before (i. e., (B1), (B2) and (B3)), but with additional counters within the program to record the computation time of each distinct part: (1) the tuples generation; (2) the instantiate phase; (3) the broadcast intermediate step; and (4) the apply phase.

Results Our results on the three benchmarks show that the total computation time is mainly driven by the parallel parts. Indeed, in the case of the biggest dataset, with a sleeping time of 0 ms, the sequential part represents only 3.5 % of the total computation time which is the maximum percentage of the sequential parts. For instance, at the opposite extreme, i. e. in the case of the smallest dataset, with a sleeping time of 2000 ms, the sequential part represents less than 1 % of the total execution time. Hence, in the following, we restrict our analysis to the speedup of the parallel parts.

Table 5.6 illustrates the impact of each phase, by comparing their relative speedup to

Table 5.6 – Relative speedups of SparkTE parallel phases on B1, B2 and B3 for sleeping times equals to 50 ms and 2000 ms. The percentage of the observed speedup compared to the theoretical ideal speedup is indicated for each result (higher the better).

	Cores	**Instantiate phase (50ms)**	**Apply phase (50ms)**	**Instantiate phase (2000ms)**	**Apply phase (2000ms)**
B1	1	1 (100%)	1 (100%)	1 (100%)	1 (100%)
	2	1.48 (74%)	1.62 (81%)	1.81 (90.5%)	1.87 (93.5%)
	4	2.40 (60%)	1.83 (45.75%)	3.21 (80.25%)	3.85 (96.25%)
	8	3.40 (42.5%)	4.50 (56.25%)	5.78 (72.25%)	7.15 (89.375%)
B2	4	1 (100%)	1 (100%)	1 (100%)	1 (100%)
	8	1.68 (84%)	1.90 (95%)	1.83 (91.5%)	1.99 (99.5%)
	16	2.39 (59.75%)	3.18 (79.5%)	3.30 (82.5%)	3.84 (96%)
	32	2.74 (34.25%)	4.66 (59.25%)	5.86 (73.25%)	7.07 (88.375%)
	64	3.17 (19.813%)	7.34 (45.875%)	10.87 (67.938%)	12.34 (77.125%)
	128	3.33 (10.406%)	8.87 (27.719%)	20.92 (65.375%)	24.70 (77.188%)
B3	4	1 (100%)	1 (100%)	1 (100%)	1 (100%)
	8	1.75 (87.5%)	1.97 (98.5%)	1.80 (90%)	1.99 (99.5%)
	16	3.04 (76%)	3.68 (92%)	3.31 (82.75%)	3.95 (98.75%)
	32	4.58 (57.25%)	6.47 (80.875%)	5.97 (74.625%)	7.87 (98.375%)
	64	7.00 (43.75%)	11.47 (71.688%)	10.59 (66.188%)	15.04 (94%)
	128	8.17 (25.531%)	15.86 (49.563%)	19.94 (62.312%)	30.14 (94.188%)

the optimal one. For reading convenience, we only show the results for sleeping times equal to 50 ms and 2000 ms. Table 5.6 confirms our previous assumption about the overhead of Spark that can be offset by increasing the computation time (i. e. sleeping duration) or the size of the dataset. Also, one can note that the *apply* phase offers better scalability than the *instantiate* phase. This result can be explained by the remaining imbalance on the tuples distribution. Let us remind first that the first *instantiate* phase is composed of two steps: (1) a guard condition; (2) the instantiation of the tuples that have satisfied this condition. Hence, even if the partitions are better balanced by our second optimization (see Section 5.4.1), not all the tuples are necessarily computed. As a matter of fact, the guard condition is always evaluated, but not the instantiation that depends on the result of the guard condition. On the contrary, in the *apply* phase apply patterns to all entries which are nearly perfectly balanced.

5.6 Conclusion

In this chapter, we presented a refinement of the CoqTL specification, designed for optimizing the parallel execution of model transformations on Spark. We have illustrated the benefits and the scalability of our proposed optimizations through the Relational2Class example.

In future work, we plan to continue experiments with other use cases (e. g., the IMDB case study). We plan to write a compiler from CoqTL to Scala, to automatically obtain an executable transformation from a Coq specification. Finally, we will study other optimizations, leveraging the vertex-centric paradigm supported by Spark (i. e., GraphX), and the integration with persistence solution (e. g., HDFS).

CHAPTER 6

CONFIGURABLE TRANSFORMATION ENGINE

Contents

In the previous chapters we have covered two cases where designing choices had a deep impact on performances on model management operations. In the chapter, we enrich this concept by proposing multiple configurations in SparkTE, based on different strategies for the computational steps of the model transformation operation. Contrary to previous work on SparkTE, to deeply explore performance impacts, we do not deal on correctness anymore.

6.1 Introduction

Multi-objective parameter optimization is a transversal research question across software engineering disciplines. It has been a concern in the compilation community, with, for

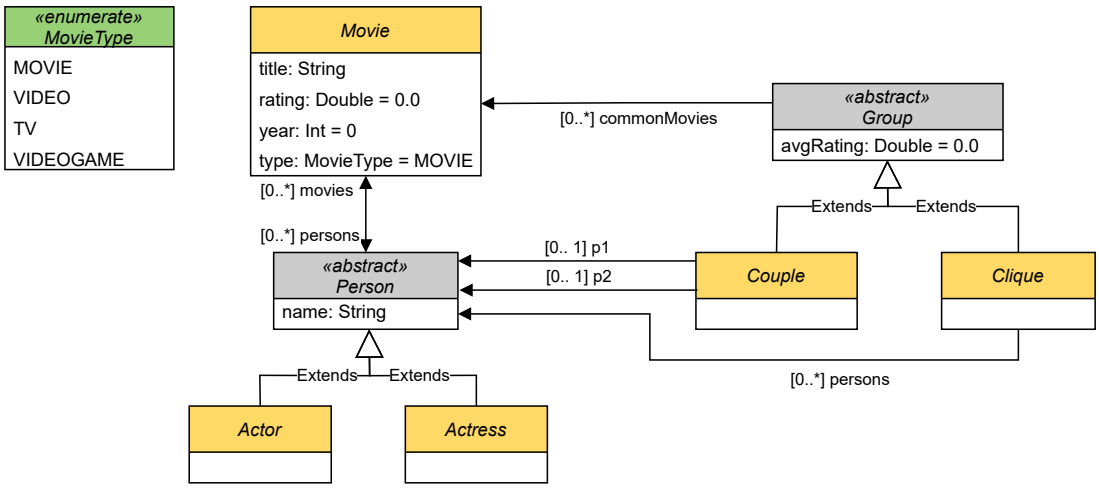


Figure 6.1 – IMDb Metamodel from [92]

example, the automatic configuration of GCC [135]. The operations research field has proposed multiple approaches over the last decades to solve such optimization problems [51], and state of the art tools such as OpenTuner [6] have been developed to practically address them. These optimization problems are also a key concern in the Software Product Lines community [147], where feature models represent multiple variations or configurations of products. Choosing the optimal product satisfying both the feature model's constraints and user-specific requirements is an active topic of research, and has been recently tackled with evolutionary many-objective optimization algorithms [90], sometimes combined with SAT solvers [87].

In the MDE area, efforts have been dedicated to classify transformation engine features [5, 143, 89]. On one side, there are languages with multiple engines (e.g., ATL [112, 166, 15, 165, 113], Epsilon [77, 117, 119]). On the other side, a user could expect a single engine, with several strategies according to different purposes. Only few attempts target such approaches for a single feature (e.g., pattern matching in [19, 93]).

In this chapter, we propose `Configurable SparkTE`, a configurable transformation engine, as an extension of SparkTE, based on multiple implementations for the engine features. The proposed configurations target model exploration strategies, execution strategies, and Spark related features.

6.2 Motivating example

The example that we have selected to illustrate the approach and perform preliminary performance assessments uses the "Movie Database" (IMDb), proposed in the Transformation Tool Contest (TTC) 2014 [92]. Listing 6.1 shows an excerpt of the code of a CoqTL transformation named *FindCouples*, and Figure 6.1 shows its related metamodel. For keeping clarity, the transformation is written with the transformation language proposed in CoqTL.

In Listing 6.1, the `FindCouples` transformation on the IMDb model is defined via four rules, composed of two parts: (i) a matching section (type and guard condition) and (ii) an output section, which contains a definition for created target elements and optional references. To keep a trace of which expression is used for mapping a source to a target element, each element of the output section of the rules is named

The first rule, `Movie2Movie` (lines 5–18) simply re-instantiate movies within the output model. Output `Actors` and `Actresses` of the constructed movie are resolved to create a `MoviePersons` reference.

The second and third rules aim at copying `Person` instances from the input model. The `Actor2Actor` rule (lines 20–32) constructs a new `Actor` with `BuildActor` using the name of the input actor. One link to the list of movies the actor played in is created in reference section. The `Actress2Actress` (line 34) rule is defined in the same manner, replacing the matched `Actor` by an instance of type `Actress`.

The last rule (lines 37–51) matches `Person` and iterate on all potential candidate it can form a `Couple` with. For each candidate, if the guard condition `areCouple` is evaluated to true for two given persons, a new instance of `Couple` is constructed, with a computed average rating from all common movies.

Additionally, two links are defined for the generated `Couple`: one for each member of the group.

Found couples are created thanks to the `Persons2Couple` rule, which matches two persons who are potentially forming a couple. If the guard condition `areCouple` is evaluated to true for two given persons, a new instance of `Couple` is created. Two links are created, respectively for the persons forming the couple. In this use case, a couple is defined as a tuple of two persons who played in at least three common movies. Coq and Scala examples of how this condition is checked are presented in Listing 6.2 and Listing 6.3.

```
1   Definition FindCouples :=
2   transformation from MovieMetamodel to MovieMetamodel
3   with m as MovieModel :=
4
5     rule Movie2Movie
6       from element mv class Movie
7       to [
8         output "movie"
9           element mv1 class Movie :=
10          BuildMovie (getTitle mv) (getRating mv) (getYear mv) (getMovieType mv);
11          links [
12            reference MoviePersons :=
13              p <- getMoviePersons mv;
14              actors <- resolve FindCouples m "person" Actor [p];
15              actress <- resolve FindCouples m "person" Actress [p];
16              return BuildMoviePersons mv1 (actors ++ actress)
17          ]
18      ]
19
20    rule Actor2Actor
21      from element a0 class Actor :=
22      to [
23        output "person"
24          element a1 class Actor :=
25            BuildActor (getName a1);
26          links [
27            reference PersonMovies :=
28              mv0 <- getPersonMovie p;
29              mv1 <- resolve FindCouples m "movie" Movie [mv0];
30              return BuildPersonMovie p mv1
31          ]
32      ]
33
34    rule Actress2Actress
35      from ... (* same than Actor2Actor *)
36
37    rule Persons2Couple
38      from element p1 class Actor;
39      for p2 in candidate m p1 when areCouple (getPerson p1) (getPerson p2)
40      to [
41          element c class Couple :=
42            mvs <- commonMovies (getPerson p1) (getPerson p2);
43            rate <- (fold + (map (mv => getRating mv) mvs)) / (length mvs);
44            BuildCouple rate;
45          links [
46            reference CouplePersonP1 :=
47            BuildCouplePersonP1 c (resolve FindCouples m "person" Person [[p1]]);
48            reference CouplePersonP2 :=
49            BuildCouplePersonP2 c (resolve FindCouples m "person" Person [[p2]])
50          ]
51      ]
```

Listing 6.1 – Excerpt of the FindCouples Transformation in CoqTL

```
Definition areCouple (m: moviesModel) (p1: Person) (p2: Person) :=
    let mv1 := Person_getMovies p1 m in
    let mv2 := Person_getMovies p2 m in
    match mv1, mv2 with
     | Some (h1:: t1), Some (h2:: t2) ⇒
         NatUtils.geb (length (intersect (h1::t1) (h2::t2) beq_Movie)) 3
     | _, _ ⇒ false
    end.
```

Listing 6.2 – CoqTL definition for checkings if two persons are a couple

Listing 6.3 – Extracted Scala code from the Coq function `areCouple`

```
def areCouple( p1: Person, p2: Person, model: MovieModel): Boolean = {
    val mv1 = MovieMetamodel.getMovies(model, p1)
    val mv2 = MovieMetamodel.getMovies(model, p2)
    (mv1, mv2) match {
        case (Some(h1::t1), Some(h2::t2)) =>
          intersect(p1_movies, p2_movies) >= 3
        case _ => false
    }
}
```

6.3 Configurable SparkTE

SparkTE was designed from `Parallelizable CoqTL`, a Coq specification defined from `CoqTL`. The engine proposes a single execution program, taking a source model, that conforms to a source metamodel, as input and producing a target model as output, that conforms to a target metamodel. Thanks to the proposed workflow, SparkTE can execute certified model transformation on Apache Spark. However, certifying software has a cost. Indeed, because of correction needs, SparkTE cannot really take advantage of the execution environment, i.e., the JVM, nor the framework it is designed with, i.e, Apache Spark. To tackle this lack of performance, we propose `Configurable SparkTE`, a transformation engine based on SparkTE that also considers a configuration conforms to a feature metamodel as input. The input of Configurable SparkTE programs remains the same, and produces the same output as SparkTE. Figure 6.2 illustrates the structure of a model transformation using Configurable SparkTE. The main difference with SparkTE is the absence of formal semantics, making the new engine not running certified model transformation. It would be possible to prove the semantic equivalence, for each feature, between all the proposed options, but proving is a long and tedious task. Proposing a fully proven equivalent configurable engine represent a part of future work.

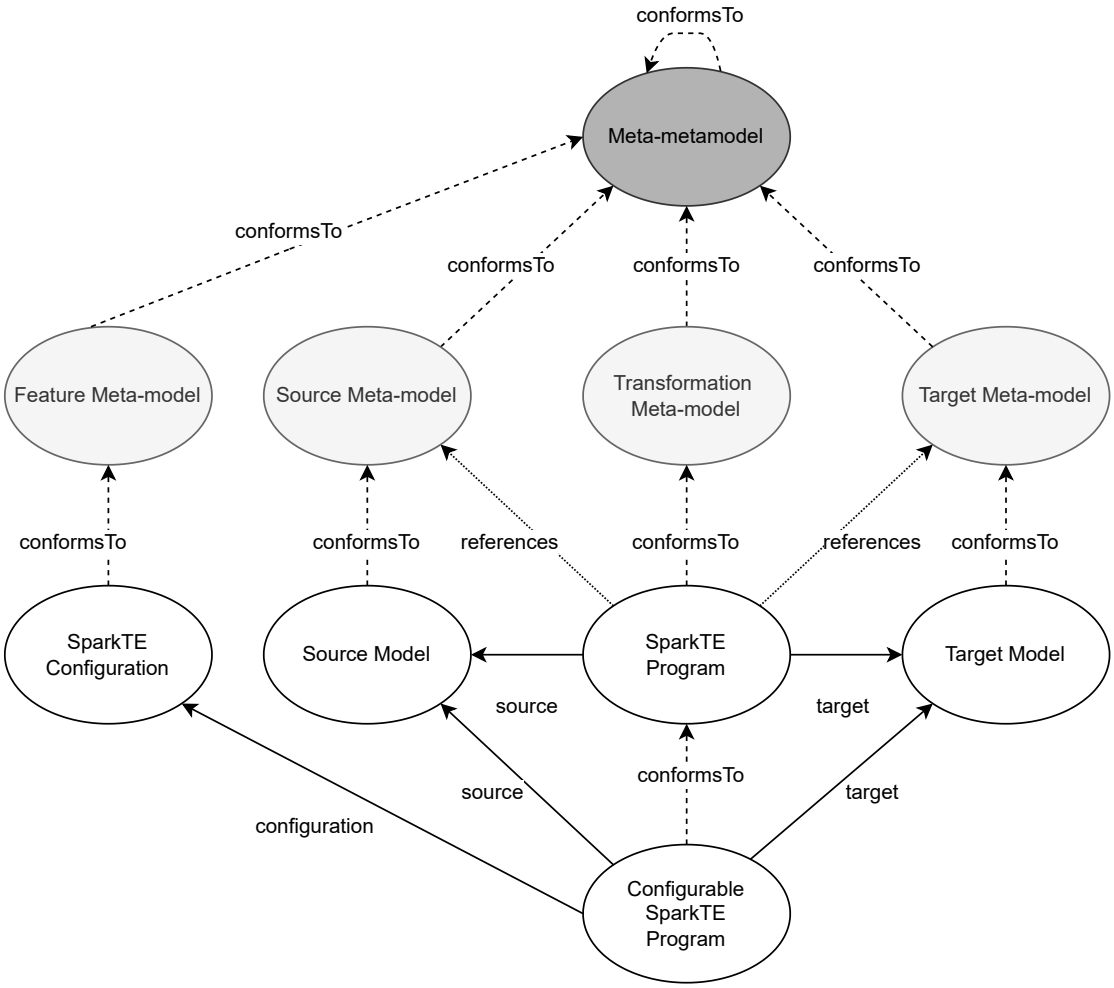


Figure 6.2 – Structure of a model transformation using Configurable SparkTE program

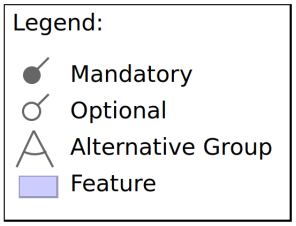


Figure 6.3 – Legend for feature diagrams

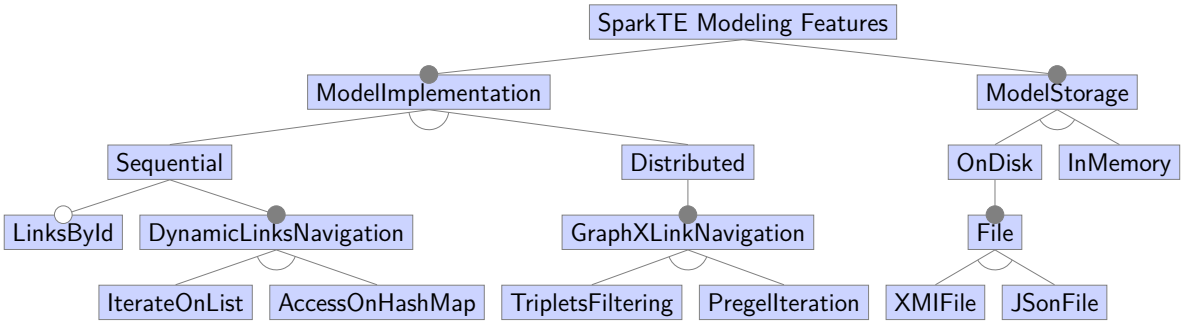


Figure 6.4 – SparkTE feature diagram for its modeling solution

6.4 SparkTE feature model

Next to a correct implementation of Parallelizable CoqTL in SparkTE, using functional structures (e. g. *List*) and pure functional features, we introduced additional solutions in SparkTE. Firstly, the representation of the model and its navigation present several approaches, based on different data structures and different approaches for representing links. Secondly, each part of the execution might follow a different strategy. As illustrated previously in the new specification of CoqTL, the instantiation of elements and links might depend on the purpose: having a solution for reasoning, or a solution for increasing parallelism opportunities. Finally, since SparkTE is based on Spark, some features are Spark-related. We discuss the latest later in the section. The legend that stands for all the feature diagrams of this Section follows the legend of Figure 6.3.

6.4.1 Modeling approaches

In SparkTE, models follow a very generic specification stating that models must only implement two functions: one to get elements, and one to get links. The concrete implementation is a couple of two sets: one for the elements, and one for the links. Figure 6.4 gives an overview of the features described below.

Links in SparkTE models can either be represented as a tuple of elements with a label, or only using their IDs. The latest stores less information and does not duplicate the objects during the distribution, only their IDs, but it leads to the need of resolution when users want to access the content of the objects from a link.

To tackle memory issues that can be faced when dealing with very large models (VLM), it is possible to distribute these two sets using RDDs and create a graph by GraphX. Once the model is distributed, it can be transparently queried by the user, using expressions as explained in Section 4.4.

On the one hand, the sequential solution offers two possible ways to navigate among the links of the model. First, links can be reached by iteration on the full set. This operation only shows benefits on very small model, since the browse of data is made instantly.

```
def iterateOnList(model: SourceModel, source: Id, type_: String): Link = {
   for (link in model.getAllLinks){
      if (link.type == type_ && link.source == source)
         return link
   }
   throw new Exception(...)
}

def accessOnHashMap(model: SourceModel, source: Id, type_: String): Link =
   model.getLinks.get(source).get(type_)
```

A second approach is to store links in a *HashMap*, using elements and types of links as keys. Accesses are direct, but creating such a structure requires additional operations with a CPU cost. It necessitates the composition of several `groupBy` and `map` operations. A Scala snippet to instantiate a map, from a list of links, is the following.

```
def makeMap(allModeLinks: List[Link])
: immutable.Map[Id, Map[String, List[Link]]] =
   allModelLinks.groupBy(link => link.getSource)
                .map(t => (t._1, t._2.groupBy(link => link.getType)))
```

Also, in Scala, *HashMap* keys are managed in a non-linear structure (a tree). It aims at improving the speed of accessing data, but requires an additional amount of memory.

On the other hand, distributed models take advantage of the distributed graph structure and allow the user to use several computational models for link navigation. It can either be using Pregel iteration, with the propagation of messages to get only a subset of links, or by

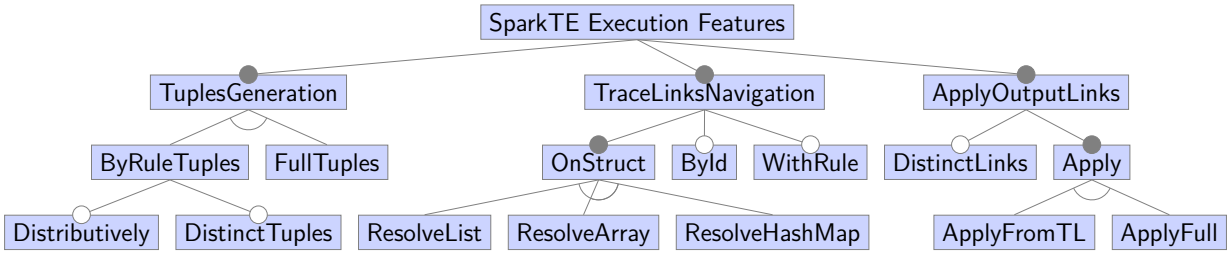


Figure 6.5 – SparkTE feature diagram for its execution strategy.

filtering the distributed set of edges represented as $triplets$ composed by a source, a label, and a target. Note that in the distributed case, links are represented by edges, are always built from the IDs of elements. However, their representation as triplets in Spark gives direct access both to the element ID, and the element itself.

Finally, the model can be stored at different level on the machine. In a sequential implementation of the model, i. e. when the model is fully duplicated on each computational node, the storage must be explicitly handled. SparkTE supports two modes: the model can be fully loaded in memory or kept on disk using XMI or JSON files. Distributed data with Spark can be natively stored at different levels by specifying a $StorageLevel$. This approach is discussed later. In future work, we consider storing the model in a remote database, for limiting the amount of data loaded in memory, but this approach would increase computation time because of the necessary communications.

6.4.2 Execution strategies

Section 5.4 has shown there exist semantics for performing a model transformation in SparkTE. In the implementation of Parallelizable CoqTL, we put a lot of importance on correctness. In the following features, we do not consider correctness aspects, but only focus on performances. Figure 6.5 gives an overview of the features described below.

The first feature, the tuples generation, can follow different strategies. As explained before, generating all the tuples is not always necessary. However, it tackles imbalance in the distribution of data since all tuples have the same weight. In the other case, generating the tuples from the types of the rules, reduces the total number of tuples. Note that if several rules have the same input types, duplicates might be found in the generated tuples. As an option, we also propose to add a distinct operation which might add an additional cost of computation. Lastly, if the input models are very large, tuples can be generated using a Cartesian product of RDDs. This approach starts to be very inefficient for tuples of size bigger than

2.

The second feature is the choice of the type of structure used for storing the generated trace-links at the end of the instantiate phase. These trace-links will be used to resolve output elements in the apply phase, to create links. We propose three data structures:

— a List, that is naturally made by construction during the instantiate phase. This approach is simple but terribly inefficient since elements cannot be accessed by index. Resolution is conducted by exploring the full list;
— an Array, that is collected from the RDDs during the distribution of computation in the instantiate phase. As for lists, the resolution is costly and inefficient since the index of the array does not consider the stored elements, only their positions. However, Java uses less memory for storing an array instead of a list;
— a HashMap with the input element as keys, and a list of output tuples as value, each containing a rule name, a pattern name, and a list of output patterns. This solution is the fastest to use, but necessitates additional computation to build, and more memory for storing the keys.

By default, all these structures store the concrete elements and their associated output. It is also possible to only consider the identifiers of the elements to save memory.

To improve the balance of computation, the trace-links can additionally store the name of the used rules to build output elements. Instead of distributing only the input patterns in the apply phase, SparkTE distributes couples of pattern and rule. This solution reduces imbalance in the case where a single pattern matches several rules. Instead of having one node dealing with all the output of this pattern, several nodes can manage to create its related output links. Concretely, a trace-link stores an additional string corresponding to the rule name, increasing the needed memory allocated for the trace-links, but also reducing imbalance in some cases.

Finally, the second part of the computation, designed in the apply phase, can be executed from trace-links or from scratch, as described in Section 5.4. The first solution reduces the global amount of computation by avoiding a second instantiation of output elements. However, it imposes a synchronization barrier called by a gather operation to collect all trace-links in the master node. Depending on the available amount of memory on the master node, this solution can slow down the computation. Recomputing the full links from scratch, including the application of instantiate part of rules to input patterns, allows a fully distributed computation. This approach takes advantage of the large amount of computational resources available in a cluster. The two approaches might lead to duplicate results.

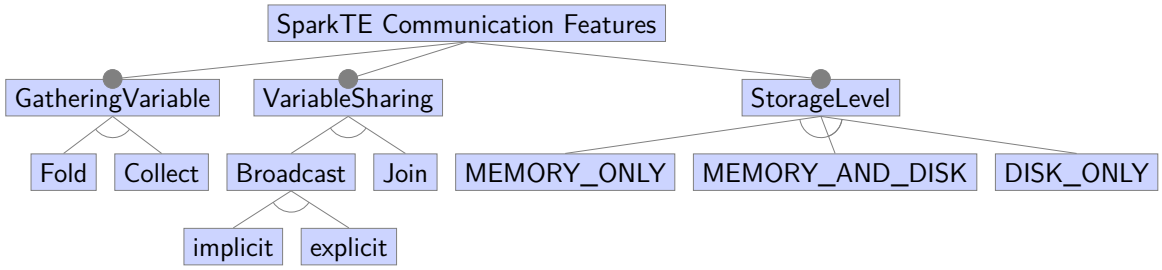


Figure 6.6 – SparkTE feature diagram for communicating data among nodes.

That is why we also propose to apply a distinct operation on the resulting links. It adds more computation time but reduces the size of the result.

6.4.3 Spark-Related Features

Spark has more than 200 parameters to configure an execution environment. Existing work has illustrated how challenging it is to configure such an environment [109]. In SparkTE, we propose two internal configurations of Spark (Figure 6.6):

— The storage level of the distributed structures varies according to the available resources. Indeed, a small amount of memory would force a user to prefer disk usage instead of keeping all the data in memory. Spark proposes to define, for each RDD, a storage level that can be: only memory, only disk, or a hybrid solution. The latest favor memory usage, but start swapping on disk to avoid out-of-memory errors. In addition, it is possible to duplicate the data on several nodes (up to 3), ensuring higher fault tolerance and reliability. Finally, a user can specify Spark to serialize the content of RDDs on nodes to reduce its size. This additional computation increases the distribution time of data (caused by serialization operations), but profits to machines with a small amount of memory. Table 6.1 shows a summary of all the impacts of StorageLevel.

— Communications in Spark are mostly implicit. Contrary to other libraries (e. g. MPI [148]), the library gives a very high level of abstraction, where managing parallelism details such as the concrete distribution of chunks among nodes or the concrete communication is not doable from the user perspective. More generally, all the underlying computations are optimized by taking advantage of functional composition equivalences [79]. However, the broadcast and the gather operations, that is sending data to all nodes and getting back all the results into a single node, are managed from the user perspective. The first can be implicit (by calling a variable from the sequential scope

StorageLevel	Space used	CPU time	In memory	On-disk	Serialized
MEMORY_ONLY	High	Low	Yes	No	No
MEMORY_AND_DISK	High	Medium	Some	Some	Some
DISK_ONLY	Low	High	No	Yes	Yes

Table 6.1 – Impact of Spark StorageLevel on a distributed data-structure management

in a RDD) or explicit using the $broadcast$ Spark operator. An additional solution is to use the $join$ operators on RDDs, to duplicate the content of a RDD on each node. The $boradcast$ operation is preferred for small amount of data to share, while the $join$ operator is designed for larger amount of data. The second is usually processed by calling a $collect$ operation at the end of a chain of distributed computation. This operation simply gathers all results into an array on the master node of the cluster. It is also possible to use a reduction operation, called $fold$, to gather results in parallel into a single value. The latter can also be used as a collect operation.

6.5 Evaluation

In the previous sections, we introduced a parametrizable transformation engine proposing different options for its features. These new strategies for navigating data, and conducting a model transformation, aims at improving the performance of SparkTE. Contrary to the strict implementation of `Parallelizable CoqTL` semantic, the new implementation does not target an engine fully based on correctness. As expected, the results presented in this section show significant improvement in performance. First, some of the additional features decrease the total computation time of a transformation. Second, as a side-effect, the better performance provided by the combination of new features lead to a better horizontal scalability, that is the capacity of dealing with models that have an increasing size. In this section, we target models conforms to the IMDB metamodel presented in Figure 6.1. To evaluate the performance of SparkTE parametrized with a given configuration, we perform two transformations on IMDB models:

— A `Identity` transformation, where the output model is simply a copy of the input model. This transformation has low complexity, and is used to show the horizontal scalability capacity of SparkTE on VLMs.

— The `FindCouples` transformation, presented in Listing 6.1. This transformation uses costly operations

Feature	Configuration 0 (C0)	Configuration 1 (C1)	Configuration 2 (C2)
ModelImplementation	Sequential	Sequential	Sequential
LinksById	false	false	false
DynamicLinkNavigation	undefined	IterateOnList	AccessOnHashMap
ModelStorage	InMemory	InMemory	InMemory
TuplesGeneration	ByRuleTuples	ByRuleTuples	ByRuleTuples
Distributively	false	false	false
DistinctTuples	false	false	true
TraceLinksNavigation	undefined	ResolveList	ResolveHashMap
byId	false	false	false
withRule	false	false	true
DistinctLinks	false	false	true
GatheringVariable	Collect	Collect	Collect
VariableSharing	Broadcast	Broadcast	Broadcast
StorageLevel	MEMORY_ONLY	MEMORY_ONLY	MEMORY_AND_DISK

Table 6.2 – Examples of configurations of SparkTE

All the experiments of the current Section have been run on the cluster `gros` (Nancy), from the G5k architecture: 2× 18 cores/CPU, *Intel Xeon Gold 5220*, 96GB of memory, with a network speed of 2× 25Gbps.

Table 6.2 sets three different configuration environment, set from the feature models (Fig 6.4, 6.5 and 6.6), for running a model transformation on Configurable SparkTE. Running all possible transformations is however not realistic. In this section, we only focus on some features, or on the comparison of two possible configurations. Appendix B gives an attempt of the design of a benchmark for running experiments for distributed multi-parameter applications.

6.5.1 Feature analysis

Links and trace-links navigation In this section, we apply our running example to a first model, with a fixed size of 100024 elements and 251732 links. We used 4 executors, with 4 cores each, to run the `Identity` transformation, on a in-memory model, with the configuration C0 of Table 6.2. We make varying the DynamicLinkNavigation feature, between IterateOnList and AccessOnHashMap, and the TraceLinksNavigation strategy between ResolveList and ResolveHashMap. The purpose of the experiment is an analysis of the impact on computation time of using one solution instead of another one. Table 6.3 illustrates the performance difference while using different data-structures for accessing input model links

DynamicLinksNavigation	TraceLinksNavigation	Computation time	Instantiate phase	Apply phase
IterateOnList	ResolveList	1636 sec	3 sec	1633 sec
IterateOnList	ResolveHashMap	1584 sec	3 sec	1581 sec
AccessOnHashMap	ResolveList	233 sec	6 sec	227 sec
AccessOnHashMap	ResolveHashMap	12 sec	6 sec	6 sec

Table 6.3 – Computation time of an `Identity` transformation on a IMDB model with varying data-structures type for accessing input model links and resolve instantiated trace-links

and resolve instantiated trace-links in the apply phase.

Since any of the links nor the trace-links are queried in the instantiate phase, its computation time remains almost the same. The slight difference is due on the construction of a map for the trace-links, that will be used in the second computation phase: the apply phase. However, in the second phase, where output links are created, the engine queries both the links and the trace-links. The results show that using ResolveList or ResolveHashMap on the trace-links has no significant impact on the computation time of the apply phase while using a list for accessing input links. However, accessing links using a map largely decreases the computation time of the second phase. In addition, coupling the ResolveHashMap strategy with AccessOnHashMap also shows significant benefits which was not the case with coupling the ResolveHashMap and IterateOnList.

Spark-related features In additional experiments, we switched options for Spark-related features. We conducted the same experiments as above, using the same model, the same `Identity` transformation, and the same computational environment (*gros* cluster on G5k). As solutions for navigating links and trace-links, we used AccessOnHashMap and ResolveHashMap. On one side, we experimented the impact of switching communication primitives of Spark both for sharing data, i.e., the use of $broadcast$ (implicit or explicit) or the use of $join$, and for gathering data, i.e., use of $collect$ or $fold$. Similarly, we conducted experiments changing the StorageLevel mode of RDDs. Any of these results have shown significant differences. This absence of difference is unexpected, especially for the StorageLevel option. Further investigation is a part of future work.

#elements	#links	Running time (Configuration 1)	Running time (Configuration 2)
1000	3000	9.799 sec	4.978 sec
2500	7300	81.047 sec	7.803 sec
5000	15000	882.708 sec	19.127 sec
7500	22000	> 2h	36.928 sec
10000	45000	Timeout error	65.198 sec

Table 6.4 – Comparison of two running configuration of SparkTE on `FindCouples` example

6.5.2 Configuration comparison

In this section, we compare two specific configurations. The first, referred as C1, corresponds to the SparkTE implementation, the one described in Chapter 5. The second, C2, is the one promising the best performance for the `FindCouples` transformation, according to the previous experiments. To run the experiment, 2 machines from the cluster `gros` have been used, with 4 working cores each. The experiment has been conducted on several models, from 1000 elements and 3000 links, to 10000 elements and 45000 links. These models correspond to real data from the IMDb database.

The results have shown that, with slight modifications, a new configuration might have a deep impact on performance. This gap is highly illustrated with larger models, as shown in Table 6.4.

First, we observe that running the transformation with the first configuration is always longer than running the same transformation with configuration 2. For a model of 1000 elements, it takes 2 times longer to end the transformation with C1. For a larger model, with for instance 10000 elements, it is not even possible to run the transformation using C1 without a timing error. In theory, no error should happen, the computation would just take a very long time, but the machine allocation in G5k is limited by time. Secondly, we observe that the computation time grows faster with C1 for increasing size models. We further discuss horizontal scalability below.

6.5.3 Horizontal scalability

Considering the configuration C2, which has shown the best performance results, we conducted experiments to evaluate its horizontal scalability, that is the capacity of the engine to deal with models with increasing size. To have pure results, that are not impacted by the

Size	100k elements	1M elements	2M elements
Computation time	21 sec	228 sec	432 sec

Table 6.5 – Horizontal scalability of running `Identity` on SparkTE with good configuration

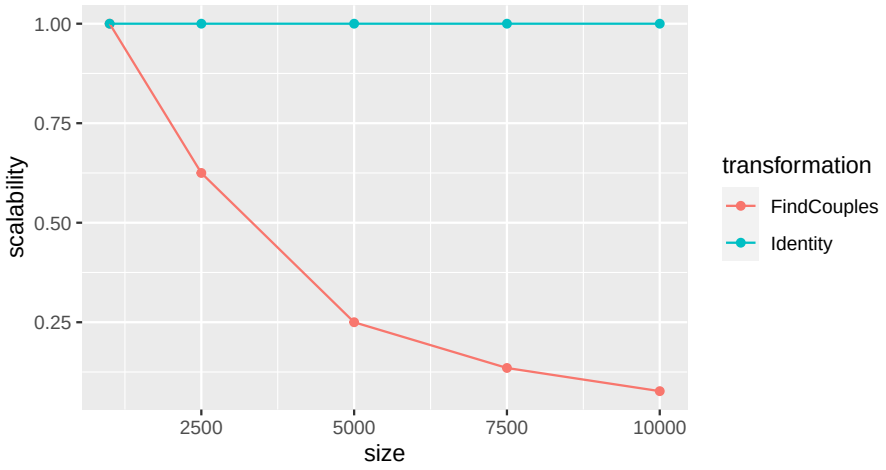


Figure 6.7 – Comparison of horizontal scalability for the Identity and FindCouples transformations on IMDb models

complexity of the transformation, we ran the `Identity` transformations on IMDb models of different sizes from 100k elements, to 2M elements. The resulting computation time are presented in Table 6.5.

The experiment shows perfect horizontal scalability for the `Identity` transformation. If we consider a more complex transformation, such as the `FindCouples` transformation, we observe an impacted horizontal scalability. Indeed, in Table 6.4, we observe that increasing the input model size, also decreases the horizontal scalability of the engine. The `FindCouples` transformation aims at comparing actors two-by-two to find who are forming a couple. The complexity the transformation is exponential on the number of actors, then it is expected to observe a largely decreasing scalability for such an example. Figure 6.7 illustrates the difference of the two horizontal scalability results for the two transformations. The scalability values are relative to the computation time of the transformations on a model of 1000 elements.

6.6 Conclusion

In this chapter, we proposed a configurable transformation engine designed on top of Spark, namely Configurable SparkTE. We have proposed the feature diagram for all proposed options, and have illustrated the benefits of the new proposed implementation, compared to the previous version of SparkTE. Finally, we have analyzed the impact of some engineering choices in the space-design of a transformation engine.

In future work, we plan to continue experiments by providing a complete analysis of the Configurable SparkTE features. We would like to compare all possible configurations, or at least a significant subset, to highlight the impact of the implementation choices, individually or combined. Finally, we want to face the configurable aspects of different kinds of input: different input models with different topologies (size, depth, connectivity), and different transformations.

CHAPTER 7

CONCLUSION

Contents

7.1 Synthesis

Model transformation is an important technique in the domain of Model-Driven Engineering. It allows the automatic mapping between two different representations of data. Two main components are involved in the model transformation. First, languages have been designed to express transformation rules to help users to write such a mapping. Second, engines compute the transformations as defined in the rules. Due to the always increasing amount of data, transforming models becomes challenging, and necessitates new strategies to tackle the lack of scalability. Among these strategies, parallelism and distributed computation have been revealed as an efficient solution, proposing to allocate additional resources to the execution of a transformation. Solutions have been designed for this purpose. However

these solutions are not comparable, and doing so necessitates an expertise in all of them. Also, designing his own transformation engine requires a global knowledge of existing features to propose an adequate software solution.

In this thesis, we proposed to contribute to the analysis of the design-space of a distributed transformation engine, namely SparkTE. SparkTE is a transformation engine, designed from a formal specification, CoqTL, on top of Apache Spark, a multi-language engine for executing data engineering. Our proposal gives an analysis at the different levels of the conception of our transformation engine.

First, we evaluated different programming models based on the distribution for evaluating queries on models. Queries are the concrete execution of expressions written in transformation rules to interrogate a model to either check a condition, or extract information for creating output content. We initially proposed a general implementation for a constraint language, based on a skeletal approach, and additional executions for a specific query, taking advantage of several programming models: MapReduce, Pregel, and hybrid solutions. The results have shown major performance differences, depending on the input model size and topology.

Second, we have studied the semantics of a transformation engine. From CoqTL, an internal DSL aiming at reasoning on model transformations, we proposed Parallelizable CoqTL. Our new specification aims at refining the former formalization to enhance parallelization opportunities in its implementation, and proposes three optimizations. These optimizations make the transformation defined as two highly-parallelized phases, with the second reusing results from the first. We formally verified that the transformation semantic in Parallelizable CoqTL preserves the one previously defined in CoqTL. We extracted, by-hand, the Coq code of our specification, to obtain a Scala transformation engine, preserving the semantic formally defined in the Coq definition. Finally, by replacing sequential lists with distributed structures, i.e., Spark RDDs, we obtained SparkTE, a transformation engine proposing distributed execution without sacrificing correctness. We studied the impact of the three design choices we made, and results have shown a gain of performance and promised scalability for long computation on small models.

We also have conducted a design-space analysis by proposing a parametrized version of SparkTE. Sacrifying correctness, we have been able to propose alternative execution for model transformations that proposes better perfomance by reducing the global computation time. We have studied and optimized the scalability of the transformation engine, by switching between configurations defined in a feature diagram.

7.2 Limits

Several limits were encountered during the research presented in this thesis, which should be considered in future works. We describe them in what follows.

7.2.1 Horizontal scalability

At the initial computation phase, the full model is stored on a single node, which means it needs to be either distributed, or fully communicated to share its content with other computational nodes. In both cases, a part of the performance really depends on the capacity of the master node to deal with a large amount of data. In theory, having an already distributed model as input would allow SparkTE to deal with models of infinite size. Besides, having an external solution for storing models, for instance distributed, would limit the memory overflow which leads to swapping with disk operations.

7.2.2 Correctness

Our approach also presents correctness weaknesses. Proving is a long and difficult task. It necessitates expertise to first formalize, and then prove properties. Coq proposes a proven extraction mechanism to obtain ML (e.g., Scheme, Haskell, OCaml) programs from Coq code. The extracted code formally respects the semantics of Coq code, and then the target code verifies all the defined properties in the Coq part. In our approach, we translated the Coq code to Scala code by hand. The only advantage of this approach is purely stylish: we have introduced OOP aspects, to ease the development of SparkTE. Also, since we only use pure functional features for the target engine, we did not change the semantics of the extracted code. A second limit of the correctness of SparkTE is that all the modifications that have been processed to obtain a parametrizable engine have not been formalized and proven not changing the semantics of the initial engine. Besides, some features are not included in the formalization (e.g., distributed aspects, non-linear structure usage) and would necessitate additional work to have a fully proven correct workflow.

7.2.3 Multi-parameters benchmarking

The number of options for each feature makes full experimentation of Configurable SparkTE impossible. Indeed, considering only the presented discrete features, the number of possible configurations is already around 100000. In addition, to have full experimentation, other

parameters must be considered: the input model size, and the execution environment (e.g., number of machines, allocated resources). On one side, we cannot experiment all the combinations to see their synergies. On the other side, we do not provide a solution for giving the best configuration for a given input.

7.2.4 Input analysis

We have shown that the transformation complexity and the model size have a deep impact on the scalability. Our analysis has been limited to: running a complex transformation on a small model shows good vertical scalability, while running a simple transformation on a large model shows good horizontal scalability. A deep analysis of both the input model and the complexity of transformation would allow us to determine what configuration, and what solution is the most adapted to the current execution.

7.2.5 Additional strategies

Our approach is built upon data-distribution and parallelism to enhance the performances of a model transformation on very large models. We have described additional solutions (e.g., incrementality, laziness) that improve performances for costly operations, but we did not integrate them to our solution. By default, Spark already proposes laziness aspects, thanks to its transformation-action mechanism but we did not analyzed its impact. Incrementality is not very suitable for data-distribution, since small modifications lead to small recomputation, which might be not worth it to execute on a large scale architecture. However, it could have been integrated as a side feature, proposing sequential operations for small model management operations. Finally, our solution only targets data-distributed approaches for performing a model transformation. No task-parallelism, nor asynchronism was considered as a solution. To have a complete analysis of performant transformation engines, these other approaches must be integrated, and compared.

7.3 Perspectives

We mention along the thesis possible perspectives for our future work. In what follows we sum up these perspectives.

7.3.1 Storing models in external database

The first perspective concerns the external storage of the models. Transforming VLMs is challenging. To tackle the scalability problems described in Section 7.2.1, we plan to use an external solution for storing the model. A first approach will be to use a database for storing the model. Doing so would not overload the memory of the master node of the Spark cluster on which a SparkTE program is running. Besides, some treatments could be processed on the database server side, like for instance the generation of possible input patterns as tuples for the instantiate phase of the transformation. Thanks to the numerous connector that have been developed for Spark, that are Spark APIs for interacting with data source, we could experiment different external solutions, including distributed databases [23, 177]. Our solution would work like NeoEMF [61, 63], as an interface between a modeling solution and an external data source. The advantage of using such a solution is the structural properties of databases, which can easily represent model elements and their relations. Data warehouses [24] are a second targeted option for storing models. Contrary to databases, the content of data warehouses is not necessarily structured, but promises very good performance. For instance, Meta efficiently used Hive [162, 161] for its social network Facebook [160].

7.3.2 A compiler CoqTL to Scala

In future work, we plan to write a compiler from CoqTL to Scala, to automatically obtain an executable transformation from a Coq specification. So far, the extraction is made by hand, but existing solutions such as [67] can be a basis of this future work. Concretely, our compiler would need (i) a translation of Coq classes into Scala interfaces, (ii) a functional proof of equivalence between Coq and Scala definitions for functions, (iii) the extraction of Coq semantics definition into Scala definitions.

7.3.3 Cost model for distributed operations

To evaluate a model transformation solution, we plan to propose an evaluation of model operation complexity with a baseline where transformation primitives are evaluated, and combined to deduce a global complexity for a given transformation. Then, depending on the target engine, the global complexity can be different. It is doable to compare sequential engines (e.g, formalized in Coq, like it is done with CoqTL). It is also possible to compare the parallelizable semantics of engines, but expressed in a sequential manner. However, there is a gap between the sequential analysis and the parallel analysis that represent a part of our

future work. There exist cost models for evaluating parallel tasks (e.g., BSP [170], Amdhal's laws [3]), that could be combined with the rule complexity evaluation to be able to compare two distributed operations. Our future analysis will separate the transformation complexity and the engine complexity, to propose a clear decision process for picking the right transformation semantics which is independent from the used engine.

7.3.4 Evaluating SparkTE features

This fourth perspective is the consequence of the previous one. Evaluating SparkTE features will be eased by the provision of the complexity of model transformation operations. To compare all possible configurations of Configurable SparkTE, we first want to compare features independently, but the consequence of combining some of them by evaluating their synergy. Ideally, we want to run the cartesian product of all the options for the parameters, and group by results by pair, triplets, etc. From these results, and by conducting deep statistical analysis, we plan to provide an automatic configuration, and reconfiguration, of our engine. First, we will be able to give an order to the parameters of the engine according to the impact they have on the engine performance. Second, using game-based strategies (e.g., Monte-Carlo approach [32]), we will find the best configuration input for our distributed transformation engine.

7.3.5 Monitoring SparkTE

Finally, we want to open our engine analysis to other perspectives. We plan to conduct monitoring of SparkTE jobs activity at different levels:

— **Memory consumption**: The computation time depends on many factors including memory consumption. Since we are using distributed architectures for running model transformations, analyzing precisely the memory usage of our jobs would allow better resource management within a running cluster.

— **Energy consumption**: Considering the current ecological challenges our world is facing, stressing the impact of running large computations must be addressed [120]. For instance, we could evaluate heat production using sensors [60]. Besides, model-based approaches could be used for contributing to energy-aware software engineering in the context of model transformation [25, 26].

Appendices

APPENDIX A

EFFICIENT LOADING OF SERIALIZABLE MODELS

Handling very large models rapidly become challenging due to the capacity of tools to deal properly with a big amount of data. While several solutions to persist EMF models exist, most of them do not allow partial model unloading and cannot handle models that exceed the available memory. Furthermore, these solutions do not take advantage of the graph nature of the models: most of them rely on relational databases, which are not fully adapted to store and query graphs. Neo4EMF [17] is a persistence layer for EMF that relies on a graph database and implements an unloading mechanism. The main purpose of NeoEMF is to face scalability issues on large-scale models [62]. On the side, additional modules allow developers to use additional persistence solutions. For instance, NeoEMF I/O deals with file solutions, as the standard XML Metadata Interchange (XMI) files. Moreover, distributed solutions whose purpose is to access and interact efficiently with large-scale models, take advantage of clustering several machines largely increasing the quantity of resources allocated to a computation. The multi-language engine for executing data engineering Apache Spark, distribute objects by streaming

We proposed an extension of NeoEMF for dynamically loading objects, independently of EMF artifacts. Figure A.1 gives an overview of how we defined a side ecosystem to load models using the NeoEMF I/O module.

A.1 Contribution to NeoEMF

Figure A.1 describes the integration of NeoEMF in modeling solutions ecosystem. Standard modeling users use model-based applications which provide high-level modeling features such as a graphical interface, interactive console, or query editor. These features internally rely on EMF's Model Access API to navigate models, perform CRUD operations, check constraints, etc. Modelers might also want to distribute their computations to improve per-

formances and increase computing capacity. Depending on the targeted solution, the input differs in how it is interpreted by the engine. Indeed, most Java frameworks for distributed computing deal with objects who must implement the `java.io.Serializable` interface. For instance, SparkTE [138], a model transformation engine based on Spark [1], designed in the context of the Lowcomote project, deals with Java serialized elements for communicating between computational nodes. In the other case, that is using model-based tools using EMF modeling solution, EMF delegates the operations to a persistence manager using its Persistence API, which is in charge of the serialization/deserialization of the model.

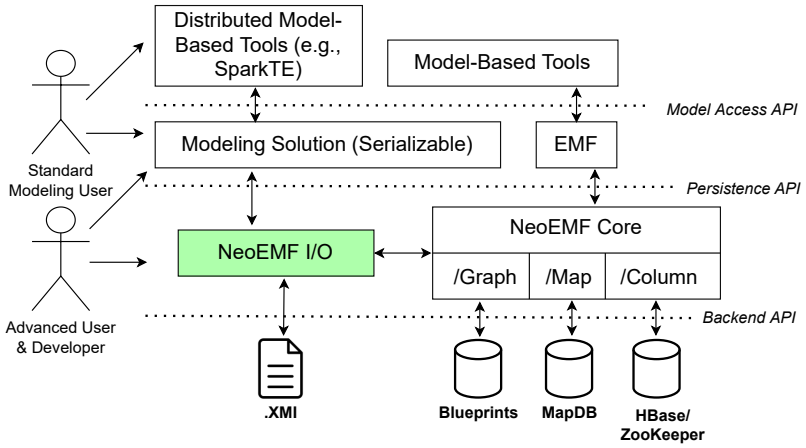


Figure A.1 – NeoEMF extension integration in a model-based development environment

Once the core component has received the modeling operation to perform, it forwards the operation to the appropriate database driver (Map, Graph, or Column), which is in charge of handling the low-level representation of the model. These connectors translate modeling operations into Backend API calls, store the results, and reify database records into EMF `EObjects` when needed. Our contribution aims at providing EventListener4Serializable an abstract class in the NeoEMF I/O module, that is completely independent of EMF core components. A NeoEMF EventListener defines the behavior for parsing elements as they are read during in file reading. To use the features of our new listener, a user define two behavior: (i) one for element, overriding createElement and (ii) for links overriding createLink. An overview of this listener and a concrete implementation for the case presented in Section 4.2 are respectively presented in Listing A.1 and Listing A.2.

Listing A.1 – EventListener4Serializable interface

1. https://spark.apache.org

```
abstract class EventListener4Serializable[E <: Serializable, L <: Serializable]
extends EventListener {
  ...
  def createElement(classname: String, id: Long): E
  def createLink(referenceName: String, parentId: Long, targetIds: List[Long]): L
}
```

Labels for class names and reference names are defined as constants, corresponding to the names that are used in the metamodel. Also to allow both the distribution of elements and links, to have a Spark distributed graph, links must be defined as a triplet of a source id, a label and a target id. The links structure is independent of elements, but must use valid ids, existing in the rest of the model.

Listing A.2 – An example usage of EventListener4Serializable for TTC18 case

```
class SocialNetworkListener
extends EventListener4Serializable[SocialNetworkElement,SocialNetworkLink] {

  override def createElement(classname: String, id: Long)
  : SocialNetworkElement = {
    classname match {
      case USER => new SocialNetworkUser(id)
      case POST => new SocialNetworkPost(id)
      case COMMENT => new SocialNetworkComment(id)
      case _ =>
    }
  }

  override def createLink(referenceName: String, parentId: Long, targetIds:
     List[Long])
  : SocialNetworkLink = {
    referenceName match {
      case COMMENT_LIKEDBY => new CommentLikedBy(parentId, targetIds)
      case COMMENT_POST => new CommentPost(parentId, targetIds.head)
      case COMMENT_SUBMISSION => new CommentSubmission(parentId, targetIds.head)
      case SUBMISSION_COMMENTS => new SubmissionComments(parentId, targetIds)
      case SUBMISSION_SUBMITTER => new SubmissionSubmitter(parentId,
         targetIds.head)
```

```
22        case USER_FRIENDS => new UserFriends(parentId, targetIds)
23        case USER_LIKES => new UserLikes(parentId, targetIds)
24        case USER_SUBMISSIONS => new UserSubmissions(parentId, targetIds)
25        case _ =>
26      }
27    }
28  }
```

A.2 Experimental results

This section presents the time it takes for loading models with different sizes and complexity. These times are compared to rhe times to load the same models using the native EMF resource loader. The experiments have been conducted on a shared memory machine with a Intel Core i7-8650U having 8 cores at 1.90GHz and a memory of 32GB. The machine was running Ubuntu 22.04 LTS. We use Java 13, Scala 2.12 with Spark 3.1.0. The used model and their respective sizes are described in Table A.1. The smallest model is composed 1274 elements and 2163 links while the biggest is made with 859114 elements and 1589908 links. Each recorded timing is the average of 30 executions.

Dataset							**Loading time**	
id	#elements	#links	#users	#posts	#comments	#likes	ext. NeoEMF	native EMF
1	1274	2163	80	554	640	6	61ms	69ms
2	2071	3548	889	1064	118	24	93ms	184ms
3	4350	7594	1845	2315	190	66	146ms	450ms
4	7530	14422	2270	5056	204	129	270ms	878ms
5	15132	27886	5518	9220	394	572	413ms	3921ms
6	30396	56261	10929	18872	595	1598	491ms	19473ms
7	58076	111197	18083	39212	781	4770	1858ms	82535ms
8	115121	218823	37228	76735	1158	13374	2699ms	389953ms
9	224816	424901	1678	74668	148470	36815	3700ms	1711354ms
10	443323	812515	2606	167299	273418	102276	8863ms	> 5 minutes
11	859114	1589908	3699	314510	540905	2684	19201ms	> 15 minutes

Table A.1 – Loading time of models from TTC18

Our solution clearly outperforms EMF native loading mechanism. Because we only pass through the file once, loading elements as they are read, the computation time is proportional to the size of the file. On the contrary, EMF concretely resolves references making its loading mechanism time depending on the topology of the input model.

APPENDIX B

MULTI-PARAMETER BENCHMARK FRAMEWORK

Every non-trivial application has a large number of parameters, each of them having varying sizes of value sets, i.e., different numbers of values for those parameters. Some of these parameters influence the performance of the application, i.e., execution time, memory, disk or network use, which is of interest to the application's users. They would like to have the application finish in the shortest time, use the least amount of memory or limit the network traffic to a certain extent. Often these goals are conflicting with each other, however, finding the best parametrization is crucial to improve the user experience of the application. (Best parametrization is a concrete binding of the parameters to values with which the application has the best performance.)

To make the general goal more specific, we focused our research on Spark applications, i.e., software that can be deployed on a Spark cluster. SparkTE, that was introduced in Chapter 5, is such an application.

B.1 Research objectives

Finding the best parametrization in a naive way is a time-consuming endeavor, due to the exponential number of combinations that have to be checked (cross-product of the parameter value sets). A way to reduce these combinations is to use good filter functions that remove the unnecessary combinations, e.g., those who do not have an influence on the optimization goal or might yield equivalent results. Another way is to leverage the easy access to the large amount of computational capacity in the cloud, so we can yield the benchmark results faster by being able to test more parametrizations in parallel. However, this approach has high cost implications that we have to pay. To summarize, our research objectives are:

RO1 develop a multi-parameter benchmark framework to find the best parametrization of a Spark application according to a goal (i.e., execution time, memory or network

use),

RO2 use cloud infrastructure to test the parameter combinations to yield the results the fastest.

B.2 Motivating example

Let's take a simple word count application that counts the number of occurrences of each word in a text (corpus), illustrated by Listing B.1: Line 1 reads the file into a RDD in Spark. After that, in line 2, it splits the content of the file by white space, groups the words by occurrence and returns them in a descending order of occurrence. Finally, line 3 prints the most frequent word to the standard output. The parameters of the application are: the name of the file (`filename`), the replication factor (`replication`, on how many nodes the RDD will be copied on the cluster), the number of partitions (`partition`, how many RDDs the file's content will be split into).

Listing B.1 – Word count example Spark application.

```
1  val file: RDD[String] = nFile(filename, replicate, spark, partition)
2  val res = file.flatMap(line => line.split(" ")).map(word =>
       (word.replaceAll("[-+.^:,;)(_]", ""), 1)).reduceByKey(_ +
       _).sortBy(e => e._2, ascending = false).collect()
3  println(res(0))
```

In our example, we used a corpus with 857116 words (`bible.txt`), experimented with replication factors `1-2` and number of partitions `1-3`, as shown in Table B.1. We measured both the execution time and the memory use of the application in six different scenarios to check all possible combinations of `replication` and `partition`.

File name	Replication factor	Number of partitions
bible.txt	1, 2	1, 2, 3

Table B.1 – Word count parameters.

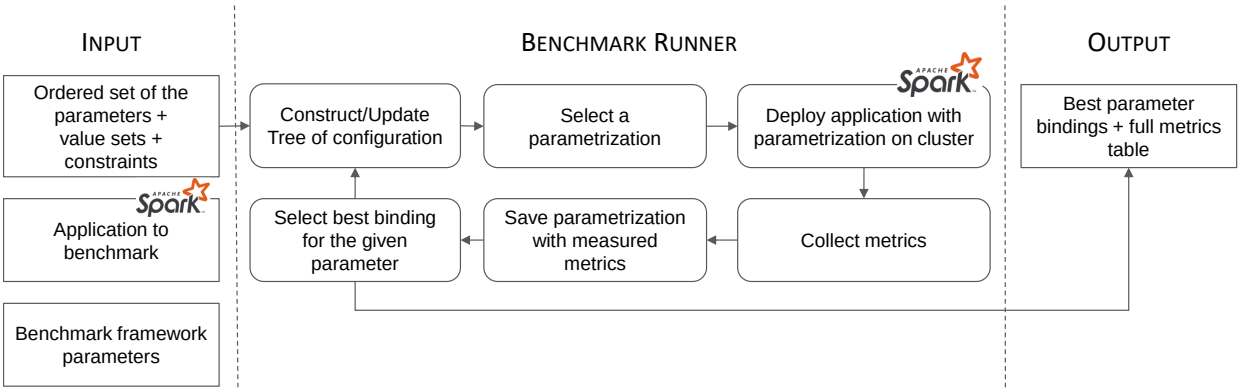


Figure B.1 – Global overview of the approach for a multi-parameter benchmark framework

B.3 Architecture & approach

Figure B.1 provides an overview of our benchmark approach. It takes as input:

— the Spark application to benchmark,

— the ordered set of parameters with their value sets, and some constraints describing the prohibited combinations of the parameter values (e. g. `Parameter A` cannot have value `a`', if `Parameter B` has value `b`'),

— the benchmark framework parameters: Spark cluster configuration, the number of warm-up and measurement rounds with each combination and the number of parameter combinations we want to benchmark.

We need the input parameters to be sorted by priority, i. e. the parameter that has the highest impact on the measurement goal (e. g. memory use, execution time or network traffic) should have the highest priority. Although several methods exist which help decide the prioritization (e. g. in an empirical way by learning from previous measurements, following theoretical assumptions, following industrial or research best practices), but it was out of the scope of our research to decide which is the best way to set the priorities. In our case, we used the experiences gained from previous measurements to decide the priorities among the parameters.

After the prioritization, we built a Monte Carlo tree to perform a Monte Carlo tree-like search [32] on the different combinations of the parameter values. During the construction of the tree, we removed those nodes that would give a forbidden combination of the parameter values (due to the constraints defined between the parameters in the input step).

In each benchmark round, we (1) select the next parametrization from the tree, (2) deploy the application with this parametrization on the cluster, (3) run the application and collect the metrics, (4) save the parametrization with the measured metrics, (5) when we have enough data collected then we save the concrete value (binding) of the parameter that yielded the best result for the given metric, and (6) update the tree with this information and move on to get the next parametrization. In order to avoid concluding results from just one measurement, each parametrization is measured in m number of measurement rounds, preceded by n number of warm-up rounds. The average of the metrics values measured in the m number of measurement rounds will be the metrics values saved for that parametrization. We repeat the benchmark loop until we experimented with all valid combinations of the parameters, or the number of combinations we wanted to benchmark. Finally, this algorithm returns the best parameter binding(s) for the given metrics and a CSV table with all measurement results.

Comparing benchmark results when multiple metrics are measured is a difficult task. Therefore, we give the opportunity to the user to define the comparison function to compare which measurement result is better in these cases. In our example (Section B.2), we measured both the execution time and the memory use of the application. However, we gave higher weight to the execution use, i. e. the lower it is the better, despite possibly yielding a higher memory use.

The benchmark framework is independent from the execution environment (computation cluster), on which the application is deployed in Spark. In the following subsection, we introduce a cloud infrastructure, Grid'5000 that can be used to run the benchmark on.

B.3.1 Evaluation

On G5k, we booked a single node with 64 GB RAM and Intel Xeon E5-2660 CPU at the Nantes site of the cluster and installed Spark on it. After that, we run the benchmark workflow on the motivating example (Section B.2) that concluded the following measurement results:

From the results in Table B.2, we can conclude that replication factor 1 with 2 partitions is the best parametrization of the running example in the given deployment environment, because it yields the shortest execution time (32 ms) despite a larger memory use (3267 MB) than the smallest one (3156 MB).

File name	Replication factor	# partitions	Memory use (MB)	Execution time (ms)
bible.txt	1	1	3156	33
	1	2	3267	32
	1	3	3494	44
	2	1	3878	44
	2	2	3436	43
	2	3	3334	44

Table B.2 – Benchmark results on the word count example.

B.4 Conclusion

We introduced a multi-parameter benchmark framework to find the best parameterization of a Spark application. We used a cloud infrastructure (G5k) to test the parameter combinations and found the best parametrization for our motivating example application. The prototype implementation of the framework is available on GitHub [1].

As future work, we are planning to extend the framework so that it is able to run the measurements in parallel, on different machines in the cluster. Thereby speeding up the overall execution time of the benchmark workflow. Besides, we will experiment with different applications and advertise the framework for a broader audience in the MDE and software engineering communities so that other software engineers can also benefit from our work.

1. https://github.com/lowcomote/multi-parameter-benchmark

BIBLIOGRAPHY

[1] Mathieu Acher et al., « Learning very large configuration spaces: What matters for Linux kernel sizes », PhD thesis, Inria Rennes-Bretagne Atlantique, 2019.

[2] Juliana Alves Pereira et al., « Sampling Effect on Performance Prediction of Configurable Systems: A Case Study », *in*: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 277–288, ISBN: 9781450369916, DOI: 10.1145/3358960.3379137, URL: https://doi.org/10.1145/3358960.3379137.

[3] G. M. Amdahl, « Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities », *in*: *IEEE Solid-State Circuits Society Newsletter* 12.*3* (2007), pp. 19–20, ISSN: 1098-4232, DOI: 10.1109/N-SSC.2007.4785615.

[4] Moussa Amrani et al., « Towards a Formal Specification of Multi-paradigm Modelling », *in*: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, ed. by Loli Burgueño et al., IEEE, 2019, pp. 419–424, DOI: 10.1109/MODELS-C.2019.00067, URL: https://doi.org/10.1109/MODELS-C.2019.00067.

[5] Moussa Amrani et al., « Towards a Model Transformation Intent Catalog », *in*: *Proceedings of the First Workshop on the Analysis of Model Transformations*, AMT '12, Innsbruck, Austria: Association for Computing Machinery, 2012, pp. 3–8, ISBN: 9781450318037, DOI: 10.1145/2432497.2432499, URL: https://doi.org/10.1145/2432497.2432499.

[6] Jason Ansel et al., « Opentuner: An extensible framework for program autotuning », *in*: *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[7] Ralf Ansorg and Lars Schwabe, « Domain-Specific Modeling as a Pragmatic Approach to Neuronal Model Descriptions », *in*: *Brain Informatics*, ed. by Yiyu Yao et al., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 168–179, ISBN: 978-3-642-15314-3.

[8] Danilo Ardagna et al., « MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds », *in*: *2012 4th International Workshop on Modeling in Software Engineering (MISE)*, 2012, pp. 50–56, DOI: 10.1109/MISE.2012.6226014.

[9] Thorsten Arendt et al., « Henshin: Advanced Concepts and Tools for in-place EMF Model Transformations », *in*: *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2010, pp. 121–135.

[10] University of Malaga Atenea team, *Lintra*, http://atenea.lcc.uma.es/projects/LinTra.html, 2018.

[11] Salman Azhar, « Building Information Modeling (BIM): Trends, Benefits, Risks, and Challenges for the AEC Industry », *in*: *Leadership and Management in Engineering* 11 (July 2011), pp. 241–252, DOI: 10.1061/(ASCE)LM.1943-5630.0000127.

[12] R.C. Backhouse, *An Exploration of the Bird-Meertens Formalism*, Computing science notes, University of Groningen, Department of Mathematics and Computing Science, 1988, URL: https://books.google.fr/books?id=zFqWGwAACAAJ.

[13] Ioannis Ballas et al., « On Exploring the Optimum Configuration of Apache Spark Framework in Heterogeneous Clusters », *in*: *25th Pan-Hellenic Conference on Informatics*, PCI 2021, Volos, Greece: Association for Computing Machinery, 2021, pp. 250–253, ISBN: 9781450395557, DOI: 10.1145/3503823.3503870, URL: https://doi.org/10.1145/3503823.3503870.

[14] Amine Benelallam, Abel Gómez, and Massimo Tisi, « ATL-MR: model transformation on MapReduce », *in*: *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems, SEPS SPLASH 2015, Pittsburgh, PA, USA, October 27, 2015*, ed. by Ali Jannesari et al., ACM, 2015, pp. 45–49, ISBN: 978-1-4503-3910-0, DOI: 10.1145/2837476.2837482, URL: https://doi.org/10.1145/2837476.2837482.

[15] Amine Benelallam et al., « Distributing relational model transformation on MapReduce », *in*: *Journal of Systems and Software* 142 (2018), pp. 1–20, ISSN: 0164-1212, DOI: 10.1016/j.jss.2018.04.014, URL: https://doi.org/10.1016/j.jss.2018.04.014.

[16] Amine Benelallam et al., « Efficient model partitioning for distributed model transformations », *in*: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November*

1, 2016, ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró, SLE 2016, ACM, 2016, pp. 226–238, ISBN: 978-1-4503-4447-0, DOI: 10.1145/2997364.2997385, URL: http://dl.acm.org/citation.cfm?id=2997385.

[17] Amine Benelallam et al., « Neo4EMF, a scalable persistence layer for EMF models », *in*: *European Conference on Modelling Foundations and Applications*, vol. 8569, Lecture Notes in Computer Science, Springer, 2014, pp. 230–241.

[18] Gábor Bergmann, István Ráth, and Dániel Varró, « Parallelization of graph transformation based on incremental pattern matching », *in*: *Electronic Communications of the EASST* 18 (2009).

[19] Gábor Bergmann et al., « Efficient model transformations by combining pattern matching strategies », *in*: *International Conference on Theory and Practice of Model Transformations*, Springer, 2009, pp. 20–34.

[20] Gábor Bergmann et al., « Incremental evaluation of model queries over EMF models », *in*: *International conference on model driven engineering languages and systems*, Springer, 2010, pp. 76–90.

[21] Gábor Bergmann et al., « Incremental Pattern Matching in the Viatra Model Transformation System », *in*: *Proceedings of the Third International Workshop on Graph and Model Transformations*, GRaMoT '08, Leipzig, Germany: Association for Computing Machinery, 2008, pp. 25–32, ISBN: 9781605580333, DOI: 10.1145/1402947.1402953, URL: https://doi.org/10.1145/1402947.1402953.

[22] Gábor Bergmann et al., « Integrating Efficient Model Queries in State-of-the-Art EMF Tools », *in*: *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, Prague, Czech Republic: Springer-Verlag, 2012, pp. 1–8, ISBN: 9783642305603, DOI: 10.1007/978-3-642-30561-0_1, URL: https://doi.org/10.1007/978-3-642-30561-0_1.

[23] Philip A Bernstein and Nathan Goodman, « Concurrency control in distributed database systems », *in*: *ACM Computing Surveys (CSUR)* 13.*2* (1981), pp. 185–221.

[24] Philip A Bernstein and Erhard Rahm, « Data warehouse scenarios for model management », *in*: *International Conference on Conceptual Modeling*, Springer, 2000, pp. 1–15.

[25] Thibault Béziers La Fosse et al., « Characterizing a source code model with energy measurements », *in*: *Workshop on Measurement and Metrics for Green and Sustainable Software Systems (MeGSuS)*, 2018.

[26] Thibault Béziers la Fosse et al., « Annotating executable DSLs with energy estimation formulas », *in*: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, 2020, pp. 22–38.

[27] Jean Bézivin, « On the unification power of models », *in*: *Software & Systems Modeling* 4.*2* (2005), pp. 171–188.

[28] Jean Bézivin and Olivier Gerbé, « Towards a Precise Definition of the OMG/MDA Framework », *in*: *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, USA: IEEE Computer Society, 2001, p. 273.

[29] R. Bird and O. de Moor, *Algebra of Programming*, Prentice-Hall international series in computer science, Prentice Hall, 1997, ISBN: 9780135072455, URL: https://books.google.fr/books?id=P5NQAAAAMAAJ.

[30] Richard S. Bird, « The promotion and accumulation strategies in transformational programming », *in*: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 6.*4* (1984), pp. 487–504.

[31] Gerth Stølting Brodal and Rolf Fagerberg, « Cache Oblivious Distribution Sweeping », *in*: *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, ed. by Peter Widmayer et al., vol. 2380, Lecture Notes in Computer Science, Springer, 2002, pp. 426–438, DOI: 10.1007/3-540-45465-9_37, URL: https://doi.org/10.1007/3-540-45465-9%5C_37.

[32] Cameron Browne et al., « A Survey of Monte Carlo Tree Search Methods », *in*: *IEEE Trans. Comput. Intell. AI Games* 4.*1* (2012), pp. 1–43, DOI: 10.1109/TCIAIG.2012.2186810, URL: https://doi.org/10.1109/TCIAIG.2012.2186810.

[33] Francesco Buonamici et al., « Reverse engineering modeling methods and tools: a survey », *in*: *Computer-Aided Design and Applications* 15.*3* (2018), pp. 443–464.

[34] Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo, « A Linda-based platform for the parallel execution of out-place model transformations », *in*: *Information & Software Technology* 79.*C* (Nov. 2016), pp. 17–35, ISSN: 0950-5849, DOI: 10.1016/j.infsof.2016.06.001, URL: https://doi.org/10.1016/j.infsof.2016.06.001.

[35] Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo, « Towards Distributed Model Transformations with LinTra », *in*: *Jornadas de Ingeniería del Software y Bases de Datos* (2016), pp. 1–6, URL: http://hdl.handle.net/10630/12091.

[36] Loli Burgueño et al., « Parallel In-place Model Transformations with LinTra », *in*: *Proceedings of the 3rd Workshop on Scalable Model Driven Engineering part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 23, 2015*. Ed. by Dimitris S. Kolovos et al., vol. 1406, CEUR Workshop Proceedings, CEUR-WS.org, 2015, pp. 52–62, URL: http://ceur-ws.org/Vol-1406/paper6.pdf.

[37] Brendan Burns et al., *Kubernetes: up and running*, O'Reilly Media, Inc., 2022.

[38] Frank Buschmann et al., *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*, Wiley Publishing, 1996, pp. 71–95, ISBN: 0471958697-9780471958697, URL: https://ff.tu-sofia.bg/~bogi/knigi/SE/Wiley%20-%20Pattern-Oriented%20Software%20Architecture%20-%20Volume%201,%20A%20System%20of%20Patterns.pdf.

[39] Fabian Büttner, Marina Egea, and Jordi Cabot, « On verifying ATL transformations using 'off-the-shelf' SMT solvers », *in*: *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2012, pp. 432–448.

[40] Fabian Büttner et al., « Verification of ATL transformations using transformation models and model finders », *in*: *International Conference on Formal Engineering Methods*, Springer, 2012, pp. 198–213.

[41] Jordi Cabot and Ernest Teniente, « Incremental integrity checking of UML/OCL conceptual schemas », *in*: *Journal of Systems and Software* 82.9 (2009), pp. 1459–1478, DOI: 10.1016/j.jss.2009.03.009, URL: https://doi.org/10.1016/j.jss.2009.03.009.

[42] Daniel Calegari et al., « A Type-Theoretic Framework for Certified Model Transformations », English, *in*: *13th Brazilian Symposium on Formal Methods*, Natal, Brazil: Springer, 2011, pp. 112–127, ISBN: 978-3-642-19828-1, DOI: 10.1007/978-3-642-19829-8_8.

[43] Dominique Cansell and Dominique Mery, « Foundations of the B Method. », *in*: *Computers and Informatics* 22 (Jan. 2003), 31 p.

[44] Nicholas Carriero and David Gelernter, « Linda in Context », *in*: *Commun. ACM* 32.*4* (1989), pp. 444–458, ISSN: 0001-0782, DOI: 10.1145/63334.63337, URL: https://doi.org/10.1145/63334.63337.

[45] Maverick Chardet et al., « Madeus: A formal deployment model », *in*: *4PAD 2018 - 5th International Symposium on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018)*, Orléans, France, July 2018, pp. 1–8, URL: https://hal.inria.fr/hal-01858150.

[46] Zheng Cheng, Rosemary Monahan, and James F Power, « A sound execution semantics for ATL via translation validation », *in*: *International Conference on Theory and Practice of Model Transformations*, Springer, 2015, pp. 133–148.

[47] Zheng Cheng, Rosemary Monahan, and James F Power, « Formalised EMFTVM bytecode language for sound verification of model transformations », *in*: *Software & Systems Modeling* 17.*4* (2018), pp. 1197–1225.

[48] Zheng Cheng and Massimo Tisi, « Deep specification and proof preservation for the CoqTL transformation language », *in*: *Software and Systems Modeling* (2022), pp. 1–22.

[49] Zheng Cheng, Massimo Tisi, and Joachim Hotonnier, « Certifying a Rule-Based Model Transformation Engine for Proof Preservation », *in*: *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems*, Montreal, Canada, Oct. 2020, DOI: 10.1145/3365438.3410949, URL: https://hal.inria.fr/hal-02907622.

[50] Avery Ching et al., « One Trillion Edges: Graph Processing at Facebook-scale », *in*: *Proc. VLDB Endow.* 8.*12* (Aug. 2015), pp. 1804–1815, ISSN: 2150-8097, DOI: 10.14778/2824032.2824077, URL: http://dx.doi.org/10.14778/2824032.2824077.

[51] CA Coello Coello, « Evolutionary multi-objective optimization: a historical view of the field », *in*: *IEEE computational intelligence magazine* 1.*1* (2006), pp. 28–36.

[52] Murray Cole, « Algorithmic skeletons : a structured approach to the management of parallel computation », PhD thesis, University of Edinburgh, UK, 1988, URL: http://hdl.handle.net/1842/11997.

[53] Hélène Coullon, Julien Bigot, and Christian Pérez, « Extensibility and Composability of a Multi-Stencil Domain Specific Framework », *in*: *International Journal of Parallel Programming* (Nov. 2017), DOI: 10.1007/s10766-017-0539-5, URL: https://hal.archives-ouvertes.fr/hal-01650998.

[54] Hélène Coullon, Claude Jard, and Didier Lime, « Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning », *in*: *IFM 2019 - 15th International Conference on integrated Formal Methods*, Integrated Formal Methods, Bergen, Norway, Dec. 2019, pp. 120–137, URL: https://hal.archives-ouvertes.fr/hal-02323641.

[55] Hélène Coullon and Sébastien Limet, « The SIPSim implicit parallelism model and the SkelGIS library », *in*: *Concurrency and Computation: Practice and Experience* 28.*7* (2016), pp. 2120–2144, DOI: 10.1002/cpe.3494, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3494.

[56] Jesús Sánchez Cuadrado et al., « Efficient execution of ATL model transformations using static analysis and parallelism », *in*: *IEEE Transactions on Software Engineering* (2020).

[57] Pascal Cuoq et al., « Frama-c », *in*: *International conference on software engineering and formal methods*, Springer, 2012, pp. 233–247.

[58] K. Czarnecki and S. Helsen, « Feature-based survey of model transformation approaches », *in*: *IBM Systems Journal* 45.*3* (2006), pp. 621–645, DOI: 10.1147/sj.453.0621.

[59] Krzysztof Czarnecki and Simon Helsen, « Classification of model transformation approaches », *in*: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, 3, USA, 2003, pp. 1–17.

[60] Georges Da Costa et al., « Energy-and heat-aware HPC benchmarks », *in*: *2013 International Conference on Cloud and Green Computing*, IEEE, 2013, pp. 435–442.

[61] Gwendal Daniel, « NeoEMF: a multi NoSQL Persistence Framework for Very Large Models », *in*: ().

[62] Gwendal Daniel et al., « Improving Memory Efficiency for Processing Large-Scale Models », *in*: vol. 1206, July 2014.

[63] Gwendal Daniel et al., « NeoEMF: A multi-database model persistence framework for very large models », *in*: *Science of Computer Programming* 149 (2017), pp. 9–14.

[64] Matthias Daum, « Reasoning on data-parallel programs in Isabelle/Hol », *in*: *C/C++ Verification Workshop*, vol. 63, Citeseer, 2007.

[65] Jeffrey Dean and Sanjay Ghemawat, « MapReduce: Simplified Data Processing on Large Clusters », *in*: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, San Francisco, CA: USENIX Association, 2004, pp. 137–149, URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[66] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz, « Efficient Big Data Processing in Hadoop MapReduce », *in*: *Proc. VLDB Endow.* 5.*12* (Aug. 2012), pp. 2014–2015, ISSN: 2150-8097, DOI: 10.14778/2367502.2367562, URL: https://doi.org/10.14778/2367502.2367562.

[67] Youssef El Bakouny and Dani Mezher, « Scallina: Translating Verified Programs from Coq to Scala », *in*: *Programming Languages and Systems*, ed. by Sukyoung Ryu, Cham: Springer International Publishing, 2018, pp. 131–145, ISBN: 978-3-030-02768-1.

[68] Jean-marie Favre, « Megamodeling and etymology - a story of words: From MED to MDE via MODEL in five milleniums », *in*: *In Dagstuhl Seminar on Transformation Techniques in Software Engineering, number 05161 in DROPS 04101. IFBI*, 2005.

[69] Péter Fehér et al., « A MapReduce-based approach for finding inexact patterns in large graphs », *in*: *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, IEEE, 2015, pp. 205–212.

[70] Ayat Fekry et al., « To Tune or Not to Tune? In Search of Optimal Configurations for Data Analytics », *in*: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 2494–2504, ISBN: 9781450379984, DOI: 10.1145/3394486.3403299, URL: https://doi.org/10.1145/3394486.3403299.

[71] Maribel Fernández and Jeffrey Terrell, « Assembling the Proofs of Ordered Model Transformations », *in*: *10th International Workshop on Formal Engineering approaches to Software Components and Architectures*, Rome, Italy: EPTCS, 2013, pp. 63–77, DOI: 10.4204/EPTCS.108.5.

[72] Robert W Floyd, « Assigning meanings to programs », *in*: *Program Verification*, Springer, 1993, pp. 65–81.

[73] Michael J. Flynn, « Some Computer Organizations and Their Effectiveness », *in*: *IEEE Transactions on Computers* C-21.*9* (1972), pp. 948–960, DOI: 10.1109/TC.1972.5009071.

[74] Charles L. Forgy, « Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem », *in*: *Artificial Intelligence* 19.*1* (1982), pp. 17–37, DOI: 10.1016/0004-3702(82)90020-0.

[75] Michael Frampton, « Apache mesos », *in*: *Complete Guide to Open Source Big Data Stack*, Springer, 2018, pp. 97–137.

[76] Antonio García-Domínguez, Georg Hinkel, and Filip Krikava, eds., *Proceedings of the 11th Transformation Tool Contest, co-located with the 2018 Software Technologies: Applications and Foundations, TTC@STAF 2018, Toulouse, France, June 29, 2018*, vol. 2310, CEUR Workshop Proceedings, CEUR-WS.org, 2019, URL: http://ceur-ws.org/Vol-2310.

[77] Marzieh Ghorbani, Mohammadreza Sharbaf, Bahman Zamani, et al., « Incremental Model Transformation with Epsilon in Model-Driven Engineering », *in*: *Acta Informatica Pragensia* 11.*2* (2022), pp. 179–204.

[78] Jeremy Gibbons, « Calculating functional programs », *in*: *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Springer, 2002, pp. 151–203.

[79] Jeremy Gibbons, « The School of Squiggol: A History of the Bird-Meertens Formalism. In Formal Methods », *in*: *FM 2019 International Workshops: Porto, Portugal, October 7-1, 2019, Revised Selected Papers, Part II*, 2019, pp. 35–53, DOI: https://doi.org/10.1007/978-3-030-54997-8_2.

[80] Joseph E. Gonzalez et al., « GraphX: Graph Processing in a Distributed Dataflow Framework », *in*: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, ed. by Jason Flinn and Hank Levy, USENIX Association, 2014, pp. 599–613, URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez.

[81] Thomas Grégoire and Adam Chlipala, « Mostly automated formal verification of loop dependencies with applications to distributed stencil algorithms », *in*: *Journal of Automated Reasoning* 62.*2* (2019), pp. 193–213.

[82] Raffaela Groner et al., « A Survey on the Relevance of the Performance of Model Transformations », *in*: *Software Engineering 2022*, ed. by Lars Grunske, Janet Siegmund, and Andreas Vogelsang, Bonn: Gesellschaft für Informatik e.V., 2022, pp. 35–36, DOI: 10.18420/se2022-ws-008.

[83] Raffaela Groner et al., « An Exploratory Study on Performance Engineering in Model Transformations », *in*: *Software Engineering 2021*, ed. by Anne Koziolek, Ina Schaefer, and Christoph Seidl, Bonn: Gesellschaft für Informatik e.V., 2021, pp. 51–52, DOI: 10.18420/SE2021_14.

[84] Object Management Group, *Object Constraint Language, OCL*, Version 2.4, 2015, URL: https://www.omg.org/spec/OCL/2.4/PDF.

[85] D. Harel and A. Pnueli, « On the Development of Reactive Systems », *in*: *Logics and Models of Concurrent Systems*, Berlin, Heidelberg: Springer-Verlag, 1989, pp. 477–498, ISBN: 0387151818.

[86] Ábel Hegedüs et al., « Ecore to Genmodel case study solution using the Viatra2 framework », *in*: *Transformation Tool Contest 2010 1-2 July 2010, Malaga, Spain* (2010), p. 187.

[87] Christopher Henard et al., « Combining multi-objective search and constraint solving for configuring large software product lines », *in*: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, IEEE, 2015, pp. 517–528.

[88] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu, « A Survey on Automatic Parameter Tuning for Big Data Processing Systems », *in*: *ACM Comput. Surv.* 53.*2* (Apr. 2020), ISSN: 0360-0300, DOI: 10.1145/3381027, URL: https://doi.org/10.1145/3381027.

[89] Soichiro Hidaka et al., « Feature-based classification of bidirectional transformation approaches », *in*: *Software & Systems Modeling* 15.*3* (2016), pp. 907–928.

[90] Robert M Hierons et al., « SIP: Optimal product selection from feature models using many-objective evolutionary optimization », *in*: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25.*2* (2016), pp. 1–39.

[91] Charles Antony Richard Hoare, « An axiomatic basis for computer programming », *in*: *Communications of the ACM* 12.*10* (1969), pp. 576–580.

[92] Tassilo Horn, Christian Krause, and Matthias Tichy, « The TTC 2014 Movie Database Case. », *in*: *TTC@ STAF*, Citeseer, 2014, pp. 93–97.

[93] Akos Horváth et al., « Experimental assessment of combining pattern matching strategies with VIATRA2 », *in*: *International Journal on Software Tools for Technology Transfer* 12.*3* (2010), pp. 211–230.

[94] Gábor Imre and Gergely Mezei, « Parallel Graph Transformations on Multicore Systems », *in*: *Multicore Software Engineering, Performance, and Tools*, ed. by Victor Pankratius and Michael Philippsen, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–89, ISBN: 978-3-642-31202-1.

[95] Javier Luis Cánovas Izquierdo et al., « API2MoL: Automating the building of bridges between APIs and Model-Driven Engineering », *in*: *Information and Software Technology* 54.*3* (2012), pp. 257–273.

[96] Benedek Izsó et al., « MONDO-SAM: A framework to systematically assess MDE scalability », *in*: *CEUR Workshop Proceedings* 1206 (Jan. 2014), pp. 40–43.

[97] Bart Jacobs, « The Essence of Coq as a Formal System », *in*: *Online manuscript* (2013).

[98] Frédéric Jouault and Ivan Kurtev, « Transforming models with ATL », *in*: *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2005, pp. 128–138.

[99] Frédéric Jouault et al., « ATL: A model transformation tool », *in*: *Sci. Comput. Program.* 72.*1-2* (2008), Special Issue on Second issue of experimental software and toolkits (EST), pp. 31–39, ISSN: 0167-6423, DOI: 10.1016/j.scico.2007.08.002, URL: https://doi.org/10.1016/j.scico.2007.08.002.

[100] Stefan Jurack and Gabriele Taentzer, « A Component Concept for Typed Graphs with Inheritance and Containment Structures », *in*: *Graph Transformations - 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - - October 2, 2010. Proceedings*, ed. by Hartmut Ehrig et al., vol. 6372, Lecture Notes in Computer Science, Springer, 2010, pp. 187–202, DOI: 10.1007/978-3-642-15928-2_13, URL: https://doi.org/10.1007/978-3-642-15928-2%5C_13.

[101] Nafiseh Kahani and James R Cordy, « Comparison and evaluation of model transformation tools », *in*: *Queen's University, Kingston, Tech. Rep.* (2015).

[102] Nafiseh Kahani et al., « Survey and classification of model transformation tools », *in*: *Software & Systems Modeling* 18.*4* (2019), pp. 2361–2397.

[103] Nafiseh Kahani et al., « Survey and classification of model transformation tools », *in*: *Software & Systems Modeling* 18.*4* (2019), pp. 2361–2397.

[104] Stuart Kent, « Model driven engineering », *in*: *International conference on integrated formal methods*, Springer, 2002, pp. 286–298.

[105] Dimitris S Kolovos, Richard F Paige, and Fiona AC Polack, « Scalability: The holy grail of model driven engineering », *in*: *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, 2008, pp. 10–14.

[106] Dimitris S. Kolovos, Richard F. Paige, and Fiona A. C. Polack, « The Epsilon Object Language (EOL) », *in*: *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA'06, Bilbao, Spain: Springer-Verlag, 2006, pp. 128–142, ISBN: 3540359095, DOI: 10.1007/11787044_11.

[107] Christian Krause, Matthias Tichy, and Holger Giese, « Implementing Graph Transformations in the Bulk Synchronous Parallel Model », *in*: *Fundamental Approaches to Software Engineering*, ed. by Stefania Gnesi and Arend Rensink, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 325–339, ISBN: 978-3-642-54804-8, DOI: 10.1007/978-3-642-54804-8_23.

[108] Rohit Kumar, Alberto Abelló, and Toon Calders, « Cost Model for Pregel on GraphX », *in*: *Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings*, ed. by Marite Kirikova and George A. Norvaag Kjetiland Papadopoulos, vol. 10509, Lecture Notes in Computer Science, Springer, 2017, pp. 153–166, DOI: 10.1007/978-3-319-66917-5_11, URL: https://doi.org/10.1007/978-3-319-66917-5%5C_11.

[109] Palden Lama and Xiaobo Zhou, « AROMA: automated resource allocation and configuration of MapReduce environment in the cloud », *in*: *9th International Conference on Autonomic Computing, ICAC'12, San Jose, CA, USA, September 16 - 20, 2012*, ed. by Dejan S. Milojicic, Dongyan Xu, and Vanish Talwar, ACM, 2012, pp. 63–72, ISBN: 978-1-4503-1520-3, DOI: 10.1145/2371536.2371547, URL: https://doi.org/10.1145/2371536.2371547.

[110] Ralf Lammel, « Google's MapReduce programming model - Revisited », *in*: *Science of Computer Programming* 70.*1* (2008), pp. 1–30, ISSN: 0167-6423, DOI: https://doi.org/10.1016/j.scico.2007.07.001, URL: http://www.sciencedirect.com/science/article/pii/S0167642307001281.

[111] Kevin Lano, T. Clark, and S. Kolahdouz-Rahimi, « A framework for model transformation verification », English, *in*: *Formal Aspects of Computing* 27.*1* (2014), pp. 193–235, ISSN: 0934-5043, DOI: 10.1007/s00165-014-0313-z.

[112] Théo Le Calvar et al., « Efficient ATL Incremental Transformations », *in*: *Journal of Object Technology* 18.*3* (July 2019), The 12th International Conference on Model Transformations, 2:1–17, ISSN: 1660-1769, DOI: 10.5381/jot.2019.18.3.a2, URL: https://doi.org/10.5381/jot.2019.18.3.a2.

[113] Théo Le Calvar et al., « Transformation de modèles et programmation par contraintes avec ATLC », *in*: June 2019, URL: https://doi.org/10.1145/1402947.1402953.

[114] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson, « Calculating parallel programs in Coq using list homomorphisms », *in*: *International Journal of Parallel Programming* 45.*2* (2017), pp. 300–319.

[115] Frédéric Loulergue, Frédéric Gava, and David Billiet, « Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction », *in*: *Computational Science – ICCS 2005*, ed. by Vaidy S. Sunderam et al., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1046–1054, ISBN: 978-3-540-32114-9.

[116] Frédéric Loulergue and Jolan Philippe, « Automatic Optimization of Python Skeletal Parallel Programs », *in*: *19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, Melbourne, Australia, Dec. 2019, DOI: 10.1007/978-3-030-38991-8_13, URL: https://hal.archives-ouvertes.fr/hal-02317123.

[117] Sina Madani, Dimitrios S Kolovos, and Richard F Paige, « Parallel model validation with epsilon », *in*: *European Conference on Modelling Foundations and Applications*, Springer, 2018, pp. 115–131.

[118] Sina Madani, Dimitris Kolovos, and Richard Paige, « Distributed model validation with Epsilon », *in*: vol. 20, Oct. 2021, pp. 1689–1712, DOI: 10.1007/s10270-021-00878-x.

[119] Sina Madani, Dimitris S. Kolovos, and Richard F. Paige, « Towards Optimisation of Model Queries: A Parallel Execution Approach », *in*: *Journal of Object Technology* 18.*2* (July 2019), ed. by Benoit Combemale and Ali Shaukat, The 15th European Conference on Modelling Foundations and Applications, 3:1–21, ISSN: 1660-1769, DOI: 10.5381/jot.2019.18.2.a3, URL: http://www.jot.fm/contents/issue_2019_02/article3.html.

[120] Matthias Maiterth et al., « Power aware high performance computing: Challenges and opportunities for application and system developers—Survey & tutorial », *in*: *2017 In-*

ternational Conference on High Performance Computing & Simulation (HPCS), IEEE, 2017, pp. 3–10.

[121] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky, « Trace-based verification of imperative programs with I/O », *in*: *Journal of Symbolic Computation* 46.*2* (2011), pp. 95–118.

[122] Grzegorz Malewicz et al., « Pregel: A System for Large-scale Graph Processing », *in*: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146, ISBN: 978-1-4503-0032-2, DOI: 10.1145/1807167.1807184, URL: http://doi.acm.org/10.1145/1807167.1807184.

[123] Hugo Martin et al., « Machine Learning and Configurable Systems: A Gentle Introduction », *in*: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, Paris, France: Association for Computing Machinery, 2019, pp. 325–326, ISBN: 9781450371384, DOI: 10.1145/3336294.3342383, URL: https://doi.org/10.1145/3336294.3342383.

[124] Salvador Martínez Perez, Massimo Tisi, and Rémi Douence, « Reactive model transformation with ATL », *in*: *Sci. Comput. Program.* 136 (2017), pp. 1–16, ISSN: 0167-6423, DOI: 10.1016/j.scico.2016.08.006, URL: https://doi.org/10.1016/j.scico.2016.08.006.

[125] Robert Ryan McCune, Tim Weninger, and Greg Madey, « Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing », *in*: *ACM Computing Surveys (CSUR)* 48.*2* (2015), pp. 1–39.

[126] Lambert Meertens, « Algorithmics : towards programming as a mathematical activity », *in*: *In Proceedings of CWI Symposium on Mathematics and Computer Science*, 1986, pp. 289–334.

[127] Gergely Mezei et al., « Towards truly parallel model transformations : A distributed pattern matching approach », *in*: May 2009, pp. 403–410, DOI: 10.1109/EURCON.2009.5167663.

[128] Martin Monperrus, Antoine Beugnard, and Joël Champeau, « A Definition of "Abstraction Level" for Metamodels », *in*: *7th IEEE Workshop on Model-Based Development for Computer Based systems*, update for BASE on Sep 20 2018, San Francisco, United States, 2009, DOI: 10.1109/ECBS.2009.41.

[129] Pierre-Alain Muller et al., « Modeling modeling modeling », *in*: *Software & Systems Modeling* 11.*3* (2012), pp. 347–359.

[130] Bradford Nichols et al., *Pthreads programming: A POSIX standard for better multiprocessing*, O'Reilly Media, Inc., 1996.

[131] Hanne Riis Nielson and Flemming Nielson, *Semantics with applications: an appetizer*, Springer Science & Business Media, 2007.

[132] Bentley James Oakes et al., « Fully verifying transformation contracts for declarative ATL », *in*: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2015, pp. 256–265.

[133] Martin Odersky, Lex Spoon, and Bill Venners, *Programming in scala*, Artima Inc, 2008.

[134] Jeho Oh et al., « Finding Near-Optimal Configurations in Product Lines by Random Sampling », *in*: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 61–71, ISBN: 9781450351058, DOI: 10.1145/3106237.3106273, URL: https://doi.org/10.1145/3106237.3106273.

[135] Leslie Pérez Cáceres et al., « Automatic configuration of GCC using irace », *in*: *International Conference on Artificial Evolution (Evolution Artificielle)*, Springer, 2017, pp. 202–216.

[136] Jolan Philippe, « systematic development of efficient programs on parallel data structures », PhD thesis, Northern Arizona University, 2019.

[137] Jolan Philippe and Frédéric Loulergue, « PySke: Algorithmic Skeletons for Python », *in*: *The 2019 International Conference on High Performance Computing & Simulation (HPCS)*, Dublin, Ireland, July 2019, URL: https://hal.archives-ouvertes.fr/hal-02317127.

[138] Jolan Philippe et al., « Executing Certified Model Transformations on Apache Spark », *in*: SLE 2021, Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 36–48, ISBN: 9781450391115, DOI: 10.1145/3486608.3486901, URL: https://doi.org/10.1145/3486608.3486901.

[139] Jolan Philippe et al., « Towards Transparent Combination of Model Management Execution Strategies for Low-Code Development Platforms », *in*: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, Virtual Event, Canada: Association for Computing Machinery, 2020, ISBN: 9781450381352, DOI: 10.1145/3417990.3420206, URL: https://doi.org/10.1145/3417990.3420206.

[140] Iman Poernomo and Jeffrey Terrell, « Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq », English, *in*: *12th International Conference on Formal Engineering Methods*, Shanghai, China: Springer, 2010, pp. 56–73, ISBN: 978-3-642-16900-7, DOI: 10.1007/978-3-642-16901-4_6.

[141] Risto Pohjonen and Juha-pekka Tolvanen, *Automated Production of Family Members: Lessons learned*, ed. by K. Schmid and B. Geppert, 2002.

[142] B Randell and JN Buxton, « Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969 », *in*: (1970).

[143] Louis M Rose et al., « A feature model for model-to-text transformation languages », *in*: *2012 4th International Workshop on Modeling in Software Engineering (MISE)*, IEEE, 2012, pp. 57–63.

[144] Margaret Rouse, *Task, Definition*, https://whatis.techtarget.com/definition/task, Accessed: 2020-07-14.

[145] J. Rumbaugh, I. Jacobson, and G. Booch., *The unified modeling language*, Reference manual, 1999.

[146] Ágnes Salánki et al., « Qualitative Characterization of Quality of Service Interference between Virtual Machines. », *in*: *ARCS Workshops*, 2011.

[147] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar, « On the value of user preferences in search-based software engineering: A case study in software product lines », *in*: *2013 35Th international conference on software engineering (ICSE)*, IEEE, 2013, pp. 492–501.

[148] E Schikuta, « MPI: A message-passing interface standard », *in*: *Techn. Ber., University of Tennessee, Knoxville, Tennesee* 30 (1994).

[149] Douglas C Schmidt, « Model-driven engineering », *in*: *Computer-IEEE Computer Society*-39.*2* (2006), p. 25.

[150] Patrick Schnider, « An introduction to proof assistants », *in*: *Student Seminar in Combinatorics: Mathematical Software*, vol. 8, 2009.

[151] Syed Asif Raza Shah et al., « Benchmarking and Performance Evaluations on Various Configurations of Virtual Machine and Containers for Cloud-Based Scientific Workloads », *in*: *Applied Sciences* 11.*3* (2021), ISSN: 2076-3417, DOI: 10.3390/app11030993, URL: https://www.mdpi.com/2076-3417/11/3/993.

[152] Allan Snavely et al., « Benchmarks for Grid Computing: A Review of Ongoing Efforts and Future Directions », *in*: *SIGMETRICS Perform. Eval. Rev.* 30.*4* (Mar. 2003), pp. 27–32, ISSN: 0163-5999, DOI: 10.1145/773056.773062, URL: https://doi.org/10.1145/773056.773062.

[153] Dave Steinberg et al., *EMF: eclipse modeling framework*, Pearson Education, 2008.

[154] Kurt Stenzel, Nina Moebius, and Wolfgang Reif, « Formal verification of QVT transformations for code generation », *in*: *Software & Systems Modeling* 14 (2015), pp. 981–1002.

[155] L.M. Surhone, M.T. Timpledon, and S.F. Marseken, *Von Neumann Architecture: Central Processing Unit, John Von Neumann, Universal Turing Machine, SISD, Read-write Memory*, Betascript Publishing, 2010, ISBN: 9786130318154, URL: https://books.google.fr/books?id=vTlOQwAACAAJ.

[156] Gabriel Tamura and Anthony Cleve, « A Comparison of Taxonomies for Model Transformation Languages », *in*: *Paradigma* 4.*1* (Mar. 2010), pp. 1–14, URL: https://hal.inria.fr/inria-00488765.

[157] OMG team, *Meta Object Facility (MOF) Specification*, Object Management Group, 2000, URL: https://www.omg.org/mof/.

[158] The Coq development team, *The Coq proof assistant reference manual*, Version 8.16, LogiCal Project, 2004, URL: http://coq.inria.fr.

[159] Julien Tesson et al., « Program calculation in Coq », *in*: *International Conference on Algebraic Methodology and Software Technology*, Springer, 2010, pp. 163–179.

[160] Ashish Thusoo et al., « Data warehousing and analytics infrastructure at Facebook », *in*: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 1013–1020.

[161] Ashish Thusoo et al., « Hive-a petabyte scale data warehouse using hadoop », *in*: *2010 IEEE 26th international conference on data engineering (ICDE 2010)*, IEEE, 2010, pp. 996–1005.

[162] Ashish Thusoo et al., « Hive: A Warehousing Solution over a Map-Reduce Framework », *in*: *Proc. VLDB Endow.* 2.*2* (Aug. 2009), pp. 1626–1629, ISSN: 2150-8097, DOI: 10.14778/1687553.1687609, URL: https://doi.org/10.14778/1687553.1687609.

[163] Massimo Tisi and Zheng Cheng, « CoqTL: an Internal DSL for Model Transformation in Coq », *in*: *ICMT 2018 - 11th International Conference on Theory and Practice of Model Transformations*, vol. 10888, LNCS, Toulouse, France: Springer, June 2018, pp. 142–156, DOI: 10.1007/978-3-319-93317-7_7, URL: https://hal.inria.fr/hal-01828344/file/main.pdf.

[164] Massimo Tisi, Rémi Douence, and Dennis Wagelaar, « Lazy Evaluation for OCL », *in*: *Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015*, ed. by Achim D. Brucker et al., vol. 1512, CEUR Workshop Proceedings, CEUR-WS.org, 2015, pp. 46–61, URL: http://ceur-ws.org/Vol-1512/paper04.pdf.

[165] Massimo Tisi, Martínez Salvador Perez, and Hassene Choura, « Parallel Execution of ATL Transformation Rules », *in*: *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, ed. by Ana Moreira et al., vol. 8107, Lecture Notes in Computer Science, Springer, 2013, pp. 656–672, DOI: 10.1007/978-3-642-41533-3_40, URL: https://doi.org/10.1007/978-3-642-41533-3%5C_40.

[166] Massimo Tisi et al., « Lazy Execution of Model-to-Model Transformations », *in*: *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, ed. by Jon Whittle, Tony Clark, and Thomas Kühne, vol. 6981, Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 32–46, ISBN: 978-3-642-24485-8, DOI: 10.1007/978-3-642-24485-8_4, URL: https://doi.org/10.1007/978-3-642-24485-8%5C_4.

[167] *Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms*, CEUR Workshop Proceedings (CEUR-WS.org), Eindhoven, Netherlands, July 2019, URL: https://hal.archives-ouvertes.fr/hal-02363416.

[168] Le-Duc Tung and Zhenjiang Hu, « Towards Systematic Parallelization of Graph Transformations Over Pregel », *in*: *Int. J. Parallel Program.* 45.*2* (Apr. 2017), pp. 320–339, ISSN: 0885-7458, DOI: 10.1007/s10766-016-0418-5, URL: https://doi.org/10.1007/s10766-016-0418-5.

[169] Tamás Vajk et al., « Runtime Model Validation with Parallel Object Constraint Language », *in*: *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, Wellington, New Zealand: Association for Computing Machinery, 2011, ISBN: 9781450309141, DOI: 10.1145/2095654.2095663, URL: https://doi.org/10.1145/2095654.2095663.

[170] Leslie G. Valiant, « A Bridging Model for Parallel Computation », *in*: *Commun. ACM* 33.*8* (Aug. 1990), pp. 103–111, ISSN: 0001-0782, DOI: 10.1145/79173.79181, URL: http://doi.acm.org/10.1145/79173.79181.

[171] Gergely Varró and Frederik Deckwerth, « A Rete Network Construction Algorithm for Incremental Pattern Matching », *in*: *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, ed. by Keith Duddy and Gerti Kappel, vol. 7909, Lecture Notes in Computer Science, Springer, 2013, pp. 125–140, DOI: 10.1007/978-3-642-38883-5_13, URL: https://doi.org/10.1007/978-3-642-38883-5%5C_13.

[172] Vinod Kumar Vavilapalli et al., « Apache hadoop yarn: Yet another resource negotiator », *in*: *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[173] Dennis Wagelaar et al., « Towards a General Composition Semantics for Rule-Based Model Transformation », *in*: *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, Wellington, New Zealand: Springer-Verlag, 2011, pp. 623–637, ISBN: 9783642244841.

[174] Edward D. Willink, « Deterministic Lazy Mutable OCL Collections », *in*: *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers*, ed. by Martina Seidl and Steffen Zschaler, vol. 10748, Lecture Notes in Computer Science, Springer, 2017, pp. 340–355, DOI: 10.1007/978-3-319-74730-9_30, URL: https://doi.org/10.1007/978-3-319-74730-9%5C_30.

[175] Dili Wu and Aniruddha Gokhale, « A self-tuning system based on application Profiling and Performance Analysis for optimizing Hadoop MapReduce cluster configuration », *in*: *20th Annual International Conference on High Performance Computing*, 2013, pp. 89–98, DOI: 10.1109/HiPC.2013.6799133.

[176] Matei Zaharia et al., « Spark: Cluster computing with working sets », *in*: *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[177] Jingren Zhou et al., « SCOPE: Parallel Databases Meet MapReduce », *in*: *The VLDB Journal* 21.5 (Oct. 2012), pp. 611–636, ISSN: 1066-8888, DOI: 10.1007/s00778-012-0280-z, URL: https://doi.org/10.1007/s00778-012-0280-z.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

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

Titre : Contribution à l'analyse de l'espace de conception d'un moteur de transformation distribué

Mot clés : Spark,Model Transformation,Model Queries,Correction,Caractérisation

Résumé : L'espace de conception pour définir un moteur de transformation de modèle distribué est un large spectre de possibilités et d'opportunités pour améliorer les performances en termes de temps de calcul et de consommation mémoire. Selon les décisions adoptées, l'utilisation d'un moteur de transformation peut être complètement différente (par exemple, une solution incrémentale pour un modèle souvent modifié contre un moteur formellement spécifié pour le raisonnement, non performant). Les solutions déjà existantes proposent des moteurs avec différents objectifs basés sur plusieurs approches, notamment la distribution, la paresse, l'incrémentalité et l'exactitude. Cependant, comparer les solutions n'est pas anodin, et n'a pas forcément de sens. C'est pourquoi nous avons mis en place un nouveau moteur, intégrant la variabilité, qui permet une analyse de son espace de conception. À partir d'un langage doté de spécifications formelles, nous avons créé SparkTE, un moteur de transformation paramétrable et distribué au-dessus de Spark. Dans cette thèse, nous cherchons à analyser l'impact des choix à différents niveaux : les modèles de programmation utilisés pour définir les expressions ; les différentes sémantiques utilisées pour définir le calcul d'une transformation ; et l'impact des choix d'ingénierie.

Title: Contribution to the Analysis of the Design-Space of a Distributed Transformation Engine

Keywords: Spark,Model Transformation,Model Queries,Correctness,Features

Abstract: The design space for defining a distributed model transformation engine is a large spectrum of possibilities and opportunities to enhance performances in terms of computation time and memory consumption. Depending on the adopted decisions, the use of a transformation engine can be completely different (e.g., an incremental solution for an often-modified model vs a formally specified engine for reasoning, not performing). Already existing solutions propose engines with different goals based on several approaches including distribution, laziness, incrementality, and correctness. However, comparing the solutions is not trivial, and does not necessarily make sense. That is why we have implemented a new engine, integrating variability, that allows an analysis of its design space. From a language that has formal specifications, we created SparkTE, a parametrizable and distributed transformation engine on top of Spark. In this thesis, we aim at analysing the impact of the choices at different levels: the used programming models for defining expressions; the different semantics used to define the computation of a transformation; and the impact of engineering choices.