UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

THESE DE DOCTORAT DE

L'UNIVERSITE DE NANTES

COMUE UNIVERSITE BRETAGNE LOIRE

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

Par

Hugo BRUNELIERE

Generic Model-based Approaches for Software Reverse Engineering and Comprehension

Thèse présentée et soutenue à Nantes le jeudi 20 décembre 2018

Unité de recherche : IMT Atlantique Bretagne-Pays de la Loire, Ecole Mines-Télécom
Laboratoire des Sciences du Numérique de Nantes – LS2N (CNRS, UMR 6004)

Rapporteurs avant soutenance :

Richard Paige Professeur – University of York, York (United Kingdom)
Benoit Baudry Professeur – KTH Royal Institute of Technology, Stockholm (Sweden)

Composition du Jury :

Président :	Jordi Cabot	Professeur – ICREA & Open University of Catalonia, Barcelona (Spain)
Examinateurs :	Richard Paige	Professeur – University of York, York (United Kingdom)
	Benoit Baudry	Professeur – KTH Royal Institute of Technology, Stockholm (Sweden)
	Corinne Miral	Maître de conférences HDR – Université de Nantes, Nantes (France)
Référent :	Gerson Sunye	Maître de conférences HDR – Université de Nantes, Nantes (France)

Note :
Les travaux scientifiques présentés dans ce manuscrit de thèse ont été réalisés sous la supervision du Professeur Jordi Cabot, également membre du jury.

Acknowledgement

Foremost, I would like to express my sincere gratitude to Dr. Gerson Sunye and Prof. Jordi Cabot for their precious advices and feedback when writing down this Ph.D. thesis manuscript that summarizes a significant part of my professional life. This has been quite a constructive experience for me. I also would like to thank Prof. Jordi Cabot for mentoring me during his years in Nantes, and I hope we can continue having more good collaborations in the future.

Besides them, I would like to thank Prof. Richard Paige and Prof. Benoit Baudry for reviewing the present manuscript and providing their insightful comments on this work. In addition, I would like to thank Dr. Corinne Miral for also taking part in my thesis committee.

I thank all the so-called AtlanMod & Friends (and now NaoMod folks) for the stimulating, international and fun working environment in which I have the chance to evolve since more than 12 years. Notably, I would like to give extra thanks to the project engineers that did a great job on the implementation of MoDisco & EMF Views: people from Mia-Software (Frédéric Madiot, Grégoire Dupé, Fabien Giquel and others), Guillaume Doux, Juan David Villa Calle, Jokin García Pérez and Florent Marchand de Kerchove.

More generally, I thank everyone I had the chance to meet, discuss and work with all along these years in the context of various projects or conferences/events, here in Nantes or elsewhere in Europe and the rest of the world. It has been a long enriching journey!

Last but not least, I would like to give very special thanks to Prof. Jean Bézivin for convincing me to work in a research environment in the first place. I am grateful for this great opportunity he offered me back then.

To my wife and little boy

To my parents and younger sister

To my entire family, and to the new Brazilian one

To my other family: friends from my hometown, from here and elsewhere

Résumé Français

Introduction

Avec l'avènement des technologies de l'information dans notre société, les organisations entreprennent de plus en plus de projets de migration et/ou de modernisation logiciel. De nombreuses entreprises, indépendamment de leur taille et secteur d'activité, font désormais face au problème bien connu de devoir gérer, maintenir, faire évoluer ou remplacer leurs systèmes logiciel existants [135]. Une telle situation est principalement due aux fréquentes et rapides modifications dans le paysage technologique. Cependant, les raisons sont généralement multiples: nouveaux besoins utilisateurs, stratégies métiers en évolution, aspects organisationnels en mouvement, législations changeantes, etc. Dans tous les cas, la motivation principale derrière les projets de migration/modernisation est habituellement économique [130].

En raison de cela, les entreprises sont régulièrement poussées à faire évoluer (au moins en partie) leurs systèmes logiciel déjà existants avant que ceux-ci ne deviennent véritablement obsolètes ou ne commencent à dysfonctionner. Ces *systèmes patrimoniaux* sont souvent de grandes applications jouant un rôle critique au sein du système d'information de l'entreprise [15]. De plus, ils ont habituellement un impact non-négligeable sur son fonctionnement au quotidien.

Les projets de migration/modernisation associés ne peuvent donc pas être pris à la légère, puisqu'ils ne viennent jamais sans d'importants challenges associés [122]. Dans le cas idéal, ils ne devraient pas débuter en raison de modes passagères mais plutôt car fortement motivés par de réels limitations technologiques ou en anticipation de problèmes à venir. Cependant, en pratique, les décideurs et ingénieurs ont seulement une vision limitée et partielle de la complexité et des potentielles conséquences de leurs projets logiciel [80]. Ainsi, améliorer la compréhension globale des systèmes logiciel concernés (e.g. en termes d'architecture, de fonctionnalités fournies, de règles métierimposées ou de données manipulées) est un point crucial [208].

L'Ingénierie Dirigée par les Modèles (IDM) [182], souvent plus généralement appelée *Modélisation*, est un paradigme du Génie Logiciel reposant sur la création, la manipulation et l'utilisation intensive de modèles de natures diverses et variées. Elle repose largement sur l'hypothèse de base que *tout est un modèle* [19]. Ces modèles peuvent décrire différents (et possiblement tous les) aspects complémentaires à la fois des systèmes modélisés et des activités d'ingénierie associées. Par conséquent, dans les approches basées sur les modèles, ceux-ci sont considérés comme des entités de première classe au sein des processus de conception, développement, déploiement, maintenance et évolution. La communauté scientifique de l'IDM est soutenue par un écosystème solide et

riche en pratiques, outils et cas d'utilisation provenant à la fois des mondes académique et industriel [24]. Des études relativement récentes sur l'application concrète de l'IDM dans l'industrie [110, 210] ont montré que l'IDM a déjà engendré des bénéfices intéréssants dans le cadre de différents types de projets ou d'activités à forte composante logiciel.

Description de la Problématique

Un projet de migration et/ou modernisation est habituellement constitué de trois phases consécutives distinctes :

1. Le processus complexe d'obtention de représentations utiles et de plus haut niveau d'un système (patrimonial) donné est appelé *rétro-ingénierie* [43]. Son objectif ultime est de fournir une meilleure compréhension du but et de l'état actuel du système concerné.
2. L'ingénierie avancée analyse ensuite ces modèles et les transforment (lorsque nécessaire) en des spécifications de la nouvelle version du système [14].
3. Les développeurs ou des techniques de génération automatique de code (ou une combinaison des deux) utilisent finalement ces modèles pour produire le code correspondant à la plateforme ciblée [107].

Des assemblages de différentes solutions et technologies peuvent aider à la réalisation de quelques unes des tâches nécessaires. Par exemple, c'est le cas d'outils qui produisent automatiquement différents types de modèles à partir de code source logiciel déjà existant. C'est aussi le cas de générateurs de code ciblant divers langages de programmation et plateformes techniques. Cependant, à notre connaissance, il n'existe pas actuellement de méthodologie globale qui permette de guider les décideurs, les architectes logiciel et les ingénieurs tout au long d'un processus de migration/modernisation.

Ceci est particulièrement vrai dans le contexte de la phase de rétro-ingénierie, qui constitue le focus du travail présenté dans cette thèse. Les challenges importants associés [38] incluent notamment :

— Le support générique à des technologies logiciel hétérogènes (e.g. différents langages de programmation).
— L'aptitude à extraire des données utiles à partir du système (e.g. allant plus loin que l'arbre de syntaxe abstraite).
— La couverture à la fois des dimensions statiques et dynamiques du système (e.g. incluant les données d'exécution).
— La capacité à obtenir des vues pertinentes du système et à gérer des liens de tracabilité entre elles.

Ce dernier point est même considéré comme une activité et un objectif fondamental de la rétro-ingénierie et compréhension du logiciel.

Assez récemment (comparé à l'histoire de la rétro-ingénierie), la Rétro-Ingénierie Dirigée par les Modèles (RIDM) [181] a été proposée afin d'améliorer les processus de rétro-ingénierie plus traditionnels. La RIDM peut être définie comme l'application des principes et techniques de l'IDM dans le contexte d'activités de rétro-ingénierie. Les approches RIDM dépendent notamment des techniques basées sur les modèles existantes pour obtenir des modèles représentant les systèmes étudiés en fonction de différents aspects et à différents niveaux d'abstraction. Les diagrammes de classe UML (Unified

Modeling Languages) [167], les machines à états ou les workflows sont des exemples bien-connus de tels modèles (pour en citer seulement quelques uns).

En s'appuyant sur ces différents modèles obtenus, il est ensuite possible d'analyser plus en profondeur et de comprendre le système faisant l'objet d'un processus de rétro-ingénierie. Dans ce but, les résultats attendus incluent la fourniture de vues (basées sur ces modèles) cohérentes et complémentaires décrivant ce système [27]. De telles vues sur des modèles permettent de fédérer l'information pertinente provenant de divers modèles souvent interdépendants, notamment ceux précedemment obtenu par RIDM. Ces vues doivent être spécifiées et calculées en fonction de points de vue qui sont appropriés aux différentes parties prenantes.

Cependant, une limitation importante des approches RIDM existantes ainsi que des approaches de fédération/vue sur des modèles est qu'elles reposent assez souvent sur des intégrations sur-mesure et spécifiques de différents outils [201]. De plus, leurs implémentations techniques peuvent parfois s'avérer (très) hétérogènes et coûteuses. Cela peut compliquer encore davantage la situation, et peut à terme entraver leurs bons déploiements et utilisations pratiques.

Dans cette thèse, nous argumentons que des solutions génériques et extensibles complémentaires sont toujours manquantes (ou tout du moins fournissent un support incomplet) afin d'être en mesure de combiner la RIDM avec des capacités avancées de fédération/vue sur les modèles. Plus particulièrement, de telles solutions doivent permettre d'adresser plusieurs types de scenarios de rétro-ingénierie et compréhension du logiciel, s'appuyant possiblement sur différentes plateformes techniques. De plus, fournir des implémentations en logiciel libre des approches proposées doit permettre de favoriser leur intéropérabilité avec l'existant ainsi que leur dissémination à plus large échelle.

Approche Mise en Oeuvre

L'objectif de cette thèse est donc de progresser vers l'amélioration du support réutilisable pour des cas d'utilisation possiblement hétérogènes de rétro-ingénierie et compréhension du logiciel. Notre proposition est de s'appuyer sur deux approches complémentaires, génériques, extensibles et basées sur les modèles dans le but de faciliter l'élaboration de solutions à ce problème.

Rétro-Ingénierie Dirigée par les Modèles (RIDM). Nous avons commencé par capitaliser à la fois sur les différentes pratiques existantes (provenant de l'état de l'art) et sur nos propres expériences passées (e.g. de collaboration avec des entreprises dans le contexte de vrais projets de modernisation). À partir de cette connaissance scientifique et pratique acquise au fil des ans, nous avons été en mesure de spécifier un processus global ainsi qu'une architecture support pour une approche de RIDM générique et extensible. Nous avons également travaillé sur une implémentation concrète de cette approche conceptuelle sous la forme d'un premier framework en logiciel libre, en partenariat direct avec une entreprise experte du domaine. Initialement, la solution conçue se focalisait principalement sur la rétro-ingénierie des aspects structurels (i.e. statiques) des systèmes logiciel. Nous avons par la suite complété ce framework avec des travaux de recherche supplémentaires sur la rétro-ingénierie d'aspects comportementaux (i.e. dynamiques).

Compréhension via Fédération/Vue sur des Modèles. De manière complémentaire à la RIDM, des capacités de fédération de modèles sont attendues afin d'améliorer les activités de compréhension (basée sur les modèles) du logiciel. Ainsi, nous avons proposé une approche générique (i.e. indépendante des types de modèles concernés) pour la construction et la manipulation transparente de vues sur des modèles. De telles vues peuvent possiblement fédérer divers ensembles de modèles hétérogènes, en fonction de différents points de vue exprimant les aspects à partir desquels le système concerné doit être observé. Basé sur la notion de virtualisation de modèles, nous avons dévelopé une implémentation non-intrusive de cette approche conceptuelle. Il en a résulté un second framework en logiciel libre, qui peut notamment consommer des modèles produits par notre premier framework dédié à la RIDM (parmi d'autres modèles provenant de différentes sources).

S'appuyant toutes les deux sur les mêmes principes et techniques de base de l'IDM, les deux approches conceptuelles proposées peuvent être combinées et chaînées ensemble de manière naturelle. L'intégration pratique des deux frameworks correspondants est rendue possible grâce à l'utilisation d'un environnement de modélisation commun et partagé. Basées sur ces deux approches et frameworks, des solutions plus élaborées de rétro-ingénierie et de compréhension du logiciel peuvent être conçues dans le but d'adresser des scénarios potentiellement complexes. De plus, les approches proposées peuvent également être réutilisées (de manière indépendante ou conjointe) en collaboration avec des approaches basées sur les modèles adressant d'autres problématiques, toujours dans un contexte de Génie Logiciel ou bien associé à d'autres types d'activités.

Contributions

Comme représenté graphiquement dans la Figure 1, les contributions principales de cette thèse sont les suivantes :

- MoDisco propose une approche de RIDM globale, générique et extensible ayant pour objectif de faciliter l'élaboration de solutions de RIDM dans de nombreux contextes différents. Elle est réalisée en pratique par un framework technique étant un projet officiel Eclipse construit au dessus de l'environnement Eclipse et son Eclipse Modeling Framework (EMF). Parmi d'autres possible sorties (e.g. code source généré, rétro-documentation), ce framework permet notamment d'obtenir différents types de modèles à partir de systèmes existants et des divers éléments qui les composent (code source, données, etc.). Les modèles obtenus peuvent ensuite être réutilisés lors des activités de fédération et compréhension, e.g. en utilisant EMF Views tel qu'introduit juste après.
  - fREX propose un framework ouvert et extensible, basé sur l'approche MoDisco , qui est capable de générer automatiquement et d'exécuter des modèles comportementaux exprimés en fUML (Foundational UML Subset). Il fournit également un mapping de base initial entre les diagrammes de classes/activités d'UML (i.e. fUML) et des éléments noyaux du langage de programmation Java.
- EMF Views propose une approche générique et extensible ayant pour objectif de spécifier, construire et manipuler des vues sur des ensembles de modèles existants divers et variés. Ces vues peuvent être pertinentes pour différents types de

parties prenantes (e.g. ingénieurs et architectes logiciel, décideurs). La solution proposée peut notamment consommer les modèles produits précédemment grâce à MoDisco , mais peut également réutiliser des modèles provenant d'autres sources (e.g. définis manuellement par des ingénieurs). Elle repose sur un backend de virtualisation de modèles qui permet de renvoyer au contenu de tels modèles de manière transparente et non-intrusive. Elle est également réalisée en pratique par un framework dédié implémenté au dessus de l'environnement Eclipse/EMF.

— En complément de l'approche EMF Views et de son framework d'implémentation, nous contribuons une étude détaillée de l'état-de-l'art sur les approches de vue sur des modèles. Grâce à ce travail étendu, un feature model est proposé afin d'aider à caractériser les différentes solutions disponibles et à sélectionner la/les plus appropriée(s) dans des contextes donnés.

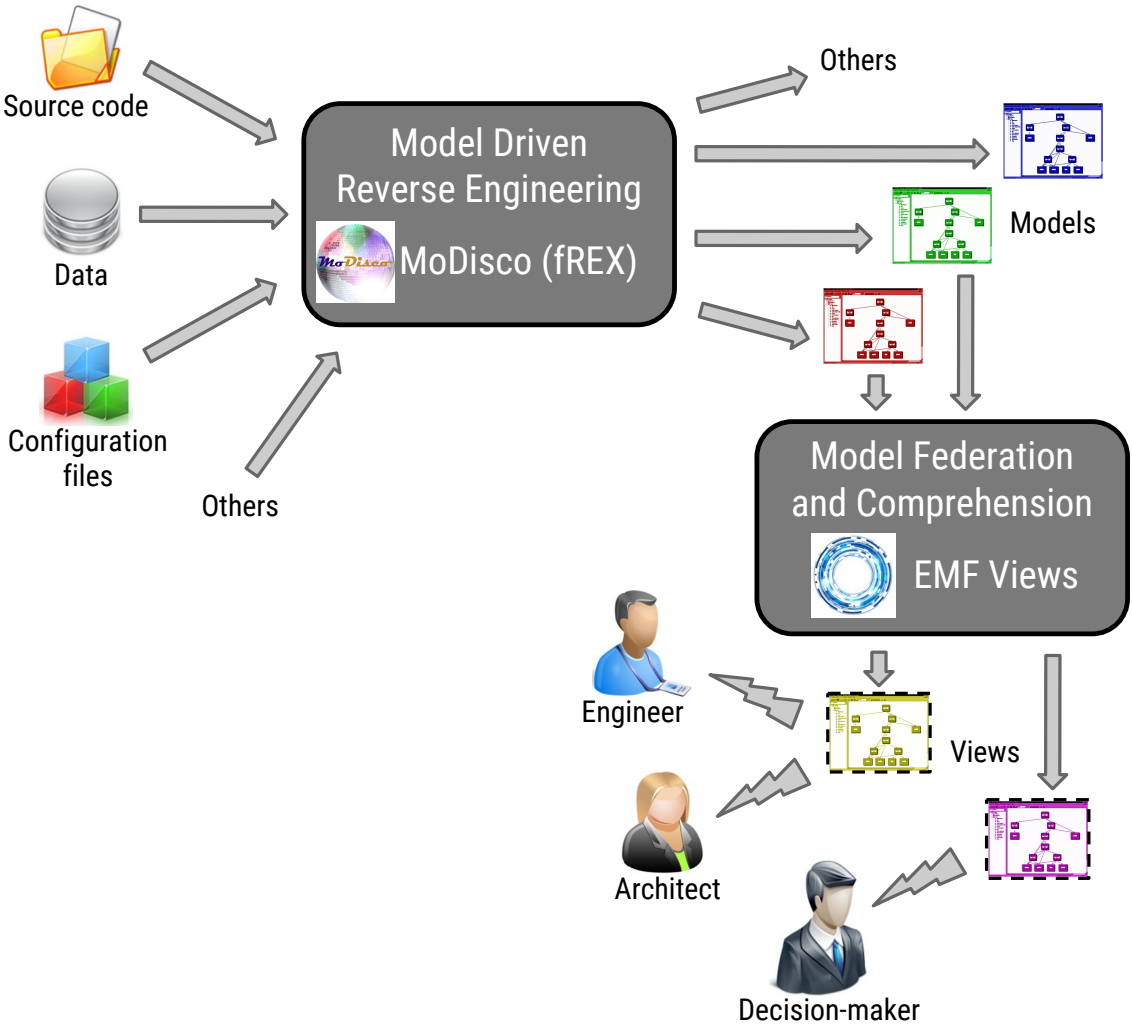

Figure 1 – Un écosystème basé sur les modèles pour la rétro-ingénierie et compréhension des systèmes logiciel.

Outils et Résultats

Les approches et frameworks présentés dans cette thèse sont tous implémentés sous la forme de plugins Eclipse en logiciel libre. L'intégralité de leur code source, leur do-

cumentation ainsi que les différentes autres resources de développement et d'utilisation associées sont disponibles en ligne via des sites ou dépôts dédiés (MoDisco [1], fREX [2], EMF Views [3]). Tous ces frameworks techniques implémentant les approches conceptuelles proposées sont construits au-dessus d'EMF, l'environnement de modélisation standard dans l'écosystème Eclipse, afin de favoriser leur interopérabilité avec d'autres solutions déjà existantes par ailleurs.

Dans cette thèse, nous évaluons notamment le passage à l'échelle de nos solutions dans le contexte de différents cas d'utilisation réels (i.e. provenant directement d'entreprises partenaires) ou réalistes (i.e. simulant des besoins réels remontés par des entreprises partenaires). Par exemple, nous montrons en pratique que nous sommes en mesure de faciliter de manière effective la création de solutions techniques répondant à des besoins de rétro-ingénierie variés (refonte d'applications existantes, qualité du logiciel). Nous montrons aussi comment nous rendons possible la fédération de divers modèles hétérogènes (dont certains provenant de la rétro-ingénierie) dans un contexte industriel nécessitant une solution de traçabilité entre conception et exécution d'un système.

En complément de ces applications concrètes et pratiques, nous réalisons des tests de performance poussés concernant plus spécifiquement certains des composants clés de nos solutions. Notamment, nous évaluons dans le détail les capacités de notre solution de RIDM (i.e. MoDisco) en termes de découverte automatique de modèles à partir de code source Java. Nous testons également la scalabilité de notre solution de fédération/-vue (i.e. EMF Views) sur des modèles très volumineux décrivant des aspects comportementaux (dynamiques) d'un système. Les résultats obtenus dans les deux cas montrent que les solutions proposées sont d'ores-et-déjà déployables et utilisables dans des projets industriels à moyenne échelle. Cependant il reste encore des challenges scientifiques à adresser, notamment en ce qui concerne la découverte automatique de modèles comportementaux ou bien la vérification/validation et la mise-à-jour de vues sur des modèles à grande échelle.

Enfin, l'ensemble du travail de recherche présenté dans cette thèse a été réalisé et appliqué dans le cadre de plusieurs projets collaboratifs nationaux et européens impliquant à la fois des laboratoires académiques et des entreprises. De ces expériences passées et présentes, nous tirons des leçons pouvant intéresser toute personne engagée dans des activités de création, conception, développement et promotion d'approches et d'outils basés sur les modèles. De manière plus générale, certains des facteurs clés de réussite (ou d'échec) que nous identifions peuvent également s'avérer pertinents dans le contexte de n'importe quel projet logiciel.

1. https://www.eclipse.org/MoDisco
2. https://github.com/atlanmod/fREX
3. http://www.atlanmod.org/emfviews

Abstract

Nowadays, companies undertake more and more software migration and/or modernization projects for different reasons (economical, business-related, organizational, technical, legal, etc.). Independently of their size and activity, they all face the problem of managing, maintaining, evolving or replacing their existing software systems. The first phase of any project of this kind is called Reverse Engineering: the complex process of obtaining various representations of an existing system, with the main objective to provide a better comprehension of its purpose and state.

Model Driven Engineering (MDE) / Modeling is a software engineering paradigm relying on intensive model creation, manipulation and use. These models can describe different complementary aspects of the modeled systems and related engineering activities. Thus, in model-based approaches, models are considered as first-class entities in design, development, deployment, integration, maintenance and evolution tasks.

Model Driven Reverse Engineering (MDRE) has been proposed to enhance more traditional reverse engineering processes. MDRE can be defined as the application of MDE in the context of reverse engineering activities, in order to obtain models representing an existing system according to various aspects. It is then possible to further comprehend this system via coherent views federating these different models.

However, existing MDRE and model view/federation solutions are limited as they quite often rely on custom or case-specific integrations of different tools. Moreover, they can sometimes be (very) heterogeneous which may hinder their practical deployments and usages. Generic and extensible solutions are still missing (or providing incomplete support) for MDRE to be combined with advanced model view/federation capabilities.

In this thesis, we propose to rely on two complementary model-based approaches: (i) A generic and extensible approach intending to facilitate the elaboration of MDRE solutions in many different contexts. It notably allows to obtain different kinds of models out of existing systems and the various artifacts composing them (e.g. source code, data). (ii) A generic and extensible approach intending to specify, build and manipulate views federating different existing models (e.g. the ones resulting from our MDRE approach). Such views can be relevant to different kinds of stakeholders (e.g. software engineers/architects, decision-makers) according to comprehension objectives.

The corresponding implementations are open source and rely on the Eclipse-EMF *de-facto* standard modeling framework. They have been designed, developed and evaluated within collaborative projects involving several companies and their realistic use cases.

1

Introduction and Context

1.1 Introduction

With the advent of Information Technology in our society, organizations undertake more and more software migration and/or modernization projects every day. Many companies, independently of their size and activity type, are now facing the well-known problem of managing, maintaining, evolving or replacing their existing software systems [135]. Such a situation is principally due to frequent and rapid modifications in the technological landscape. However, the reasons are generally multiple: new user requirements, evolving business strategies, moving organizational aspects, changing legislation, etc. In any case, the main rationale behind migration/modernization projects is usually economical [130].

Because of that, companies are regularly pushed to evolve (at least part of) their already existing software systems before these systems actually become obsolete or start to malfunction. These *legacy systems* are often large applications playing a critical role in the information system of the company [15]. Moreover, they usually still have a non-negligible impact on its daily operations.

The related migration/modernization projects cannot be taken too lightly, as they never come without significant associated challenges [122]. In the ideal case, they should not start because of passing fads but rather because strongly motivated by real technological limitations or forthcoming system issues. However, in practice, decision-makers and engineers have only a limited and partial vision over the complexity and potential consequences of their software projects [80]. Thus, improving the overall comprehension of the concerned software systems (e.g. in terms of architecture, provided features, enforced rules or handled data) is a crucial point [208].

Model Driven Engineering (MDE) [182], often more generally referred to as *Modeling*, is a software engineering paradigm relying on the intensive creation, manipula-

tion and use of models having various and varied natures. It is largely based on the core assumption that *everything is a model* [19]. These models can describe different (and possibly all) complementary aspects of both the modeled systems and the related engineering activities. As a consequence, in model-based approaches, models are considered as first-class entities within design, development, deployment, integration, maintenance and evolution processes. The MDE scientific community is backed by a solid ecosystem of existing practices, tools and use cases coming both from academics and industrials [24]. Relatively recent studies on the practical application of MDE to the industry [110, 210] have shown that it has already brought interesting benefits to different kinds of software-intensive projects and activities.

1.2 Problem Statement

A software migration and/or modernization project usually has three distinct and consecutive phases, as shown in Figure 1.1:

1. The complex process of obtaining useful higher-level representations of a given (legacy) system is called *reverse engineering* [43]. Its main final objective is to provide a better comprehension of the purpose and current state of the concerned system.
2. Forward engineering then analyzes those models and transforms them (when necessary) into the specification of the new version of the system [14].
3. Developers or automated code generation techniques (or a combination of both) use these models to produce the corresponding code for the targeted platform [107].

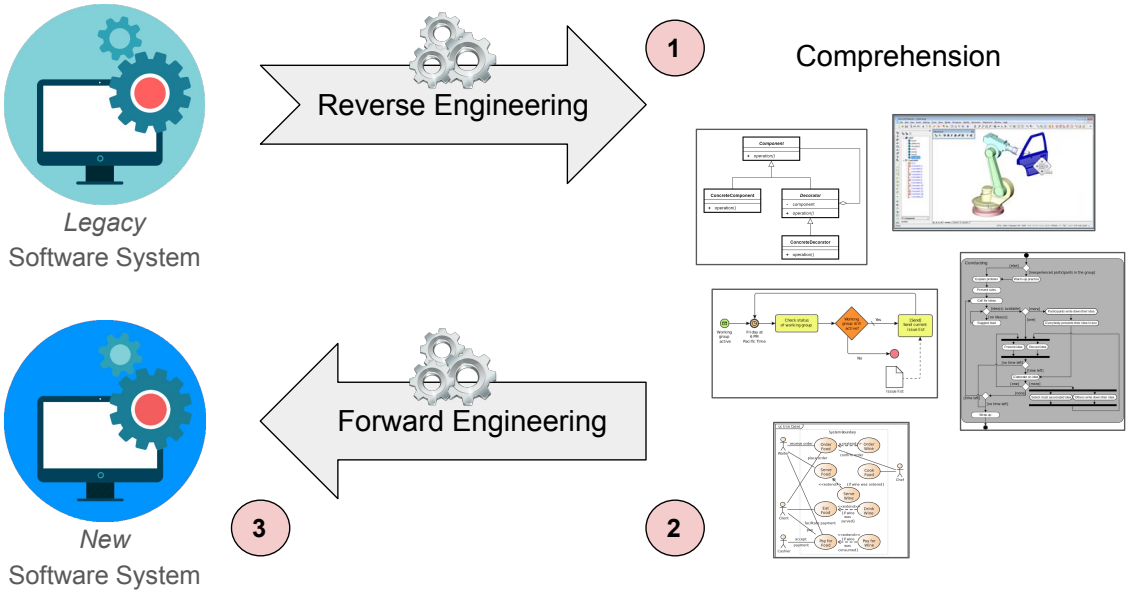

Figure 1.1 – The three phases of a software modernization and/or migration project.

Combinations of different existing solutions and technologies can help realizing some of the necessary tasks. For example, this is the case of tools that automatically produce different kinds of models from already existing software source code. This is also the case of code generators targeting various programming languages and technical platforms.

However, to the best of our knowledge, there is currently no global methodology that can guide decision makers, software architects and engineers through the whole process.

This is particularly true in the context of the previously introduced reverse engineering phase, which is the focus of the work presented in this manuscript. Related important challenges [38] notably include:

— The generic support for heterogeneous software technologies (e.g. different programming languages).
— The ability to extract useful data from the system (e.g. going further than the abstract syntax tree).
— The coverage of both the static and dynamic dimensions of the system (e.g. including runtime data).
— The capability to obtain relevant views of this system and to deal with the traceability between them.

The latter is even considered as a fundamental activity and goal of software reverse engineering and comprehension.

Quite recently (compared to the global history of reverse engineering), Model Driven Reverse Engineering (MDRE) [181] has been proposed to enhance more traditional reverse engineering processes. MDRE can be defined as the application of MDE principles and techniques in the context of reverse engineering activities. Supporting approaches notably depend on existing model-based techniques in order to obtain models representing the system according to different aspects and at different abstraction levels. Well-known examples of such models are Unified Modeling Language (UML) [167] class diagrams, state machines or workflows (just to cite a few).

Relying on these different obtained models, it is then possible to further analyze and comprehend the reverse engineered system. To this intent, expected outcomes include the provisioning of coherent and complementary model-based views over this system [27]. Such model views allow federating relevant information coming from various interrelated models, notably the ones previously obtained by MDRE. These model views have to be specified and computed according to viewpoints which are appropriate to the different stakeholders involved in the process.

However, an important limitation of the existing MDRE and related model view/federation approaches is that they quite often rely on custom and case-specific integrations of different tools [201]. Moreover, the corresponding technical solutions can sometimes be (very) heterogeneous or costly. This can complicate even more the situation and may hinder their actual deployments and practical usages.

In this thesis, we argue that complementary generic and extensible solutions are still missing (or providing an incomplete support) for MDRE to be combined with advanced model view/federation capabilities. More specifically, such solutions should allow addressing several kinds of reverse engineering and comprehension scenarios which possibly rely on different legacy technologies and platforms. Additionally, providing open source technical implementations of the proposed approaches should allow fostering their interoperability as well as their dissemination.

1.3 Global Approach

The objective of this thesis is to improve the reusable support for possibly heterogeneous reverse engineering and comprehension use cases. We notably intend to facilitate the elaboration of conceptual and technical solutions to this kind of problems. In order to achieve this, our proposition is to rely on two complementary, generic and extensible model-based approaches.

We started by capitalizing on both the different existing practices (from the state-of-the-art) and our own past experiences (e.g. collaborating with companies in the context of real software modernization projects). From this scientific and practical knowledge gained over the years, we were able to specify the overall process and supporting architecture of a generic and extensible MDRE approach. We also worked on a concrete implementation of this conceptual approach as a first open source framework, in direct partnership with a company expert in the area. Initially, the designed solution was mostly focusing on the reverse engineering of the structural (i.e. static) aspects of software systems. We then complemented this framework with further research and related developments on the reverse engineering of the behavioral (i.e. dynamic) aspects of these systems.

Complementary to MDRE, and as introduced before in Section 1.2, model federation capabilities are expected in order to improve software (model-based) comprehension activities. Thus, we proposed a generic (in the sense of metamodel-independent) approach for building and handling model views in a transparent way. Such model views can possibly federate together various sets of heterogeneous models, according to different viewpoints expressing the aspects from which the reverse engineered system has to be observed. Based on the notion of model virtualization, we have also developed a non-intrusive implementation of this conceptual approach. This has resulted in a second open source framework, that can notably consume models produced by our first framework dedicated to MDRE (among other models coming from different sources).

As both relying on the same MDE base principles and techniques, the two proposed conceptual approaches can be naturally combined or chained together. The practical integration of the two corresponding open source frameworks is made possible thanks to the use of a commonly shared modeling environment. Based on these two approaches and frameworks, more elaborated software reverse engineering and comprehension solutions can be designed in order to address potentially complex scenarios. Moreover, it is important to notice that the proposed contributions can also be reused (independently or not) in collaboration with model-based approaches addressing other issues, in a software engineering context or related to other kinds of activities.

1.4 Proposed Contributions

In this section, we summarize the contributions of this thesis and introduce how they can be combined together to provide support for software reverse engineering and comprehension activities. As mentioned before, the technical frameworks implementing the proposed conceptual approaches are all built on top of the Eclipse Modeling Framework (EMF). EMF is the *de-facto* standard modeling framework and environment in the Eclipse ecosystem (cf. Section 2.3). Note that all the presented technical solutions are fully open

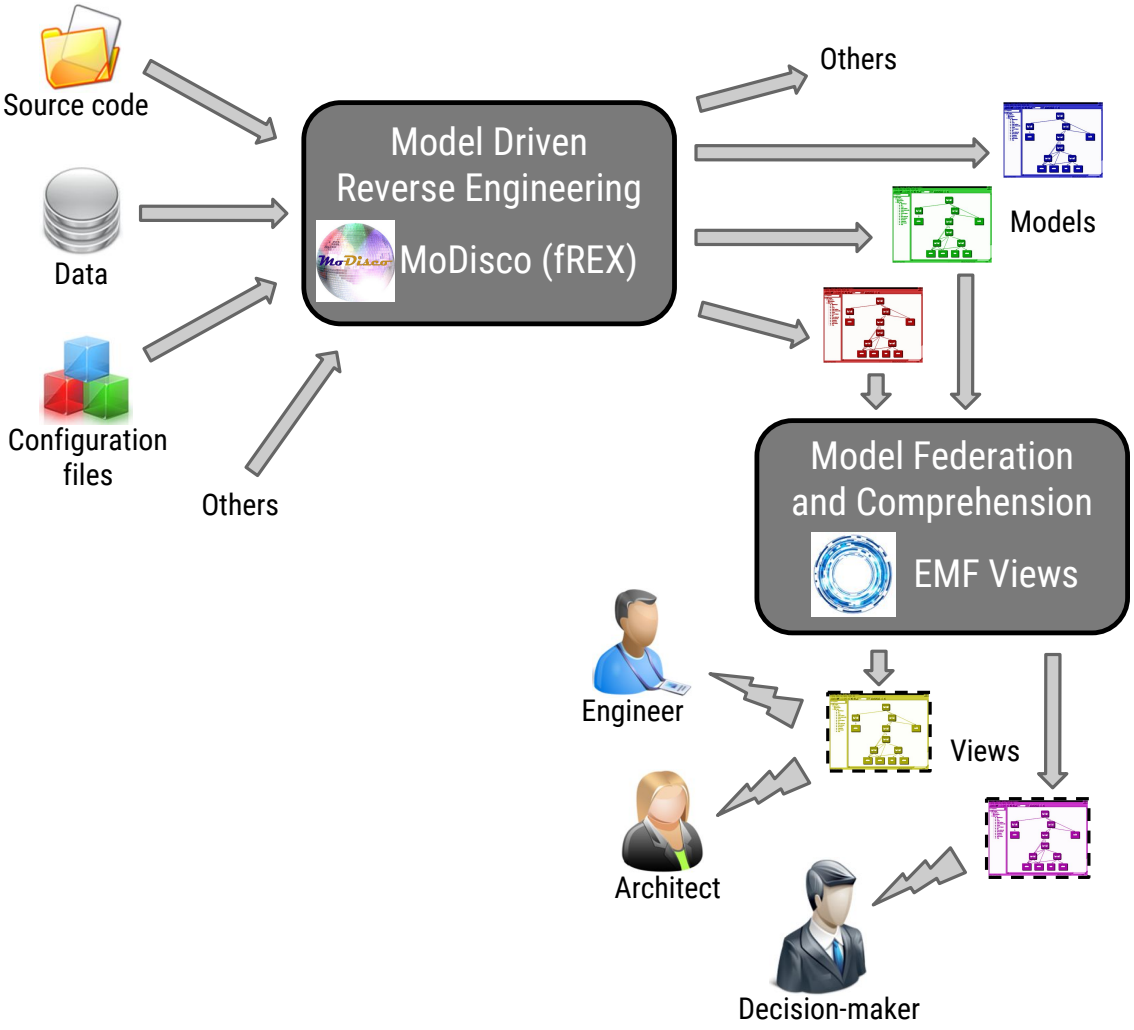

Figure 1.2 – An ecosystem for the model-based reverse engineering and comprehension of existing Software systems.

source, their source code is available online through their respective websites (cf. also corresponding Chapter 3 and Chapter 4 for more resources on them).

As depicted in Figure 1.2, the thesis contributions are the following:

— MoDisco proposes a generic, extensible and global MDRE approach intending to facilitate the elaboration of MDRE solutions in many different contexts. It is realized as a ready-to-use framework being an official Eclipse project on top of the Eclipse/EMF environment. Among other possible outputs (e.g. generated source code, retro-documentation), this framework notably allows to obtain different kinds of models out of existing systems and the various artifacts composing them (source code, data, etc.). Such obtained models can then be reused for further federation and comprehension, e.g. using EMF Views as introduced right after.

— fREX proposes an open and extensible framework, based on the MoDisco approach, that is capable of automatically generating and executing behavioral models expressed in Foundational UML Subset (fUML). It also provides an initial base mapping between UML activity/class diagrams (i.e. fUML) and the core language features of Java.

— EMF VIEWS proposes a generic and extensible approach intending to specify, build and manipulate views over sets of various and varied already existing models. Such views can be relevant to different kinds of stakeholders (e.g. software engineers or architects, decision-makers). The proposed approach can notably consume the models previously produced thanks to MODISCO , but can also reuse models coming from other sources (e.g. manually defined by some engineers). It relies on a model virtualization backend that allows referring to the content of such models in a transparent and non-intrusive way. It is also realized as a dedicated framework implemented on top of the Eclipse/EMF environment.

— Complementary to the EMF VIEWS approach and implementing framework, we also contribute a detailed survey on model view approaches. Thanks to this extended study of the state-of-the-art, a feature model is notably proposed in order to help characterizing the different available solutions and selecting the one(s) more appropriate in given contexts.

It is important to mention that these contributions can also be used individually in other contexts than the one described as the main motivation and scope of this thesis. This is notably the case for the second contribution that provides general model federation and integration capabilities via a model-based view support.

1.5 Thesis Context

I have been working as a research engineer in the NaoMod research group (formerly part of the ATLAS team, and then known as the AtlanMod team) for more than 12 years. This has allowed me to gain a solid research experience in Modeling/MDE principles and techniques as well as their possible applications to different problems. Thus, during all these years I have been able to conduct various research works in the *Software Engineering* area. The topics I had the opportunity to work on notably include global model management (also known as *megamodeling*) [205], tool and language interoperability [28], reverse engineering [30], viewpoint/view approaches [27], Cloud Computing [26, 2] or Cyber-Physical System (CPS) [1].

The thesis presented in this manuscript has been written in the context of the *Validation of Professional Experience and Knowledge* program (known under the VAE acronym in France) that is proposed by French universities. At PhD-level, this graduation program is dedicated to persons already having a long-term research experience and significant corresponding scientific results. It offers them the possibility to validate such previously obtained results via the writing and official defense of a PhD thesis, under the supervision and evaluation of habilitated professors.

In the present case, as the concerned working period is quite long and spans over more than a decade, I decided to put the focus on its second half and more particularly on the last 4-to-5 years (voluntarily excluding some of my work on other topics than the ones addressed in this manuscript). This also corresponds to a period in which I have been actively doing research (i.e. studying the state-of-the-art, designing conceptual approaches, architectures and related technical solutions, specifying required metamodels and/or languages, writing research papers accordingly) while supervising and collaborating with engineers that dealt with the associated implementation work.

Thus, as previously introduced in Section 1.3 and Section 1.4 (respectively), the approach and contributions proposed in this thesis relate to a restricted set of my publications coming mostly from this period of time. These publications, listed in next Section 1.6, are the ones particularly relevant in the context of the scientific problem described in Section 1.2.

1.6 Scientific Production

As introduced in the previous section, the present thesis focuses on a coherent subset of the research work I conducted during the last decade or so. The contributions described in this manuscript (cf. Section 1.4) correspond to a number of publications selected among the ones I have been first-authoring (for most of them) or involved in (for a few of them) during this period of time. Thus, in the particular context of this thesis, we have considered the following 11 papers that have been peer-reviewed and published in different venues: 5 international journals, 4 international conferences and 2 international (selective) workshops.

— Journals

1. Afzal, W., **Bruneliere, H.**, Di Ruscio, D., Sadovykh, A., Mazzini, S., Cariou, E., Truscan, D., Cabot, J, Gomez, A., Gorronogoitia, Y., Pomante, L. & Smrz, P. The MegaM@Rt2 ECSEL Project MegaModelling at Runtime - Scalable Model-based Framework for Continuous Development and Runtime Validation of Complex Systems. In *Journal of Microprocessors and Microsystems (MICPRO)*, 2018. Elsevier.
   — https://hal.archives-ouvertes.fr/hal-01810002
2. **Bruneliere, H.**, Burger, E., Cabot, J. & Wimmer, M. A Feature-based Survey of Model View Approaches. In *Journal on Software and Systems Modeling (SoSyM)*, 2017. Springer. **SoSyM 2018 Best Paper Award**, also published and presented at the *ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*.
   — https://hal.inria.fr/hal-01590674
3. **Bruneliere, H.**, Cabot, J., Canovas Izquierdo, J.L., Orue-Echevarria, L., Strauss, O. & Wimmer, M. Software Modernization Revisited: Challenges and Prospects. In *Computer Magazine*, 2015. IEEE.
   — https://hal.inria.fr/hal-01186371
4. **Bruneliere, H.**, Cabot, J., Dupe, G. & Madiot, F. MoDisco: a Model Driven Reverse Engineering Framework. In *Information and Software Technology (IST)*, 2014. Elsevier.
   — https://hal.inria.fr/hal-00972632
5. Menychtas, A., Konstanteli, K., Alonso, J., Orue-Echevarria, L., Gorronogoitia, J., Kousiouris, G., Santzaridou, C., **Bruneliere, H.**, Pellens, B., Stuer, P., Strauss, O., Senkova, T. & Varvarigou, T. Software Modernization and Cloudification Using the ARTIST Migration Methodology and Framework. In *Scalable Computing: Practice and Experience (SCPE)*, 2014. West University of Timisoara.
   — https://hal.inria.fr/hal-01021002

— International Conferences

1. **Bruneliere, H.**, Marchand de Kerchove, F., Daniel, G. & Cabot, J. Towards Scalable Model Views on Heterogeneous Model Resources. In *ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*, 2018. ACM.
   — https://hal.archives-ouvertes.fr/hal-01845976
2. **Bruneliere, H.**, Garcia, J., Wimmer, M. & Cabot, J. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In *34th International Conference on Conceptual Modeling (ER 2015)*, 2015. Springer.
   — https://hal.inria.fr/hal-01159205
3. **Bruneliere, H.**, Garcia, J., Desfray, P., Khelladi, D.E., Hebig, R., Bendraou, R. & Cabot, J. On Lightweight Metamodel Extension to Support Modeling Tools Agility. In *11th European Conference on Modelling Foundations and Applications (ECMFA 2015) (a STAF 2015 conference)*, 2015. Springer.
   — https://hal.inria.fr/hal-01146802
4. Bergmayr, A., **Bruneliere, H.**, Canovas Izquierdo, J.L., Gorronogoitia, J., Kousiouris, G., Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria, L., Pezuela, C. & Wimmer, M. Migrating Legacy Software to the Cloud with ARTIST. In *17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, 2013. IEEE.
   — https://hal.inria.fr/hal-00869268

— International Workshops

1. Bergmayr, A., **Bruneliere, H.**, Cabot, J., Garcia, J., Mayerhofer, T. & Wimmer, M. fREX: fUML-based Reverse Engineering of Executable Behavior for Software Dynamic Analysis. In *8th Workshop on Modelling in Software Engineering (MiSE 2016), co-located with the ACM/IEEE 38th International Conference on Software Engineering (ICSE 2016)*, 2016. IEEE.
   — https://hal.inria.fr/hal-01280484
2. **Bruneliere, H.**, Cabot, J., Drapeau, S., Somda, F., Piers, W., Villa Calle, J.D. & Lafaurie, J.C. MDE Support for Enterprise Architecture in an Industrial Context: the TEAP Framework Experience. In *TowArds the Model DrIveN Organization (AMINO 2013) workshop - co-located with the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, 2013. CEUR-WS.
   — https://hal.inria.fr/hal-00869282

1.7 Outline

The rest of this thesis manuscript is structured as follows.

Chapter 2 introduces the basic model-based principles and techniques that are required to properly understand the remaining content of this thesis. This notably includes general knowledge on MDE core definitions and concepts, on some related modeling standards and techniques as well as on the Eclipse modeling ecosystem.

Then, chapter 3 presents our generic and extensible solution that intends to provide support for MDRE. This includes the presentation of both the proposed conceptual approach and the MoDisco implementing framework. The chapter also provides a detailed description of the state-of-the-art in the MDRE area, as well as a real evaluation of the contributed approach and tooling on industrial use cases.

Chapter 4 describes our generic solution that aims at offering support for Model Federation and Comprehension via the use of model views. This includes the presentation of both the proposed conceptual approach and the EMF Views implementing framework. The chapter also provides a detailed description of the state-of-the-art in the model view area, as well as a practical evaluation of the contributed approach and tooling via large-scale performance benchmarks.

Finally, Chapter 5 concludes this thesis by summarizing the key contributions. It also presents the main impact of the obtained results via the (industrial) collaborative projects in which these contributions have been actually developed and applied. Moreover, it provides our main general lessons learned as well as our perspectives and related future work concerning the proposed contributions.

2

Background

In this chapter, we introduce the background knowledge that is relevant in order to properly understand the contributions presented later in this manuscript. We start by describing the Modeling/MDE core principles, concepts and current related challenges. Then, we continue by giving an overview of some important modeling standards and technologies. Finally we present the Eclipse open source project and platform, notably EMF as the main technical basis behind the work described in this thesis.

2.1 Modeling and Model Driven Engineering (MDE)

As introduced before, the contributions presented in this manuscript take part in the Modeling/MDE area. Thus, in what follows, we provide its general definition (Section 2.1.1), explain its core concepts (Section 2.1.2) and summarize its current main challenges (Section 2.1.3).

2.1.1 General Definition

MDE [120, 182, 20] is a software engineering paradigm relying on the intensive creation, manipulation and use of models having various and varied natures. Even though they may differ in terms of scope, MDE and its frequently encountered siblings (Model-based Engineering [82], Model-driven Development [207], etc.) can be all grouped together under the generic name *Modeling* [24]. They all state that many benefits can be gained by moving from traditional code-centric approaches to model-based or model-driven ones, thus raising the considered level of abstraction. In any case, the main big objectives can be summarized as follows:

1. Capitalize on the already existing generic knowledge (e.g. on concerned domains, processes, technologies, data) in order to foster extensibility and reusability.

2. Increase problem and/or system comprehension in order to better manage the complexity, and thus facilitate the integration, maintenance or evolution phases.
3. Improve the overall efficiency of the different software engineering activities, notably by (semi-)automating certain tasks whenever relevant and possible.

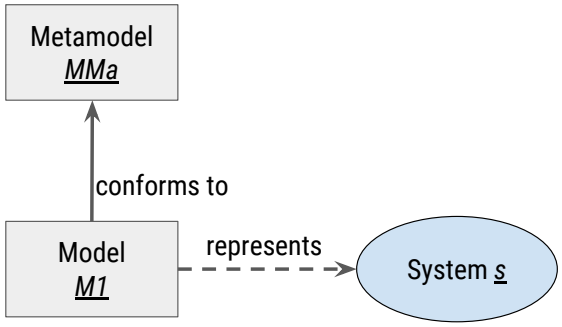

Figure 2.1 – System, model and metamodel.

As depicted in Figure 2.1, a model can be defined as the result of the observation of a system considering a given metamodel that specifies particular concerns or purposes (from which this system is observed) [9]. Depending on the situation, such a model can be obtained manually, automatically or via a combination of approaches requiring both. Modeling is largely based on the assumption that *everything is a model* [19]: models can describe different (potentially all) complementary aspects/dimensions of both the modeled systems and the related engineering activities. Thus, in model-based approaches, models are considered as first-class entities within design, development, deployment/integration, maintenance and evolution processes. This concerns the software systems to be produced, but also their environments as well as their handled data.

2.1.2 Core Concepts

The Three-level Modeling Stack

Several slightly different *Modeling* definitions coexist in the state-of-the-art [146]. They usually come with associated frameworks allowing to design and relate together models that can deal with various abstraction or hierarchical levels (cf. *multilevel metamodeling* for instance [8, 204]). However, it is generally admitted in the community to rely on a fixed three-level modeling stack [20]. As shown from Figure 2.2, this stack is based on three main concepts at three different levels of abstraction: *metametamodel*, *metamodel* and *model*.

In order to facilitate comprehension, we can make the analogy between the MDE technical space and two others technical spaces having a similar global structuring:

— MDE: models, metamodels and the metametamodel.
— Extended Backus-Naur Form (EBNF) [112]: programs, grammars and the grammar of EBNF.
— EXtensible Markup Language (XML) [215]: documents, schemas and the schema of XML Schema.

A *metametamodel* (corresponding to the *M3* level) defines the minimal set of meta-elements required in order to allow the specification of *metamodels*. Similarly to the

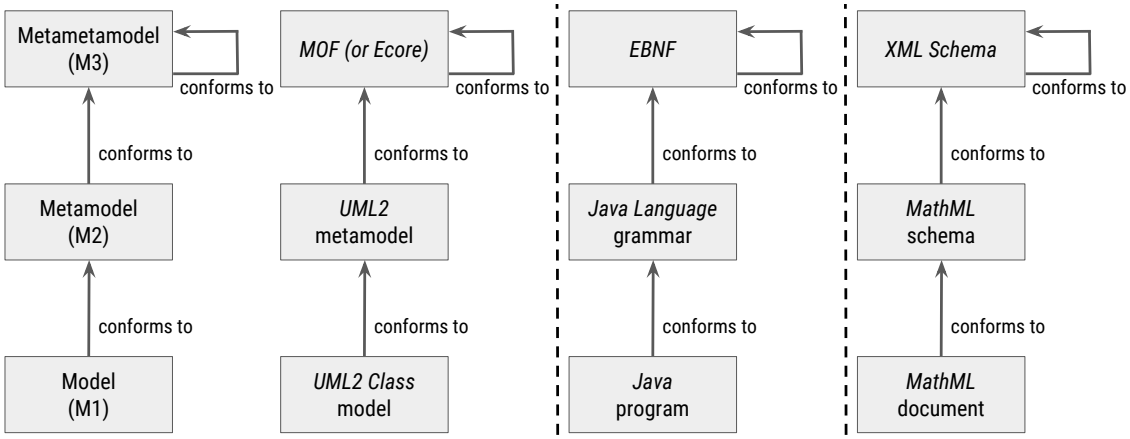

Figure 2.2 – The three-level Modeling stack and similar structuring in the EBNF & XML technical spaces.

grammar of EBNF or to the schema of XML Schema, it is reflexive: it can also be expressed using these same meta-elements it defines. In other words, it *conforms to* itself. There are various existing metametamodels, e.g. in different modeling environments [121]. For example, MetaObject Facility (MOF) is a well-known standard metametamodel specification and Eclipse Modeling Framework (EMF) (meta)metamodel (Ecore) is often considered as its de-facto reference implementation (cf. Section 2.2 and Section 2.3 respectively).

A *metamodel* (corresponding to the *M2* level) defines the possible model element types (i.e. their structure) and relationship types in order to allow the specification of *models*. In the same way than a grammar (e.g. for Java [170]) in the EBNF technical space or a schema (e.g. for MathML [216]) in the XML one, a *metamodel* is expressed based on the higher-level reference model: it *conforms to* a *metametamodel*. There is an infinity of possible metamodels, some of them being general-purpose (e.g. UML for software engineering, as described in Section 2.2) while some others being more domain-specific (i.e. for particular application domains such as the automotive industry [94] or the manufacturing industry [56]).

A *model* (corresponding to the *M1* level) defines the actual model elements describing the observed system according to the element and relationship types, as provided by the metamodel it conforms to (cf. also Section 2.1.1). As a program in a given language conforms to the grammar of this language, or an XML document conforms to an XML schema, a *model conforms to* a *metamodel*. There is also an infinite number of possible models, depending on the metamodels they conform to and the systems which are observed/modeled. For example, a UML Class model describes a given set of classes (as well as associations between them) according to the UML metamodel specification.

Model Transformation

Model transformation is another fundamental concept in MDE [186, 142, 47]. According to the state-of-the-art in this area, there are two main types of model transformation: *model-to-model transformations* and *model-to-text transformations*. While the first type is commonly referred to as *model transformation*, the second one is very often called *code generation* and more particularly in the industry [107, 119].

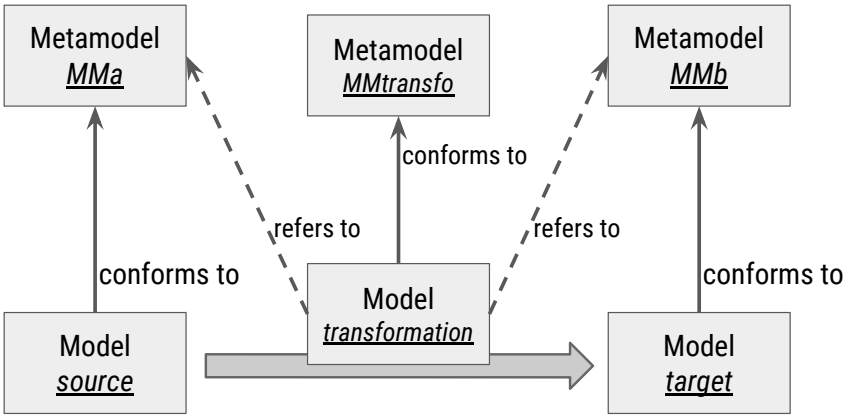

Figure 2.3 – Model-to-model transformation.

A *model-to-model transformation* specifies a mapping from a source metamodel to a target metamodel, as shown on Figure 2.3. When executed on an actual source model (as input) that conforms to the source metamodel, the defined transformation produces a new target model (as output) that conforms to the target metamodel. Some transformation languages can support multiple (meta)models as possible inputs and outputs. In this case, the specified mapping at metamodel-level can be one-to-one but also one-to-many, many-to-one or many-to-many. Moreover, some transformation languages can be bidirectional [108]. In these cases, a single mapping definition between two metamodels can allow performing (at least partially) transformations on corresponding models in both directions, i.e. from one metamodel to the other and vice-versa.

As depicted in Figure 2.3, a transformation can be itself expressed as a model that *refers to* the source and target metamodels it maps together. This *transformation model* thus *conforms to* a *transformation metamodel* that specifies the transformation language used to express the transformation. Interestingly, such a *transformation model* can be generated automatically as an output of a model-to-model transformation or can be taken as input of another model-to-model (or model-to-text) transformation. This particular kind of transformation, and its interesting characteristics, is known as Higher-Order Transformation (HOT) in the community [197].

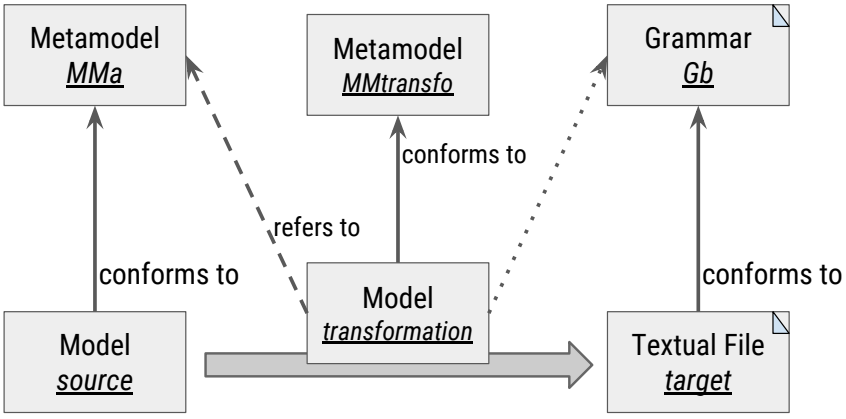

Figure 2.4 – Model-to-text Transformation, also commonly known as Code Generation

A *model-to-text transformation*, or code generator, usually specifies a mapping from a source metamodel to a target grammar, as shown on Figure 2.4. When executed on

an actual source model (as input) that conforms to the source metamodel, the defined transformation produces a new target textual file (as output) that conforms to the target grammar. Theoretically, the underlying conceptual approach is very similar to the one of model-to-model transformation: only the nature of the target differs in the case of model-to-text transformation. However, in practice, model-to-text (or code generation) techniques are quite different. They are often based on template languages that do not directly refer to the target grammar, but rather express explicitly the actual strings to be printed out in the target textual file.

2.1.3 Challenges

Since more than a decade, there is an active international research community in the Modeling/MDE domain that is already rich of principles and approaches. This comes with many related scientific challenges [92] concerning notably:

— Modeling languages themselves, i.e. their proper design, development or rigorous analysis.
— Separation of concerns, e.g. heterogeneity and integration of multiple models and modeled systems.
— Model management, e.g. model transformation, traceability, evolution, consistency or versioning.

In addition to these vertical research problems, practitioners have also regularly emphasized on important horizontal issues such as *scalability*, *efficiency*, *quality* of models or *integration* with other technologies [184]. In order to address all these challenges, the community is backed by an already solid ecosystem of existing practices, tools and concrete use cases coming both from academics and industrials [24].

Over the past years, there have been several extended studies intending to assess the general level of maturity of Modeling/MDE principles and techniques, more particularly within the industry. Initial ones were showing interesting benefits at a small- or medium-scale but also a lack of empirical evidences of the success of MDE at a large-scale [145]. More recent ones [110, 210] have been able to measure more accurately its real impact in terms of productivity, portability, maintenance, etc. One of the main findings is that MDE is already quite widespread, but many times hidden behind technical solutions or languages (notably Domain Specific Language (DSL)). Another important claim is that raising the level of abstraction has a cost, e.g. in terms of training or evolution of the legacy. Thus MDE appears to be more suitable when introduced progressively, starting by well-identified projects in which significant gains can be easily observed. This generally facilitates the faster and wider acceptance of model-based approaches, even in organizations not traditionally dealing with software.

Among the most frequently encountered practical usages of MDE, we can notably mention (i) (semi-)automated software development (e.g. via code generation techniques), (ii) the support for system, data or language interoperability (e.g. via a combination of metamodeling and model transformation), (iii) reverse engineering from existing software (e.g. via model discovery and understanding techniques, cf. Chapter 3 of this manuscript) or (iv) overall software and system comprehension (e.g. via model federation techniques, cf. Chapter 4). In addition to these, and since already several years, one of the most prominent areas concerns the support for DSL design, development and use [90].

Finally, the growing development of more and more complex CPSs in different sectors of the industry has created new needs in terms of Modeling [55]. This notably implies to take into account not only design models (generally static), as in the case of traditional Modeling approaches, but also more dynamic models at runtime [22]. Moreover, this concerns not only software models but also complementary physical models (representing relevant aspects of the hardware parts of the modeled CPSs). In this sense, Modeling/MDE has a role to play within the various initiatives related to the "Industry 4.0" [97] or "Industrie du Futur" [93] for example.

2.2 Modeling Standards and Techniques

Along the years, Modeling/MDE has been largely implemented and disseminated by relying on different related standards and techniques. In what follows, we introduce the Model Driven Architecture (MDA) initiative and vision as a key long-term actor in this domain (Section 2.2.1). We also provide insights on modeling standards which are relevant to the overall context of this manuscript (Section 2.2.2).

2.2.1 OMG's Model Driven Architecture (MDA)

In the early 2000s, Modeling has been largely popularized by the Object Management Group (OMG) under the MDA trademark [158]. Probably because of that, there has been many confusions over the years when it comes to the actual similarities and differences between MDA and MDE. To make it simple and concise, MDA can be considered as a particular subset of MDE [24] that proposes a global modeling approach based on a standardized set of generic modeling language specifications [21] (cf. Section 2.2.2 for details on some of these standards).

The main principle behind the MDA approach is to create and maintain a clear separation between 1) the business and application logic (i.e. the domain) and 2) the platform technologies on which the system is intended to be finally deployed. This approach has been designed having two main concerns in mind:

— Portability of the approach and related (meta)models in different contexts (vendor-neutrality).
— Interoperability between different technology platforms and tools.

As already introduced in Section 2.1.1, these concerns directly relate to the overall MDE objectives of capitalizing on existing knowledge and on fostering its reusability within the various software engineering activities. Figure 2.5 shows that the global MDA vision principally relies on three kinds of models [123].

The Computation Independent Model (CIM) is the higher-abstraction model that represents the business view of the solution to be produced. Its objective is to describe the requirements for the final application(s) in a way that is fully independent from how it/they will be actually implemented. Then each Platform Independent Model (PIM) provides a description of the algorithms and handled data for a given technological paradigm, but still independently from any concrete implementation technology. Finally, each Platform Specific Model (PSM) is a lower-level model representing the detailed information about the real implementation on a given technical platform.

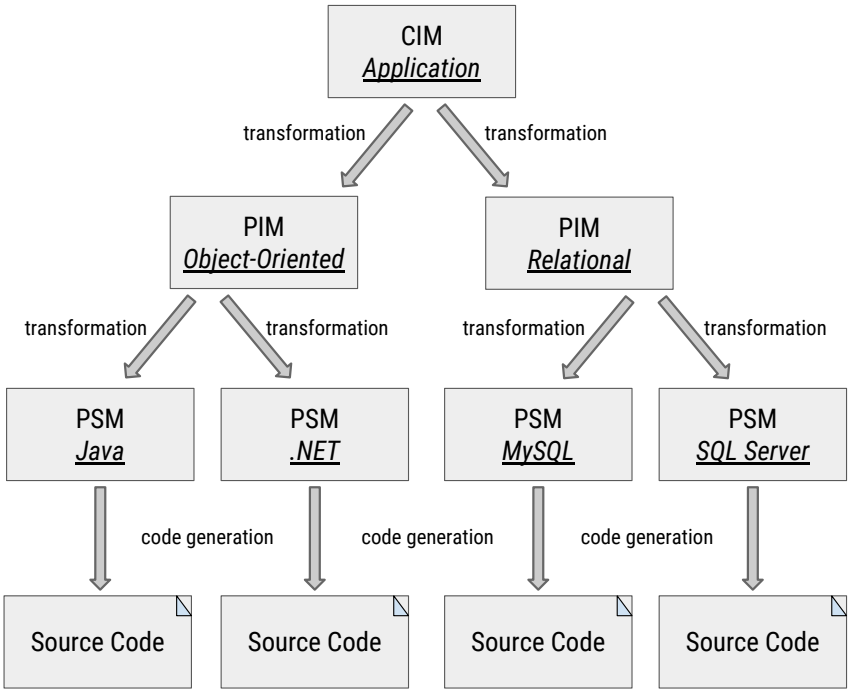

Figure 2.5 – MDA vision: CIM, PIM and PSM.

In terms of process, a single CIM (defined manually) can be first transformed into different PIMs, each one of them being then transformed into different PSMs. From these various PSMs, the source code of distinct implementations of the same solution can be generated targeting different implementation platforms. Theoretically, by previously building or retrieving the needed PIM and PSM metamodels as well as the related model transformations and code generators, the whole process could be completely automated. However this vision is very complicated to fully realize for various reasons, cf. the challenges presented in Section 2.1.3 for example. In practice, the different metamodels, models and related artifacts (transformations, generators) are often partial and the automatically generated code has to be completed manually.

2.2.2 Related Standard Specifications

As introduced in the previous section, MDA relies on a set of standard modeling languages to be used together in order to materialize its vision. Among all the available standard specifications provided by the OMG, we can notably list the following ones. These correspond to well-known or widely used standards which are also relevant in the context of this manuscript:

— MOF [157], a base metametamodel that defines the reference types for specifying various metamodels (including those listed right after).
— Object Constraint Language (OCL) [160], a declarative textual language that allows to define constraints and queries (and thus to navigate) over any MOF-based metamodel or model.
— XML Metadata Interchange (XMI) [168], a common model interchange format that is used to serialize and then share in an uniform way all the MOF-based metamodels and models.

- General-purpose modeling languages including more particularly
  - UML [167], a modeling language dedicated to software engineering and the design of software systems (covering both the structural and behavioral aspects of such systems).
  - Systems Modeling Language (SysML) [165], a modeling language (defined as an extension of UML via its profile mechanism) dedicated to system engineering and the design of complex system or systems-of-systems,
- Generic (in the sense of metamodel-independent) and complementary transformation languages
  - Query/View/Transformation (QVT) [161], three model-to-model transformation languages named *QVT-Operational* (imperative, for unidirectional transformation only), *QVT-Relations* (declarative, for both unidirectional and bidirectional transformation) and *QVT-Core* (declarative, a target simple executable language for *QVT-Relations*).
  - MOF Model to Text Transformation Language (MOF2Text) [159], a model-to-text transformation language for generating source code or documentation from MOF-based models (e.g. from PSMs as introduced in Section 2.2.1).

We have decided not to present individually within this manuscript each one of the previously mentioned standards/metamodels, as the details can be obtained from the latest versions of the corresponding OMG specifications. An important point to mention is that, as a vendor-neutral industrial standardization organization, the OMG does not provide any (reference) implementation of their published specifications. However they do recognize compliant formats and modeling tools in addition to de-facto standard modeling environments, cf. Section 2.3 introducing the Eclipse platform and its EMF for instance.

In addition to the MDA initiative, which is still a significant part of its activity and visibility, the OMG also works on and promotes other modeling standards addressing different target communities. Some of these standards are already well-established such as the Business Process Model and Notation (BPMN) [155] (in the Business Process Management (BPM) domain), while others are the result of more recent work such as the Unified Architecture Framework (UAF) [166] (in the Enterprise Architecture (EA) domain). Another OMG initiative that is of particular interest in the context of this thesis is the Architecture Driven Modernization (ADM) [154] task force. Indeed, this working group focuses on specifying standard metamodels useful in the context of (software) system modernization activities. This directly relates to the thesis contribution presented in Chapter 3, that notably provides detailed insights on the topic of MDRE.

2.3 Modeling in/with Eclipse

Eclipse is an international and industrially-supported open source community organized around the independent not-for-profit Eclipse Foundation [61]. It has the particularity of being commercial-friendly: its Eclipse Public License (EPL) [66] allows building either open source or proprietary software on top of components licensed under EPL. Eclipse plays an important role regarding the contributions of this thesis, as providing the required technical basis for the implementation of the presented work. It is at the same time an open source community/foundation but also an open source platform for building

new tools (Section 2.3.1) and a well-recognized modeling environment (Section 2.3.2 and Section 2.3.3).

2.3.1 The Eclipse Open Source Platform

Eclipse is a well-known Integrated Development Environment (IDE) that provides development tooling (i.e. wizards, editors, views, debuggers, builders, etc.) for several widely used programming languages, for example Java, C/C++ or PHP (and now many others) [211]. The Eclipse Project [65] is a core project in Eclipse. It acts as the top-level umbrella for the following important projects (also including the modeling ones described further in Section 2.3.2 and Section 2.3.3):

— *Platform* - the set of common frameworks and features that make it possible to build and integrate within Eclipse various tooling for different purposes, or to produce standalone Rich Client Platform (RCP) applications [140]. This includes the base customizable Eclipse workbench, its user interface, resource management and versioning capabilities, a language-independent debugging infrastructure, etc.

— *Eclipse Java Development Tools (JDT)* - the complete Java IDE in Eclipse, coming with the related Java project nature, perspective, views, editors, builders, debugging facilities, etc. It directly relies on the Platform project mentioned before and is notably used to develop Eclipse plugins as mentioned right after.

— *Plug-in Development Environment (PDE)* - the tooling for plugin building and deployment in Eclipse (plugin description and extensions, related views and editors, etc.). Similarly to the Eclipse overall architecture, it implements the principles of Open Services Gateway Initiative (OSGI) [171] since the version 3 of Eclipse [104].

These three core projects, along with the Eclipse Modeling Project (EMP) and EMF ones introduced in next Section 2.3.2 and Section 2.3.3 (respectively), have been used in order to implement the Eclipse tooling realizing the scientific contributions presented in this manuscript.

2.3.2 The Eclipse Modeling Project (EMP)

EMP [64] is the top-level Eclipse project in charge of gathering and promoting model-based technologies both inside and outside the Eclipse community [103]. Thus, EMP hosts several Eclipse modeling projects offering important expected capabilities when elaborating on model-based solutions, as previously described in Section 2.1. Note that the following list is voluntarily non-exhaustive and focuses on the most widely-recognized projects as well as the ones particularly relevant in the context of this thesis:

— *Model Storage* - In addition to the default XMI file-based model serialization provided by EMF, there are complementary frameworks allowing to store, retrieve and version models into/from various kinds of databases (Connected Data Objects (CDO) [60] or model-specific repositories (EMFStore [68]).

— *Model Transformation* - The two main types of model transformation, as introduced in Section 2.1.2, are supported by tools such as ATL [59, 116] for model-to-model transformation (a QVT-like language, cf. Section 2.2.2), Acceleo [58] for code generation (an implementation of the MOF2Text standard, cf. also Section

2.2.2) or Epsilon [69] that provides a family of model transformation and manipulation languages.
— *Textual Modeling* - The support for the design, development and deployment of textual DSLs in Eclipse is ensured by the Xtext project [77].
— *Graphical Modeling* - Complementary to the support for textual DSLs, the Sirius project [74] enables the specification of graphical modeling workbenches (i.e. graphical DSLs and corresponding diagrams).
— *Modeling Tools* - There are also frameworks or tools for the creation and editing of different kinds of EMF-based models. This includes reference implementations of some OMG MDA standard metamodels (namely UML, OCL or BPMN, cf. Section 2.2.2), a UML/SysML modeling environment named Papyrus [72] or the MDRE framework MoDisco [71, 29] (cf. the thesis contribution described in Chapter 3).

All these EMP tools, frameworks or standard implementations are based on and unified by the use of EMF (cf. Section 2.3.3). This way, they can be more easily integrated together and combined in order to provide practical support for different model-based activities.

2.3.3 The Eclipse Modeling Framework (EMF)

As introduced in previous Section 2.3.2, EMF [63] is the reference modeling framework in Eclipse. It is used and shared between all the Eclipse model-based solutions, and comes with core complementary facilities for base model querying, transaction management or validation [194]. It basically consists in two main components:
— Ecore which is the EMF metametamodel commonly considered as the de-facto reference implementation of MOF (actually of Essential MOF (EMOF), the minimal subset of MOF, cf. Section 2.2.2) in the Eclipse community. It comes with a default XMI serialization support as well as a reflective model manipulation API (i.e. metamodel-independent).
— Code generators to automatically produce 1) the (extensible) Java manipulation APIs for the metamodels previously specified using Ecore and 2) (extensible) base editors to display and manipulate the corresponding models.

As a key element in order to understand the contributions presented in this manuscript, Figure 2.6 shows a simplified version of the Ecore metametamodel provided by EMF.

A metamodel defined in Ecore has a named root model element that is an *EPackage*. In such a package, two main kinds of (*EClassifier*) named elements are contained:
— *EClass* elements that specify the various concepts of the metamodel.
— *EDataType* elements that specify the base data types to be used in this metamodel.

A given data type can be either a primitive one (e.g. named String, Integer, Boolean, etc.) or an enumeration (*EEnum*). An enumeration provides a specific set of literals (*EEnumLiterals*) to be used as values of corresponding attributes (cf. next paragraph).

Each *EClass*/concept of the metamodel contains a set of named structural features (*EStructuralFeature*). An *EClass* can inherit from one of several other *EClass* elements, then also inheriting their structural features. There are two types of structural features:
— *EAttribute* elements that describe simple attributes having base data types.
— *EReference* elements that describe references to another *EClass*/concept of the

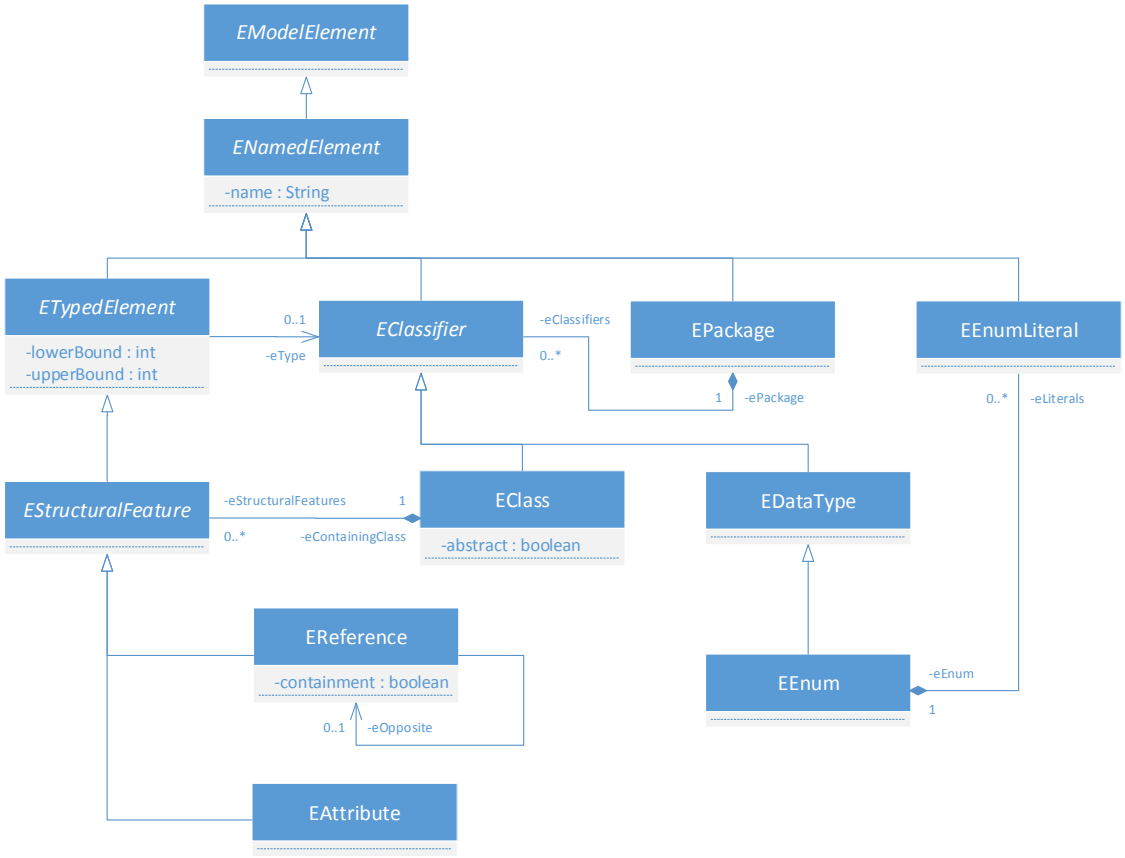

Figure 2.6 – Simplified version of the Ecore metametamodel from EMF.

metamodel.
A given reference has specific multiplicities (i.e. its *lowerBound* and *upperBound*). It can be either a standard reference or a composition (in this case *containment* is set to true). Finally it can also have an opposite reference, thus allowing navigation in both directions.

2.4 Conclusion

In this chapter, we have introduced the main principles, concepts, standards and technologies that we have used as a basis of the work presented in this manuscript.

We first described Modeling/MDE as our main background domain and defined its core concepts (metamodels, models, model transformations). We also discussed major scientific challenges in this area, directly connected to the problems we are addressing via the contributions of this manuscript.

Then, we provided an overview of the OMG's MDA vision and corresponding modeling standards. This particular realization of MDE, and more specifically several of the standards we introduced, have been used in the different contributions of this manuscript (as part of the proposed approaches or to evaluate our solutions on realistic sets of models).

Finally we presented the Eclipse open source project, platform and Modeling project as the main technical basis of the work presented in the coming chapters. We notably

put a focus on EMF as a de-facto reference modeling environment in the Modeling/MDE community.

Note that the details concerning the proposed contributions can be found from the corresponding chapters of this manuscript (namely Chapter 3 and Chapter 4). For each contribution, this notably includes a description of the state of the art in the related area(s) as well as a discussion on main current problems and challenges.

3

Model Driven Reverse Engineering

Reverse engineering is almost as old as computer science itself. Initially targeting hardware analysis [178], it quickly extended its scope to also focus on software systems [43]. Then, following the spectacular expansion and advent of software from the end of the 80s, Reverse Engineering has been usually regarded in the context of dealing with *legacy systems* (i.e. already existing applications) which are often still running critical operations for companies.

In contrast with forward engineering, *reverse engineering* is commonly defined as *the process of examining an already implemented software system in order to represent it in a different form or formalism and at a higher abstraction level* [43]. The key notion here is the one of *representation*, that can be associated to the concept of *model* as defined in previous Section 2.1.

The overall objective of such representations is to have a better comprehension of the current state of a software system, for instance to correct it (e.g. to fix bugs or ensure regular maintenance), update it (e.g. to align it with the constantly evolving organizational policies and rules), upgrade it (e.g. to add new features or additional capabilities), reuse parts of it in other systems, or even completely re-engineer it. These evolutions are happening more and more for various reasons. This is notably due to the need for not only satisfying new user requirements and expectations but also for adapting legacy systems to emerging business models, adhering to changing legislation, coping with technology innovation (in terms of used environments, frameworks, libraries, etc.) and preserving the system structure from deteriorating [38].

Given that reverse engineering is a time-consuming and error-prone process, any reverse engineering solution that could (semi)automate the process would bring precious help to users (e.g. software architects or engineers) and thus facilitate its larger adoption. Previous attempts in the 90s to build (semi-)automated solutions were first based on object-oriented technologies [40]. Several proposals and tools in that area, e.g. aimed at *program comprehension* [149], appeared around at that time. Among the many propo-

sals, some focused on the extraction and analysis of relevant information from existing source code or software components in general [37] whereas others focused on relational databases [175], compiled code or binary files [79], etc. However, these efforts were quite specific to a particular legacy technology or a given reverse engineering scenario (e.g. technical migration, software analysis).

With the emergence of MDE (cf. Chapter 2), MDE principles and core techniques have been used in order to build effective reverse engineering solutions: this is called MDRE (cf. Section 3.1). MDRE formalizes the representations (i.e. models) derived from legacy systems to ensure a common interpretation of their contents. These models are then used as the starting point of all the reverse engineering activities: MDRE can then directly benefit from the various capabilities of MDE as introduced in Section 2.1. Nevertheless, there is still a lack of complete solutions intended to cover full MDRE processes.

In this chapter we present MoDisco (and its child fREX), as the first main contribution of this thesis (cf. Section 1.4), that is both:

1. A generic, extensible and global MDRE approach to facilitate the elaboration of MDRE solutions in many different contexts.
2. A ready-to-use framework, implementing this approach as an official Eclipse project on top of the Eclipse/EMF environment.

Our proposed approach is based on a modular architecture which has been created to address different types of legacy systems and to target different (model driven) reverse engineering processes such as technical/functional migration, refactoring, retro-documentation, quality assurance, etc.

The rest of this chapter is structured as follows. Section 3.1 provides the current state of the art and main challenges related to MDRE. Section 3.2 describes in details the proposed conceptual approach and its two main phases, namely *Model Discovery* and *Model Understanding*. Section 3.3 presents the MoDisco technical framework implementing this approach, i.e. its overall architecture and different provided components. Section 3.4 explains how we evaluated the approach and related Eclipse/EMF-based framework, via both concrete use cases and performance benchmarks. Section 3.5 introduces the fREX component, as the initial result of further research work on applying the proposed approach (and MoDisco framework) to target the reverse engineering of system executable behaviors. Section 3.6 concludes this chapter by summarizing the main realizations as well as their current limitations.

3.1 State of the Art and Challenges

As introduced before, MDRE is commonly defined as the application of MDE principles and techniques to Reverse Engineering challenges. MDE and its main related techniques and standards have already been presented in Chapter 2. Thus, in what follows, we start by giving an overview of MDRE (in Section 3.1.1) before describing specific Reverse Engineering solutions (in Section 3.1.2) as well as generic Reverse Engineering platforms and frameworks (in Section 3.1.3). We end this section by presenting some important challenges regarding Reverse Engineering and MDRE (in Section 3.1.4) that we intend to address via the contribution presented in this Chapter.

3.1.1 Overview

The application of MDE to Reverse Engineering (i.e. MDRE) is relatively recent [181]. At the beginning, models (in the MDE sense) were mainly used to specify systems prior to their implementation (in other words, for forward engineering activities). Instead MDRE proposes to build and use models from the system implementation, thus directly benefiting from these higher-level views of the system (e.g. design models) or of its domain (e.g. application domain models) in order to improve the maintenance and evolution processes. MDRE is considered a fundamental application of MDE [83]. There is an inherent complementarity between forward engineering and reverse engineering, especially when homogeneously treated and combined using models. This integration notably enables a continuous re-engineering of the systems.

The growing interest in MDRE motivated the OMG to launch the ADM Task Force [154] with the main objective to propose a set of standard metamodels useful for modernization projects, i.e. technical migrations from old or obsolete technologies to more recent ones. Based on this, various proposals combine these standards in a methodological framework [84] . MDRE is also useful for software analysis purposes (e.g. as proposed by the Moose platform [99]). More generally, MDRE is required when dealing with *Model Driven Software Evolution* [202], i.e. global scenarios including any kind of possible modification on legacy systems (structural, functional, maintenance, etc.) and not only pure (technical) modernization. In all cases, a MDRE phase is needed first in order to obtain the required models from the considered systems so that they can then be analyzed, modified or evolved.

Language workbenches (also sometimes referred as metamodeling tools) have paved the way for generic and extensible MDE approaches like ours, by facilitating the creation of new metamodels such as the ones we used. Nevertheless, these initiatives (such as MetaEdit [187] and then MetaEdit+ [199]), GME [51] or MetaEnv [13] were more focused on the creation of a modeling environment for new metamodels and on providing some related support for typical forward engineering activities, i.e. code generation to some popular languages.

When it comes to approaches supporting reverse engineering, we first distinguish two main families: specific solutions and general-purpose solutions. This is determined depending on whether they aim to reverse engineer the system from a single technology and/or with a predefined scenario in mind (e.g. a concrete kind of analysis), or to be the basis for any other type of manipulation in later steps of the reverse engineering process. Of course, both lists are not completely disjoint and sometimes it can be argued that a tool could go either way.

Within Table 3.1 and Table 3.2, we provide some representative examples for each one of these two families of solutions. We also classify these different solutions according to simple high-level categories depending on their family. As expected, none of the specific solutions covers all the mentioned MDRE capabilities (note that we just considered the most common MDRE objectives, we did not intend to be exhaustive). Moreover, apart from the generic software modeling tools also providing some MDRE capabilities as part of their long feature lists, there are only a few general-purpose MDRE solutions in the current state-of-the-art. In next Section 3.1.2 and Section 3.1.3, we give more insights on the different solutions listed in Table 3.1 and Table 3.2 (respectively). In these same

	Discovery	Migration	Integration	Analysis / comprehension
Columbus [86]	✓			
JaMoPP [106]	✓			
Spoon [173]	✓			
Briand et al. [25]	✓			
Sun et al. [195]	✓			
Sneed [188] [189]	✓	✓		
Barbier et al. [12]	✓	✓		
Fleurey et al. [89]	✓	✓		
Clavreul et al. [45]	✓		✓	
Alnusair et al. [3]	✓		✓	
Ramon et al. [177]	✓		✓	
ConQAT [53]	✓			✓
GUPRO [57]	✓			✓
SWAG Kit [180]	✓			✓
CodeCrawler [133],CodeCity [209]	✓			✓
Pacione et al.[172]	✓			✓
Olsson et al.[169]	✓			✓
Fradet et al.[91]				✓

Table 3.1 – An overview of existing MDRE approaches - Specific solutions.

	Single core-metamodel approaches	Multiple-metamodel appraoches	General software modeling tools
CORUM-II [118]	✓		
Moose [99]	✓		
Our approach [29]		✓	
Garces et al. [95],[96]		✓	
Rational Software Architect [111]			✓
MagicDraw [151]			✓
Enterprise Architect [193]			✓
Modelio [191]			✓
Visual Paradigm [206]			✓

Table 3.2 – An overview of existing MDRE approaches - General-purpose solutions

sections, we also explain the rationale behind the provided classification as well as how we positioned our proposed solution.

3.1.2 Specific Reverse Engineering Solutions

There exist several specific tools allowing to discover different types of models out of legacy artifacts. Columbus [86] is offering parsing capabilities from C/C++ source code and allows serializing the obtained information using different formats (e.g. XML, UML XMI). JaMoPP [106] or Spoon [173] are providing alternative (complete) Java metamodels and corresponding model discovery features from Java source code. Complementary to this, some other kinds of software artifacts can also be relevant as inputs of MDRE processes. For example, execution traces captured during the running of a given system have been used to generate UML sequence diagrams showing dynamic views on this system [25]. In other experiments, available Web service descriptions have been expressed as UML models to be able to compose them for future integration in a different framework [195]. All these components can be seen as potential *model discovers* that could be used either jointly with our approach (cf. Section 3.2) or adapted to be plugged into our implementing framework (cf. Section 3.3).

Past works have already described how to migrate from a particular technology to another one using dedicated components and mappings, e.g. from procedural COBOL to object-oriented COBOL [188], from COBOL to Java [189], from COBOL to Java EE [12] or from Mainframe to Java EE [89]. All these tools use specific (and also sometimes proprietary) parsers, grammars, metamodels, etc. Contrary to our intent, their genericity and reusability in other contexts (e.g. in other paradigms, for different legacy technologies or target environments) is quite limited. Moreover, we are clearly differentiating from them by being fully open and relying on generic components.

MDE has also been applied in the context of legacy system integration [45], including the reverse engineering of API concepts and interfaces as high-level models. The main focus of such work was on the mapping definition between different APIs and the generation of corresponding wrappers. We could consider the obtained API models as useful inputs of more general MDRE processes, but our approach go much more beyond this specific scenario. Other particular examples of applications are automated design patterns detection using ontologies [3] or graphical interfaces separation and later reusability [177], that can also be potentially adaptable in our approach if some models can be exchanged between the solutions.

More related to software analysis, the ConQAT tool [53] is dedicated to the analysis of the source code and also of related models, textual specifications, etc. Its objective is to perform quality assessment activities such as clone detection, architecture conformance or dashboard generation. However, our goal is not to address specifically software or quality analysis problems. It is rather to provide more generic components for helping people elaborating on solutions to various MDRE scenarios, potentially including software or quality analysis but not only.

Some other works are more focused on the related problem of software/program comprehension [208]. At code-level, GUPRO [57] is a solution, based on a central graph repository plus related queries and algorithms, that offers different visualizations over a given program (via tables, code excerpts, etc.). SWAG Kit [180] is another tool that addresses

the graphical visualization of complex C/C++ systems. The CodeCrawler visualization tool [133] and more recently CodeCity [209], an advanced 3D graphical visualization tool (both based on the Moose platform, cf. Section 3.1.3), allow deeper analyzing the legacy code. There are also solutions that go further and propose a more generic software visualization model [172]. However, the purpose of our approach is not to address the problem of advanced visualizations. It is rather to provide relevant representations of legacy artifacts as models that could feed such visualization solutions if needed.

At artifact-level, some solutions provide the capability of extracting key information from software artifacts (based on a generic artifact description metamodel) and supports traceability and consistency checking between them [169]. Our approach also offers representations of such artifacts and their relationships, as models that could complement the ones obtained from the application of such solutions. The other way around, the information models produced by these solutions could be used as valuable inputs in our approach.

At the architecture level, a framework has been proposed [91] providing a formal definitions of architectural views (as diagrams/models) and an algorithm to perform consistency checks over these views. However there is no support for the automated discovery of such views/models, which is one of the main objectives of our approach. Similarly than before, our approach is not specifically intended to provide tooling for consistency checking issues (as this could be brought by other solutions such as this one).

3.1.3 Generic Reverse Engineering Platforms and Frameworks

Contrary to specific Reverse Engineering solutions/tools addressing particular activities or scenarios, there have been fewer research initiatives providing more generic integrated environments that could be extended and adapted to different application scenarios.

One of the first to appear was CORUM-II [118] which tried to integrate together various architecture and code re-engineering tools It is based on a common inter-operation schema called CORUM to be considered as a reference for the different tool providers in the area. The path our approach is following is relatively similar, i.e. using an horseshoe-like approach (that has been adapted to the MDE domain in our case). However, we are not proposing to systematically conform to such a standard representation schema. We rather provide different base metamodels (e.g. from the OMG ADM Task Force[154] but not only) and we also allow plugging other ones into the framework if required by a given MDRE scenario.

Moose [99] is a well-recognized platform providing a toolbox dedicated to the building of various software and data analysis solutions. Thus the Moose Suite proposes a set of tools built-up around this same platform, all relying on a common fixed core named FAMIX that is dedicated to the representation of object-oriented systems. The main difference with our approach is that we are not relying on a single common core, but on a more open solution where technology-specific metamodels (for object-oriented technologies or others) are used to avoid information loss during the initial discovery phase. A more generic metamodel, e.g. to reuse already existing capabilities, could be used later on by transforming the specific models to such a generic single one.

Recently, a generic white-box solution has been developed in order to support the semi-automated modernization of legacy applications [95]. It proposes to use open and iterative model transformation processes that rely on technology-agnostic metamodels adapted to the targeted architectures. This solution has notably been applied and evaluated in the particular context of database forms (graphical interfaces) migration [96]. Interestingly, its underlying approach directly inspires from our overall MDRE approach and could quite naturally extend or reuse some of the components from our implementing framework (e.g. model discoverers, standard metamodels).

In parallel with the previously mentioned research initiatives, there are several existing commercial tools providing some reverse engineering features covering the most common programming languages (e.g. Java, C/C++, C#, VB.NET, etc. .) and standard modeling languages (e.g. UML, SysML, BPMN). Well-know examples of such tools are Rational Software Architect - RSA (by IBM) [111], MagicDraw (by NoMagic/3DS) [151] or Enterprise Architect (by Sparx Systems) [193]. Some tools also come with a base open source or free version to be then completed with specific extensions to be bought, such as Modelio (by Softeam) [191] or Visual Paradigm (by Visual Paradigm International) [206].

However, all these commercial solutions are not particularly dedicated to MDRE but rather intend to support the full software development life-cycle in terms of modeling. This goes from higher-level business process and architecture specification to lower-level forward engineering (e.g. code generation), also dealing with regular project management activities for instance. In any case, the reverse engineering features these tools provide (e.g. model discoverers) could be integrated to MDRE solutions based on our proposed approach and could be combined with components from our implementing framework when needed.

3.1.4 Challenges

As introduced at the very beginning of this Chapter 3, Reverse Engineering is a complex process that can take many different forms. Thus any solution that could (semi-)automate its design, implementation and execution would bring very valuable support to the different kinds of Reverse Engineering actors. Nevertheless, such a Reverse Engineering solution would need to face several important problems:

— **Technical heterogeneity of the legacy systems**.
— **Structural complexity of these legacy systems**.
— **Scalability of the developed solution**.
— **Adaptability/portability of this solution**.

As described in Section 2.1, MDE provides useful capabilities for supporting *genericity*, *extensibility*, *reusability*, *integration* or *automation*. Inheriting of these characteristics from MDE, we believe MDRE has interesting capabilities allowing to address the Reverse Engineering problems mentioned before.

As a consequence, taking into account the state-of-the-art presented in this Section 3.1 and also our experiments on concretely applying MDRE in real projects (both from a research and industrial perspective, cf. Section 3.4 for instance), the main challenges MDRE solutions must be able to overcome are the following ones:

— **To avoid information loss due to the heterogeneity of legacy systems.** To ens-

ure the quality of the overall MDRE process, and thus to validate its actual relevance, it is very important to be able to retrieve as much information as possible from legacy systems that are often technically heterogeneous.

— **To improve understanding of the typically complex legacy systems.** The goal of MDRE is also to facilitate the understanding of structurally complex legacy systems by their users and developers. This requires going beyond the provisioning of simple low-level representations, and deriving efficiently higher abstract views with the most relevant information.

— **To manage scalability.** Legacy systems are usually huge and complex systems. Scalability of MDRE techniques must be improved to be able to load, query and transform in a suitable way the very large models usually involved in MDRE processes.

— **To adapt/port existing solutions to different needs.** Many MDRE solutions are still technology- or scenario- dependent, meaning that they target a very concrete legacy technology or reverse engineering scenario. Progress must be done in the development of generic MDRE solutions that, even if they still keep some specific components, are largely reusable in various contexts and for a minimized cost.

By proposing a generic and extensible approach to facilitate the elaboration of MDRE solutions, we aim at providing support targeting these previous challenges. As stated before, our approach itself is not intended to be a complete solution for all specific scenarios, even though its default components can be used as such for particular technologies. Its goal is rather to provide the basic (interconnected) building blocks to construct other MDRE solutions on top of it.

3.2 Proposed Conceptual Approach

This section presents the global MDRE approach we propose in order to treat as homogeneously as possible all potential reverse engineering scenarios. Intending to overcome the MDRE challenges that have been identified in previous Section 3.1.4, our objective is:

1. To identify the main steps and components commonly used in MDRE solutions.
2. To assemble them coherently as a generic and extensible approach.

In what follows, we start by providing an overview of our approach in Section 3.2.1. Then, we go more into the details of its first main phase *Model Discovery* in Section 3.2.2. We also explain its second main phase *Model Understanding* in Section 3.2.3. Finally, we end by summarizing the main benefits of our proposed approach in Section 3.2.4.

3.2.1 Overall Approach

Based on our previous analysis of the state-of-the-art in the area (cf. Section 3.2), we believe a full MDRE approach must provide the following characteristics in order to fulfill the expressed requirements:

— **Genericity** based on technology-independent standards (i.e. metamodels) and customizable model-based components where specific technology support is ensu-

red by additional dedicated features that can be plugged into the generic ones.
— **Extensibility** relying on a clear decoupling of both the represented information (as models) and the different consecutive steps of the process (as MDE operation workflows).
— **Full coverage** (if necessary) of the source artifacts based on complementary interrelated representations of the same system at different abstraction levels and using different perspectives (i.e. metamodels).
— Direct **(re)use** and **integration** of both the provided components and possibly external ones, but also of all the obtained results (i.e. models).
— Facilitated **automation** of (at least parts of) the process thanks to the already available MDE techniques, notably by chaining predefined sets of model transformations.

To achieve them in our approach, we followed the strategy of switching very early from the heterogeneous world of legacy systems to the more homogeneous world of models. Thus, we can directly benefit as soon as possible from the interesting properties of MDE as well as of the full set of already available standards and technologies (cf. Chapter 2).

The main principle is to quickly get *initial models* representing the artifacts of the legacy system without losing any of the information required for the process. These raw models are sufficiently accurate to be used as a starting point of the considered MDRE scenario, but do not represent any real increase of the abstraction or detail levels. They can be seen as (full or partial) abstract syntax graphs that bridge the syntactic gap between the worlds (i.e. the technical spaces) of source code, configuration files, etc. and the world of models. From this point on, any reverse engineering task to be performed on the system can be done with the expected result using these models as a valid input representation. Therefore, we have reduced the heterogeneity of the reverse engineering process to a modeling problem.

The models can then be used as inputs for chains of MDE operations, e.g. model transformations, in order to navigate, identify and extract the relevant information. These processed models obtained at the end of the chain, called *derived models*, are then finally used to generate and/or display the expected representations of the reverse engineered legacy system.

As presented in Section 3.1.1, MDRE is the application of MDE in a Reverse Engineering context. In order to realize it, the two main steps mentioned right before are summarized by this simple equation:

Model Driven Reverse Engineering
=
Model Discovery + Model Understanding

Model Discovery is a **metamodel-driven** phase in charge of representing the legacy system as a set of models with no loss of required information (cf. Section 3.2.2 for more details on this first phase). The **Model Understanding** phase is completely **model-based**: it takes as input the models from the previous phase and generates the required output (models) thanks to a chain of model transformations (cf. Section 3.2.3 for more details on this second phase). While the components to use in the model discovery phase largely depend on the source technology to analyze (i.e. the metamodels to employ are for instance adapted to the legacy system programming languages), the components in

the model understanding phase are more related to the actual objective of the overall reverse engineering process itself. Such a goal could be system comprehension (e.g. of its architecture or implemented functionalities), further analysis (e.g. business rule extraction, non-functional property verification), re-engineering (e.g. technical refactoring or migration), etc.

To structure these phases and components in practice, we propose the global architecture depicted in Figure 3.1. It is composed of three vertical complementary layers that favor the five main characteristics described before:

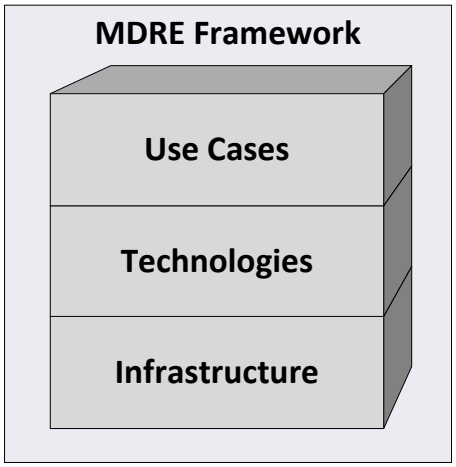

Figure 3.1 – MDRE framework architecture.

The *Infrastructure* layer provides genericity and automation via a set of basic bricks totally independent from any legacy technology and reverse engineering scenario. Such components offer for instance generic metamodels and model transformations. They also propose extensible model navigation, model customization and model orchestration capabilities (manual via dedicated user interfaces and/or programmatic via specific APIs). They often come with the corresponding generic interfaces and extension features required for the components from the other layers to be plugged in.

The *Technologies* layer is built on top of the *Infrastructure* one. It offers (partial or full) coverage for some legacy technologies and also gives the chance to extend this coverage to other ones. Its goal is to provide technology-dedicated components which however stay independent from any specific reverse engineering scenario. Such components can be either technology-specific metamodels or their corresponding model discoverers (cf. Section 3.2.2), as well as related transformations (cf. Section 3.2.3). They are the concrete bricks addressing the different (kinds of) legacy systems to be potentially reverse engineered.

Finally the *Use Cases* layer provides some reuse and integration examples, which are either relatively simple demonstrators or more complete ready-to-use components implementing a given reverse engineering process. Such components are mostly intended to realize the actual integration between components from the two other layers, and can be either reuse as-is or extended/customized for a different scenario.

According to these principles and architecture, the next sections details the two main MDRE phases we propose, namely *Model Discovery* and *Model Understanding*.

3.2.2 Model Discovery

Model Discovery can be defined as the fundamental action of automatically obtaining raw model(s) from the legacy system to be reverse engineered. Note that, even if some models were created during the development process, they could already be outdated compared to its actual state. In that case, model discovery can provide a useful complementary support to validate the relevance of such already existing models.

These models are *initial models* because they have a direct correspondence with the elements of the legacy system: they can be considered as (full or partial) abstract syntax graphs as they do not imply any deeper analysis, interpretation or computation at this stage. To be able to represent the legacy system without losing any information, the metamodels employed at this stage are often metamodels of a low abstraction level and that closely resemble the source technology such as a general programming language (e.g. Java, C#, C++), a DSL (e.g. SQL, HTML), a file format (e.g. Microsoft Word or Excel), etc.

The metamodel-driven software components allowing to generate these raw models are called *model discoverers*, where each discoverer targets a specific technology. The overall principle of a model discoverer is shown on Figure 3.2.

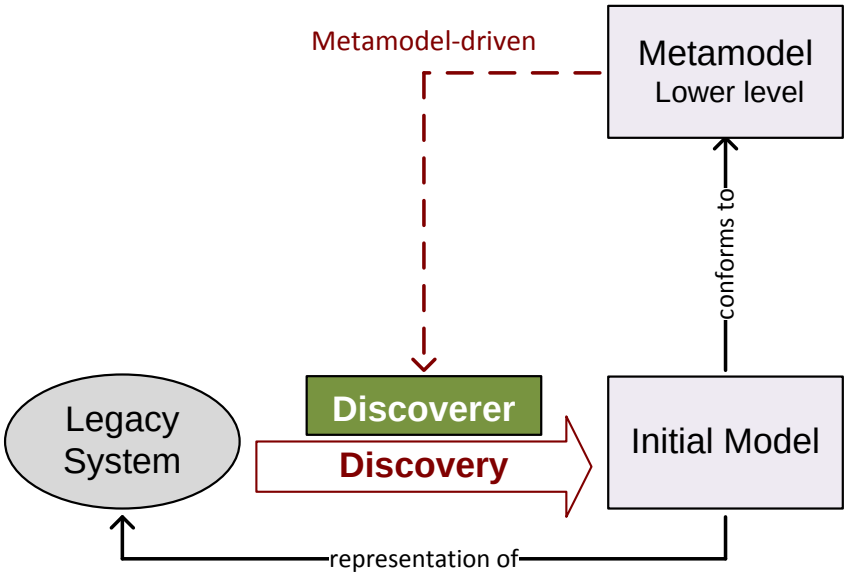

Figure 3.2 – General principle of Model Discovery.

Firstly, the metamodel (defining the perspective the system will be observed from) has to be designed depending on the kind of legacy system that is concerned. For some well-known technologies like Java/JEE, appropriate metamodels can already be available (e.g. cf. the ones included in our MoDisco implementation as presented in Section 3.3). This metamodel is the main factor ensuring the quality and completeness of the discovery phase. This phase is in charge of creating instances of this metamodel from the legacy system. Thus, the model resulting from the execution of the built discoverer will conform to this base metamodel.

In some cases, the discovery process can be split up into two complementary steps as shown in Figure 3.3:

1. **Injection** - focusing on bridging between the technical spaces of the legacy system and the MDE technical space, e.g. using parsers or APIs to access the content of legacy artifact(s) and then create the corresponding model accordingly.
2. **Transformation** - an additional syntactic (or structural) mapping, already inside the MDE technical space, to produce the actual initial model from the result (i.e. a model) of the previous injection step if required.

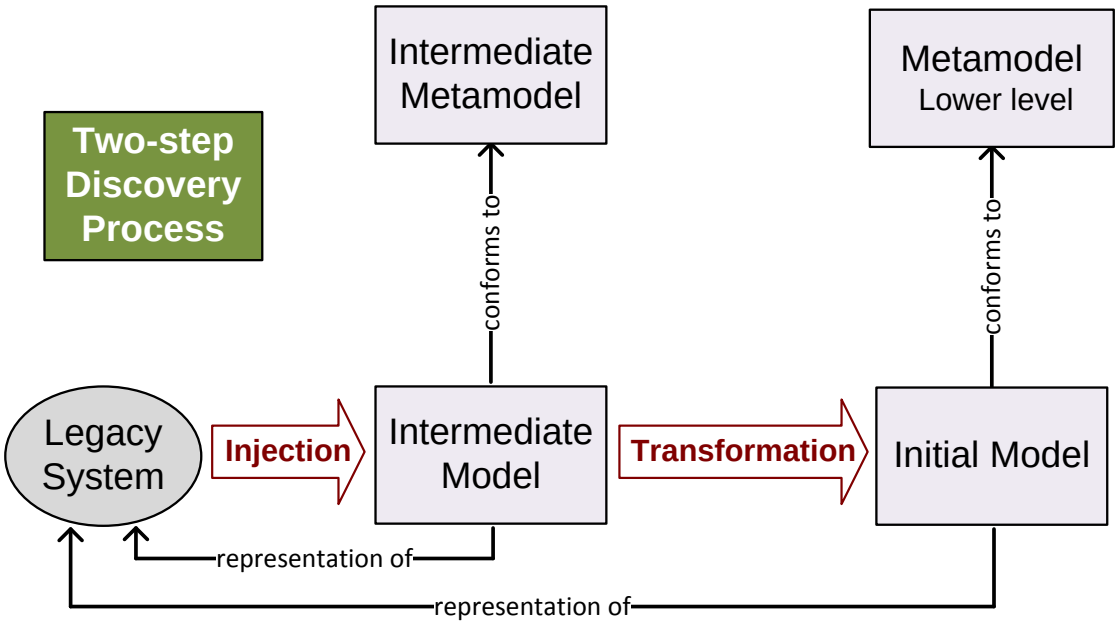

Figure 3.3 – Two-step approach for Model Discovery.

These two steps are not necessarily atomic operations performed at once. They can rather be composed of several sub-operations possibly iterative and interactive, notably chains of model transformations. The length of this chain depends on the complexity of the discovery process. It can also vary according to the availability of some intermediate metamodels that can be used to simplify it.

For instance, considering the case of discovering the model corresponding to a set of source code files in Java, the discovery process implementation can differ:

— The **two-step approach** can be strictly applied: a generic (intermediate) model of the abstract syntax tree can be obtained first from the program (based on a generic metamodel for instance [39]) and then transformed into a language-specific lower-level model (i.e. a Java model).

— A more **direct approach** can be considered: visit the abstract syntax tree of the program (e.g. using a dedicated API or the result of a previous parsing) in order to build directly the language-specific lower-level model (i.e. the Java model), the *transformation* step being directly integrated within the *injection* one.

There are different arguments to help deciding between these two implementation options. In the first case, considering such a two-step process usually allows reducing the complexity of each individual step and thus can facilitate the global discoverer implementation. The first part of the discoverer (the injector) becomes generic and can be directly reused by similar model discoverers. Nevertheless, these benefits can sometimes be not so relevant for the reverse engineering scenarios interesting for the user. In such cases, one can decide to go for the second option which involves less artifacts to think about

and offers better performance. Anyway, the *initial models* obtained at the end of both alternatives have to represent the same system at the same level of details.

3.2.3 Model Understanding

The *initial models* discovered in the previous phase can be exploited in different ways in order to obtain the final expected representations of the reverse engineered system. Thus, the *Model Understanding* phase as described in Figure 3.4 is mainly transformation-based: it largely relies on (chains of) model transformations to perform semantic mappings (in the data sense) that generate a set of *derived models*. This is realized according to the information/structure expected by the reverse engineer from the initial models obtained in the previous step. The outputs can be the *derived models* themselves or the result of extracting these models into some external tools (e.g. for specific purposes such as visualization, cf. Section 3.1.2).

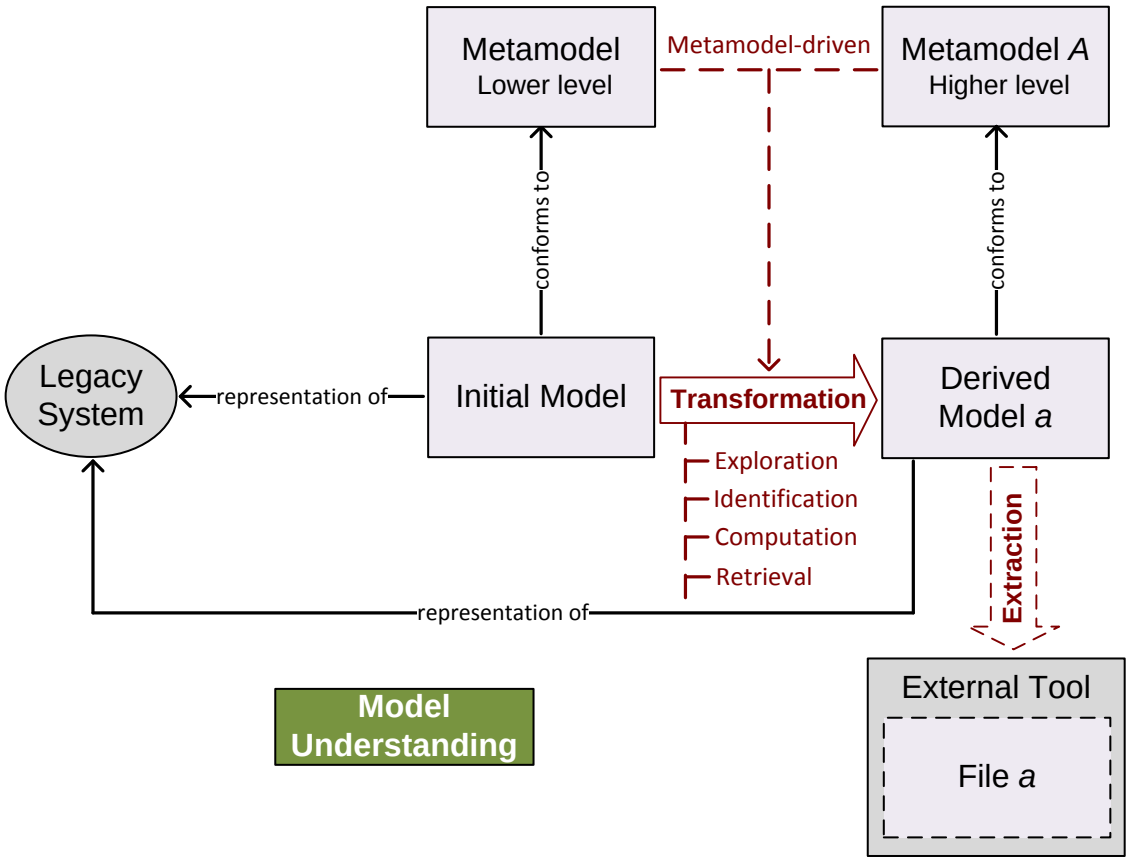

Figure 3.4 – General principle of Model Understanding.

Several *derived models*, temporary or final, can be obtained from the same *initial models*. This is done by using different automated (chains of) model-to-model transformations depending on the goal of the reverse engineering process. It is an important benefit of such a two-phase approach, as reusing the discovery phase facilitates a lot the exploitation and analysis of the legacy system. Each one of these derived models conforms to a metamodel of (usually) a higher abstraction level. This metamodel is tailored to the required representation of the system or to the targeted external tool. Note that an exception would be a re-engineering scenario where the goal is to re-implement the

system using a new technology. In this case, lower level models (representing the system in this new technology) are required for a final model-to-text transformation to actually generate the corresponding source code.

As shown on Figure 3.4, a Model Understanding process generally includes the following main actions, all performed by/within **model transformations** and usually executed in an iterative process that refines the results until the desired derived models are obtained:

1. Legacy system **exploration** via its *initial models* (**model navigation**).
2. Required information **identification** via these models (**model querying**).
3. Derived model **computation** using the identified information as source (**model computation**).
4. Representation **retrieval** into *derived models* (**model building**).

The legacy system exploration is performed on the *initial models* of that system rather than directly on it. Thus, extended model navigation capabilities (e.g. the ones provided in model transformation via dedicated languages such as OCL for instance, cf. Section 2.2.2), are required to browse them efficiently. This implies notably navigating inside these models at all detail levels and so returning as a result different sets of model elements, structural features (i.e. attributes and references), annotations, etc. These results are then queried, often iteratively, in order to select/filter and obtain only the information strictly needed. This information is used in order to perform the actual computation of the expected derived models over the system represented by the initial models.

As said before, all these steps can be implemented by model transformations specifying the corresponding refinements. As introduced in Section 2.1.2, such transformations can be defined using different languages that can be declarative (e.g. QVT [161]), imperative (e.g. Java, Kermeta [115]) or hybrid (e.g. ATL [116]). Basically, they take as input(s) the navigated/queried model(s) and generate as output(s) the computed model(s) that can conform to the same metamodel, its augmented or reduced version, or a totally different one.

The target models (or representations) are built as the final results from the various obtained *derived models* (i.e. refinements). Thanks again to other model transformations (e.g. targeting visualization formats such as SVG or DOT) but also sometimes to external tools (e.g. being able to render models graphically), these last refinements are actually used to retrieve the expected representations of the initial legacy system.

As an example, Figure 3.5 shows a possible reverse engineering scenario from Java source code. The *initial model* is the complete Java model previously discovered from a given Java EE application. This model can be then used to generate different representations of the system, like the dependency graph of the Java classes according to the internal method calls or a set of metrics on the complexity of the code such as the number of classes per package, the average number of methods per class, etc. These metrics can be finally sent to an external reporting tool able to provide analytical graphical visualizations for the provided data.

3.2.4 Main Benefits

A clear separation of concerns in our approach has been combined with the generalized use of MDE during the two consecutive phases of a full MDRE process: Model Dis-

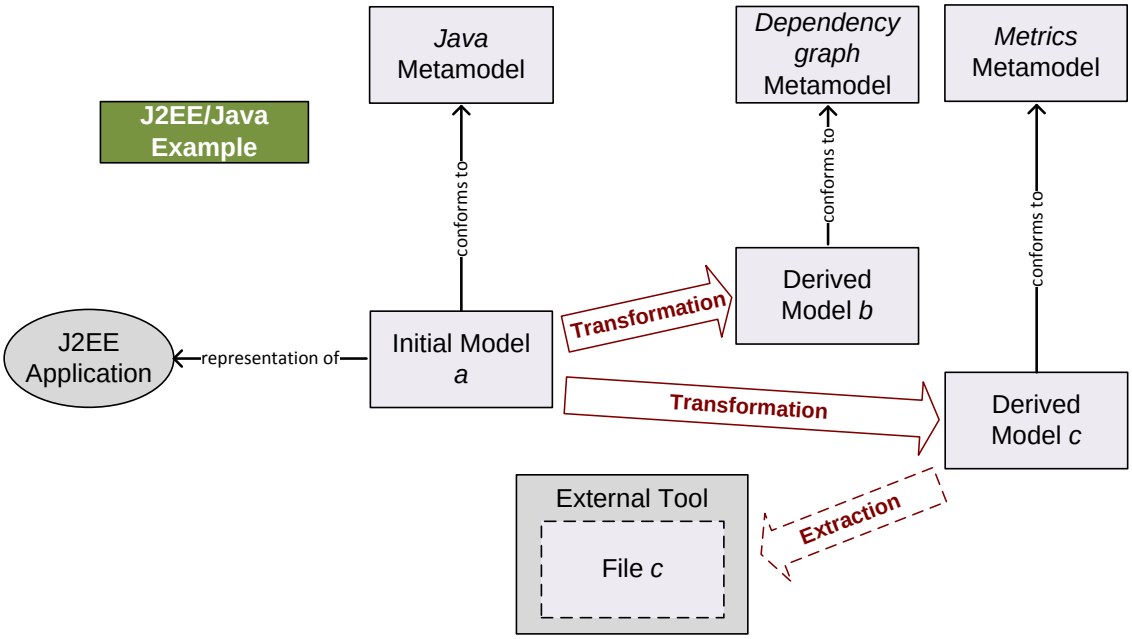

Figure 3.5 – A J2EE/Java example of a Model Understanding phase.

covery and Model Understanding. This allows answering to the four MDRE challenges previously identified in Section 3.1.4. For each one of them, next paragraphs respectively explain how.

Firstly, the explicit distinction in the proposed architecture between technology and scenario-independent (*Infrastructure* layer), technology-specific (*Technologies* layer) and scenario-specific (*Use Cases* layer) components can provide a high adaptability at two different levels:

1. The nature of the legacy system technology.
2. The kind of reverse engineering scenario.

Secondly, the metamodel-driven approach followed in our proposition can enable covering different levels of abstraction and satisfying several degrees of detail depending on the needs of the reverse engineering scenario. All the required information can be actually represented as models so that there is no information loss during the MDRE processes. Only those details that the user explicitly wants to left out, as part of a transformation process of the initial discovered models, can be ignored.

Thirdly, the use of MDE techniques can allow the decomposition and automation of the reverse engineering processes. They can be divided in smaller steps focusing on specific tasks, and be largely automated thanks to the appropriate chaining of corresponding MDE operations (notably model transformations). They can also directly benefit from the model exploration and extraction capabilities provided by these MDE techniques in order to improve legacy systems overall comprehension. All the involved modeling artifacts (models, metamodels, transformations, etc.) can be homogeneously re-used, modified for maintenance and evolution reasons, or extended for other purposes. Moreover, new transformations can be developed and plugged adding more capabilities without altering the already implemented features.

Finally, the treatment of the potentially huge amount of concerned data can be simplified. This is because the models of the systems are the elements actually processed

(thanks to the available modeling techniques) rather than directly the systems themselves (which are not modified during the process). We have observed the performance of key components of our MoDisco implementation of the approach we propose, as described in next Section 3.3. We have been able to conclude that they are already acceptable for an industrial use in several concrete scenarios (cf. Section 3.4.2 for instance).

3.3 The MoDisco framework

This section describes the MoDisco framework implementing the generic and extensible MDRE approach described in previous Section 3.2. The goal of MoDisco is to facilitate the development of model-based and model-driven solutions targeting different reverse engineering scenarios and legacy technologies. In what follows, we start by providing an overview of the MoDisco project (in Section 3.3.1). Then, we give more insights on the content of the MoDisco infrastructure and technology layers as supporting both the Model Discovery and Model Understanding phases (in Section 3.3.2 and Section 3.3.3 respectively). Finally, we briefly describe the MoDisco use case layer (in Section 3.3.4) before we end with base indications on how to extend the framework (in Section 3.3.5).

3.3.1 Project Overview

MoDisco is an open source project which is officially part of the Eclipse Foundation [71] and is integrated in the Modeling top-level project promoting MDE techniques and their development within the Eclipse community (cf. Section 2.3). Several years ago, it was officially recognized by the OMG as providing relevant implementations for several of the ADM task force industry standards [154]:

— Knowledge Discovery Metamodel (KDM) [156], a metamodel offering a common intermediate representation for modeling existing software systems and their environments independently from any particular technological platform.

— Structured Metrics Metamodel (SMM) [164], a metamodel offering an extensible representation for modeling (and exchanging) any kind of measurement information related to software systems, their design and operation.

— Abstract Syntax Tree Metamodel (ASTM) [153], a metamodel offering a common representation for modeling abstract syntax trees of programs implemented in various different languages (imperative, object-oriented, etc.).

Initially created as an experimental research framework, it has now evolved into an industrialized project thanks to a long-term collaboration with the MIA-Software company (Sodifrance Group) [190] (which is still maintaining the official project releases, cf. next paragraph). It is important to note that most of the components presented in this section have been developed by members of this company based on the research work, approach and architecture we proposed. These components are in a stable state since several years already: they are now mostly maintained (e.g. bug fixing, documentation updating) and no totally new features have been recently added to the project.

Indeed, the MoDisco framework has been integrated in the Eclipse Simultaneous Releases since several years (including the latest Mars, Neon, Oxygen and Photon ones).

This year again, it will be part of the coming Eclipse Simultaneous Release (to be released by June 2019) along with other Eclipse Modeling projects. These releases provide several ready-to-use Eclipse bundles targeting different families of user, and notably the MDE engineers via the *Eclipse Modeling Tools* bundle MoDisco is part of. MoDisco is structurally delivered as a set of Eclipse features and related plug-ins whose builds are directly downloadable via the MoDisco update sites. Each MoDisco component is actually composed of one or more plug-ins and can be classified according to the previously described three-layer architecture (cf. Section 3.2.1).

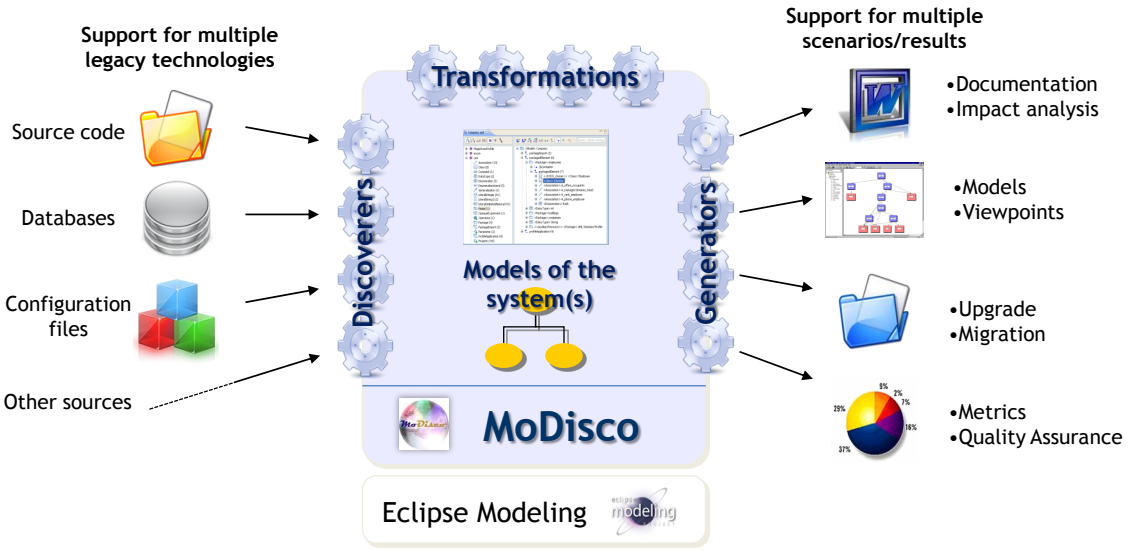

Figure 3.6 – Overview of the Eclipse MoDisco project.

As shown on Figure 3.6, from all kinds of possible legacy artifacts (e.g. source code, databases, configuration files, documentation, etc.) MoDisco aims at providing the required capabilities for creating the corresponding model representations and allowing their handling, analysis and use. Relying on EMP and notably on EMF (cf. Section 2.3), it offers various kinds of components such as discoverers, transformations, generators, etc. to implement this support. As output, the framework targets the production of different artifacts on/from the considered legacy systems, depending on the expected usage of the reverse engineering results (e.g. software modernization, refactoring, retro-documentation, quality analysis, etc.). One of the main goals of MoDisco is to remain adaptable to many different scenarios, thus facilitating its adoption by a potentially larger user base.

The overall architecture of MoDisco is summarized in Figure 3.7, and detailed further in the next subsections. Together with generic components (i.e. the infrastructure) allowing to create dedicated MDRE solutions, MoDisco also provides predefined technology-specific components that allow users to directly target some types of legacy artifacts and use cases. It is worth to note that some MoDisco components have been deemed useful beyond a strictly reverse engineering context and are now being externalized to facilitate their reuse in other projects, e.g. EMF Facet [67] as explained in Section 3.3.2. Apart from these components, the MoDisco project is also equipped with all the standard Eclipse tooling to support its development and the relation with its user community.

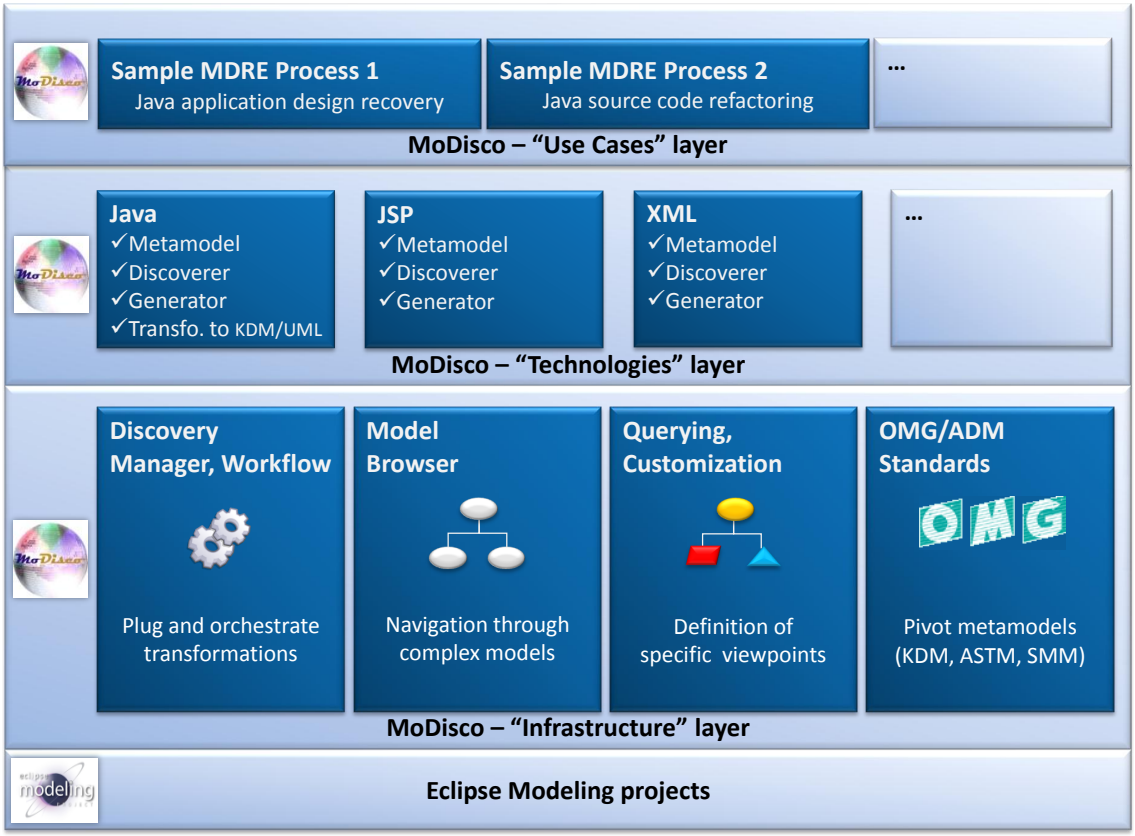

Figure 3.7 – Architecture of the MoDisco framework.

3.3.2 Infrastructure Layer

As part of its infrastructure layer, MoDisco currently provides a set of generic components that are relevant independently from the concerned legacy technologies or reverse engineering scenarios.

OMG ADM Standard Implementations

From a standardization perspective, concrete implementations of three OMG ADM standard metamodels are available, namely KDM, SMM and ASTM as introduced in Section 3.3.1. KDM allows representing the entire software system and all its entities at both structural and behavioral levels. It deals with the legacy system metadata, artifacts and higher level structure of the code in a generic way. As for ASTM, it focuses more on the lower level abstract syntax tree of the sources (independently from the used language). SMM is used to both specify any kind of measure/metric on legacy software and express the obtained results (measurements). All of them come with an EMF Ecore version of the metamodel in addition to the generated model handling API.

Moreover, there is a more advanced support for KDM. A corresponding model discoverer, using a model transformation to KDM as explained in subsection 3.3.3, allows the automated analysis and representation of the file hierarchy of applications as so-called KDM *Source* models (using a subset of KDM). A predefined transformation also allows obtaining UML models (class diagrams in that case) from KDM *Code* models (using

another subset of KDM). Moreover, a small framework has been developed based on the KDM metamodel to facilitate the future building of new model discoverers mixing both physical resource metadata (KDM models) and code content information (e.g. Java models).

Discovery Manager and Workflow

To globally manage all the model discoverers registered within the MoDisco environment, a Discovery Manager is provided. It comes along with the simple generic interface a discoverer should implement in order to be plugged into the framework (plus the corresponding extension point). It also comes with a Discoverers View providing a quick Eclipse view over all registered discoverers. In addition, the MoDisco Workflow enables the chaining and launching of a set of registered discoverers, transformations, scripts, etc. as part of larger MDRE processes (cf. Section 3.3.4).

Model Browser and Navigation

One of the most powerful MoDisco component is the Model Browser. It has been designed to make the navigation through complex models much easier by providing advanced features such as full exploration of all model elements, infinite navigation (as being a graph, the model is fully navigable and not restricted by a tree representation), filtering, sorting, searching, etc. It is mainly composed of two panels: the left panel is displaying the possible element types (i.e. the concepts from the concerned metamodel) while the right panel is showing the model elements themselves. It is also completely metamodel-independent and customizable (via the definition of specific customization models). For instance the icons and global formatting of the displayed information can be specialized for a given metamodel, as shown with UML in Figure 3.8.

Querying and Customization

MoDisco is also equipped with model querying capabilities that are particularly relevant when elaborating on MDRE processes. Thus, the component named Query Manager allows registering, gathering and executing model queries over any kind of model. A dedicated metamodel has been designed, along with the corresponding tooling, for the users to be able to describe their query sets as models. The key point is that the offered mechanism is fully query language-independent: queries can currently be written in Java or OCL and new drivers for other languages can be added in the future (cf. section 3.3.5).

As also part of this *Infrastructure* layer and complementary to these model querying facilities, MoDisco integrates the use of dynamic metamodel extension capabilities. This is particularly relevant during the Model Understanding phase (cf; Section 3.2.3) when aiming to retrieve and represent useful extra-information on discovered models. These extensions, called *facets*, are computed at runtime and allow adding useful information to already existing models without actually modifying them. Initially developed within MoDisco, the Facet mechanism and tooling have been externalized and are now provided by the dedicated EMF Facet project [67].

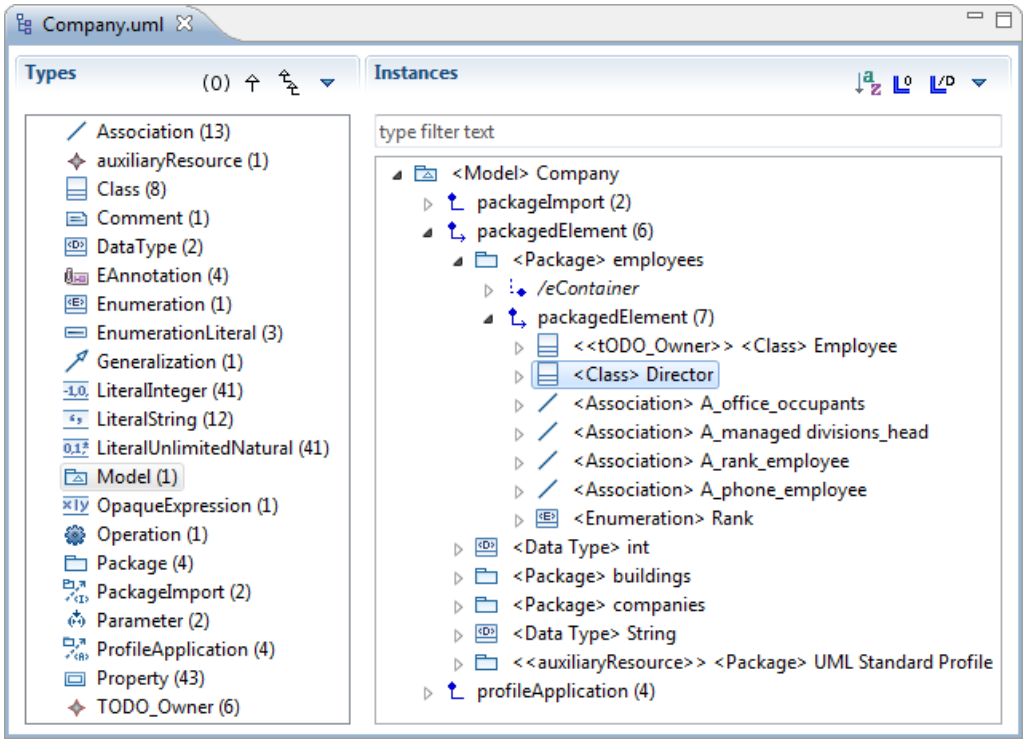

Figure 3.8 – The generic MoDisco Model Browser, customized for the UML metamodel.

3.3.3 Technology Layer

As part of its legacy technology dedicated support, MoDisco already provides a set of useful deployable components. These components can be directly combined (by exchanging models via the Discovery Manager and Workflow for instance, cf. Section 3.3.2) with the ones from the infrastructure in order to address concrete reverse engineering scenarios. However, they are themselves independent from any specific use case. MoDisco currently offers ready-to-use support for three different legacy technologies, namely Java, Java Platform, Enterprise Edition (JavaEE) (including JavaServer Pages (JSP)) and XML. This list could be extended in the future by new research or industrial contributions covering other languages (e.g. C#) or technologies (e.g. databases).

The Java dedicated features show the capabilities of the generic MoDisco framework by applying them in the context of a widely used technology such as Java. A complete Java metamodel based on the JDT [62] is offered by the framework. It covers the full abstract syntax tree of Java programs from the package and class declarations to the method bodies, expressions and statements (that are also modeled in detail). Relying on this metamodel (cf. some examples of the metamodel concepts in the left panel of the model browser on Figure 3.9), a corresponding discoverer is available allowing to automatically obtain complete Java models out of any Java projects (cf. also Figure 3.9). As previously introduced in Section 3.2.2 when describing the *injection* step, the Java model discoverer uses a dedicated technology, JDT and more particularly its in-memory representation of the Java sources, as the technical solution to navigate the program abstract syntax tree (an associated visitor implementing the actual building of the Java model). The automated regeneration of the Java source code from these handled (and in the meantime possibly modified) Java models is also ensured thanks to a specific code generator im-

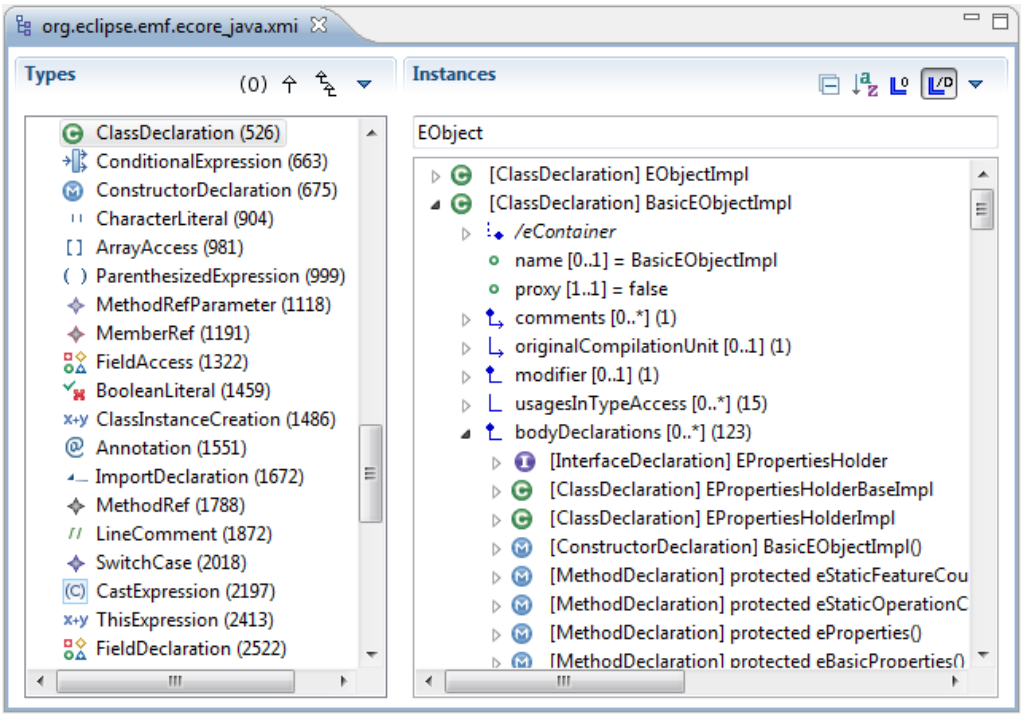

Figure 3.9 – Example of a discovered Java model opened in the MoDisco Model Browser.

plemented with a model-to-text dedicated technology (Acceleo [58]). A complementary model transformation allows the KDM discoverer directly producing generic KDM *Code* models (cf. Section 3.3.2) from such Java models. Still related to KDM, an additional discoverer is provided to automatically get composite (trace) models integrating both the Java and KDM "Source" elements (cf. Section 3.3.2) from any Java project.

Given the widespread use of XML documents in (legacy) software systems, a full and generic XML support is natively provided by MoDisco. A complete XML metamodel has been implemented conforming to the subset of the related World Wide Web Consortium (W3C) specification defining the XML core concepts: root, elements, attributes, etc. Thus, this metamodel is schema-independent and can be used to model any XML file, i.e. both XML documents and XML schema definitions. To concretely allow this, the associated model discoverer is made available. This notably prevents from having to implement a particular discoverer for each XML-based file type, thus saving some useful effort. If really needed, an additional model transformation can still be built quite systematically from this generic XML metamodel to a given specific metamodel (cf. the Model Discovery two-step approach as introduced in Section 3.2.2).

A specific support for JavaEE technologies has also been developed. This includes notably a metamodel for the JSP language (extending some core concepts from the XML one) as well as the corresponding discoverer allowing to get complete JSP models out of JSP source code. In addition to this JSP support, the JavaEE dedicated components cover the automated modeling of the most common JavaEE Web application configuration files (namely *web.xml* and *ejb-jar.xml*) via specific model discoverers. Finally, there is a predefined set of ready-to-use queries and facets for either highlighting existing JavaEE-specific information or extracting new JavaEE-related data from previously discovered Java models.

3.3.4 Use Cases Layer

Complementary to all these components and to illustrate their actual use in MDRE processes, the MoDisco *Use Cases* layer offers some more features for specific reverse engineering scenarios.

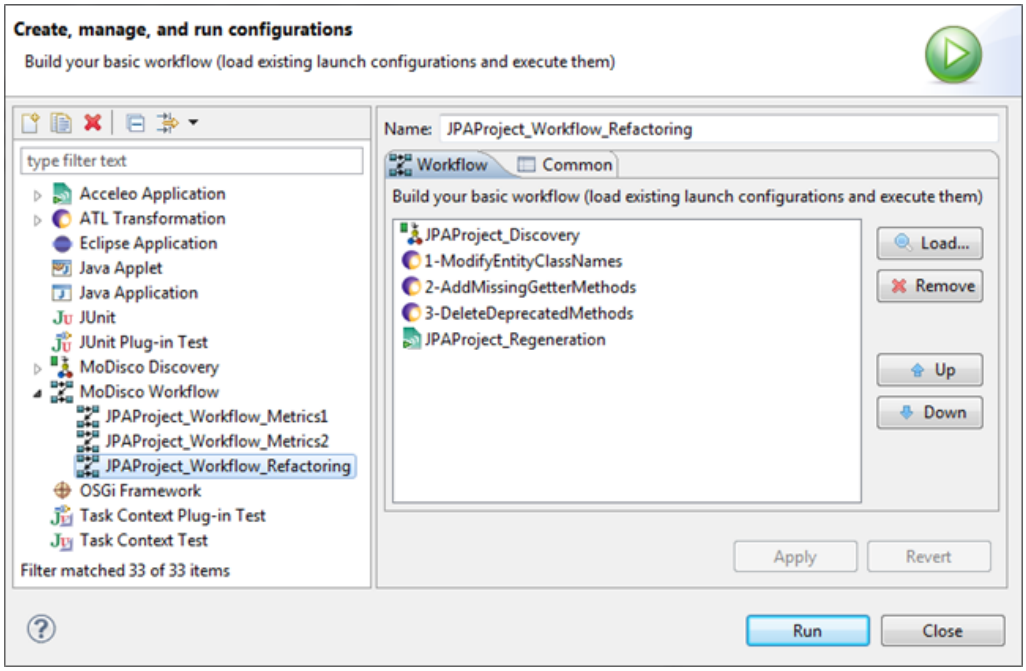

Figure 3.10 – The generic MoDisco Workflow, used for a sample Java refactoring process.

MDRE processes have the natural capability to be largely automated. Thus, the different MoDisco components can be combined together to address particular reverse engineering use cases. As mentioned before in Section 3.3.2, MoDisco offers a dedicated workflow support specifically intended to the chaining of model discoveries with consecutive model transformations, related scripts, final code generations, other programs, etc. This comes with a dedicated window to select and order the operations (calling to MoDisco components or others) to be part of the given MDRE workflow.

As concrete examples, complete automated MDRE workflows are provided for instance to recover the design of Java applications or to perform some refactoring on Java code. The example in Figure 3.10 shows such a workflow composed of an initial Java model discover, three consecutive refining (model) transformations and a final Java code (re-)generation.

3.3.5 Extending MoDisco

MoDisco has been designed as a generic and extensible framework. There are different ways of extending it in order to allow a more in-depth or specific support to some MDRE scenarios or legacy technologies. Notably, as an Eclipse-based solution, the generic components come with extension points [217] and related interfaces that can be used to implement these more particular features.

The most straight-forward manner is to simply complement the *Use Cases* layer (cf.

Section 3.3.4) with other examples of MoDisco component combinations (e.g. defining different workflows chaining these components) to address a given MDRE process.

Another direct way is to complement the *Technologies* layer (cf. Section 3.3.3) by adding the support for another (legacy) language, framework, file format, etc. This implies first developing the corresponding metamodels (in EMF Ecore [63]), model discoverers, browser customizations, associated queries (e.g. in Java or OCL [160]) or transformations (e.g. in ATL [59]), generators (e.g. in Acceleo [58]), etc. For instance, for new model discoverers, a generic interface (*IDiscoverer*) has to be implemented and a specific extension point (*org.eclipse.modisco.infra.discovery.core.discoverer*) used so that the Infrastructure (the Discovery Manager in the present case) can then automatically identify and declare the component as part of the MoDisco environment. After that, such a newly added discoverer can be considered identically to other similar components provided with the base version of the framework.

The other way is to directly work on the *Infrastructure* level (cf. Section 3.3.2). Apart from the addition of a new implementation of a standard metamodel, this is generally more complex and requires a deeper knowledge of the MoDisco internals and generic component APIs. For instance, advanced browser customizations (e.g. add new options, contextual actions or a different viewer) have to be implemented using the Model Browser specific API. Moreover, in addition to Java or OCL as already offered, new languages support can also be developed and plugged into the Query Manager. Another example of infrastructure extension is the possibility of supporting different workflow engines (than the provided base one) via the use of a dedicated extension point.

For getting more insights on all these technical aspects, please refer to the MoDisco Developer Documentation as available with any official Eclipse MoDisco release [70].

3.4 Evaluation

This section explains how we have evaluated the proposed conceptual approach and its implementing MoDisco framework. In order to provide a relevant evaluation covering some of the general challenges introduced in Section 3.1.4, this has been made using both qualitative and quantitative ways. In Section 3.4.1, we start by summarizing the research questions / challenges we intended to evaluate. In Section 3.4.2, we describe two real use cases in which MoDisco has been practically applied. In Section 3.4.3, we also present additional performance benchmarks addressing more specifically the scalability aspects.

3.4.1 Research Questions (RQs)

The evaluation described in what follows in this section was performed to both qualitatively and quantitatively assess the relevance and usefulness of our approach when applied to real-world scenarios. More specifically, we aimed to answer the following research questions and underlying MDRE challenges (cf. Section 3.1.4):

1. *RQ1 - Technical heterogeneity*. Are we able to deploy the MoDisco conceptual approach and/or its implementing components in practical scenarios of various

technical natures (e.g. as far as the input legacy system is concerned)? To evaluate this, we have been able to work on two practical use cases from our partner company Mia-Software (cf. Section 3.4.2).

2. *RQ2 - Adaptability and portability*. Are we able to adapt and reuse the MoDisco conceptual approach and/or its implementing components as part of other (MDRE) solutions? To evaluate this, we have been able to work on a particular integration use case from our partner company Mia-Software (cf. Section 3.4.2).
3. *RQ3 - Scalability*. Are we able to use the MoDisco conceptual approach and/or its implementing components in the context of large-scale scenarios? To evaluate this, we have worked on dedicated scalability benchmarks intending to measure different scalability aspects of our solution (cf. Section 3.4.3).
4. *RQ4 - Structural and behavioral complexity*. Are we able to use the MoDisco conceptual approach and/or its implementing components in order to deal with all levels of structural and behavioral complexity in legacy systems? To evaluate this, we can consider the two practical and quite complex use cases from our partner company Mia-Software (cf. Section 3.4.2). However, this evaluation is currently limited as far as structural complexity is concerned and not really relevant as far as behavioral complexity is concerned.

Of course the presented evaluation could still be extended in the context of future work, e.g. to cover (many) more different MDRE scenarios as well as to consider even (much) larger legacy systems as inputs. Nevertheless, we believe it already allows providing interesting insights on the actual capabilities of our approach and its current implementation in a real-world context.

3.4.2 MDRE Concrete Use Cases

Industrial reverse engineering scenarios typically involve different types of input artifacts like source code, documentation or raw data. They also involve different kinds of resulting outputs such as new/modified source code, documentation, models, metrics, etc. . The proposed approach and MoDisco framework have already been applied and deployed in several of such real use cases with the underlying objectives of:

1. Ensuring its actual usability in various technical scenarios (cf. RQ1 in Section 3.4.1).
2. Collect feedback from developers/users integrating MoDisco into another solution (cf. RQ2 in Section 3.4.1).
3. Validate its capacity to manage the complexity of some particular scenarios (cf. RQ4 in Section 3.4.1).

The various MoDisco components can be used by combining, chaining or integrating one or more of them into the MDRE solutions to be built. Thanks to the characteristics of the used EPL license [66], in all cases the elaborated solutions can be fully open source, fully proprietary or follow an hybrid approach. To illustrate qualitatively the suitability of MoDisco in industrial scenarios, this section presents a couple of projects where our partner company Mia-Software built a MoDisco-based solution to address real customer problems.

Use Case 1: Java Application Refactoring

MoDisco has been used to handle a critical project concerning a legacy system of a big car rental company. The idea was to automate massive refactoring tasks on a large Java application (approximately 1000K Lines of Code (LOC)) in order to improve both its performances (notably in terms of memory usage) and code readability.

Process Overview. To be able to refactor the Java legacy system, specific patterns first needed to be identified in its source code. As a consequence, an initial reverse engineering phase was required in order to obtain an exploitable model of this system. Based on the discovered representation, the code upgrades have then been automatically performed at the model level (cf. Figure 3.11) by means of in-place model transformations. As a last step, the (upgraded) source code of the refactored Java application has been automatically generated from the modified models.

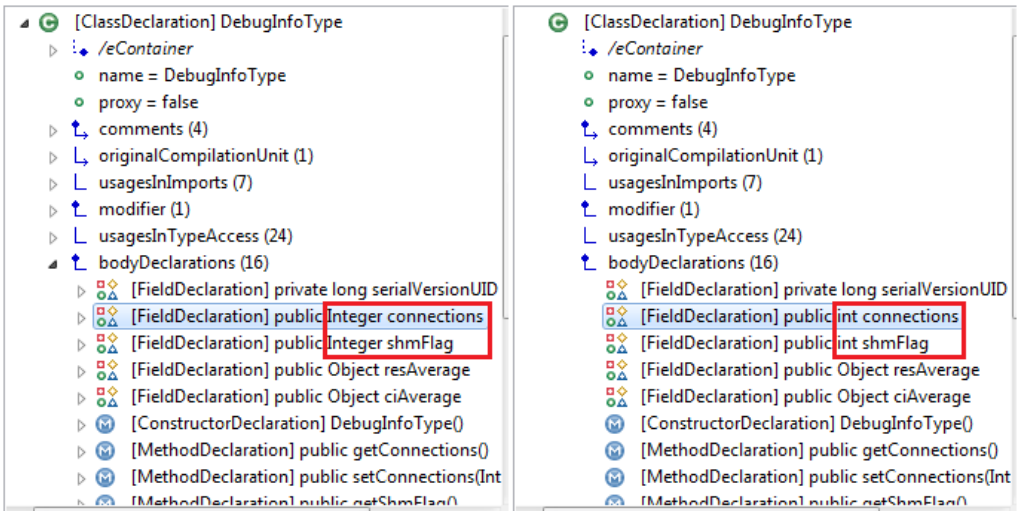

Figure 3.11 – Java model before (left) and after (right) refactoring, using the MoDisco Model Browser to show the effects of the first refactoring.

To perform the refactoring, three model transformations were used with the aim to:

— Replace the use of Java wrapping types (Integer, Double, Float, Long) by Java primitive types (int, double, float, long) in some identified parts of the code, as visible on Figure 3.11 (cf. also Figure 3.12 for an excerpt of the corresponding transformation rule).

— Clean the useless code calling to *log* (based on the *Apache Commons Logging* framework).

— Delete some abusive uses of a specific client class (named *ValueHolder*, that was simulating some kind of C++ like pointers).

Note that these *refactoring* rules had to be applied on each of the different application releases.

This **model driven refactoring process** is an adaptation of the well-known Horseshoe model. Figure 3.13 depicts its three main steps:

1. **Reverse engineering** from the input Java application by using the MoDisco Java metamodel and corresponding model discoverer (MoDisco *Technology* layer)).

```
private void replaceBoxedFieldByPrimitiveType() {
    for (FieldDeclaration fieldDeclaration : allFieldDeclarations()) {
        if (isFormCode(fieldDeclaration) || isExcludedFromPrimitiveRules(fieldDeclaration)) {
            continue;
        }

        TypeAccess replacementForBoxedPrimitiveType = replacementForBoxedPrimitiveType(
                                                        fieldDeclaration.getType());
        if (replacementForBoxedPrimitiveType != null) {
            System.out.println("field declaration : replacing "
                    + fieldDeclaration.getType().getType().getName()
                    + " by "
                    + replacementForBoxedPrimitiveType.getType().getName()
                    + " in " + locate(fieldDeclaration));
            fieldDeclaration.setType(replacementForBoxedPrimitiveType);
            this.count2b++;
        }
    }
}
```

Figure 3.12 – An example of Java application refactoring rule for type replacement.

2. **Restructuring** of the obtained application model thanks to model transformations in Java implementing the previously described rules (that could be positioned as part of the MoDisco *Use Cases* layer).
3. **Forward engineering** of the output (refactored) Java application from this modified model by running the MoDisco Java generator (MoDisco *Technology* layer)).

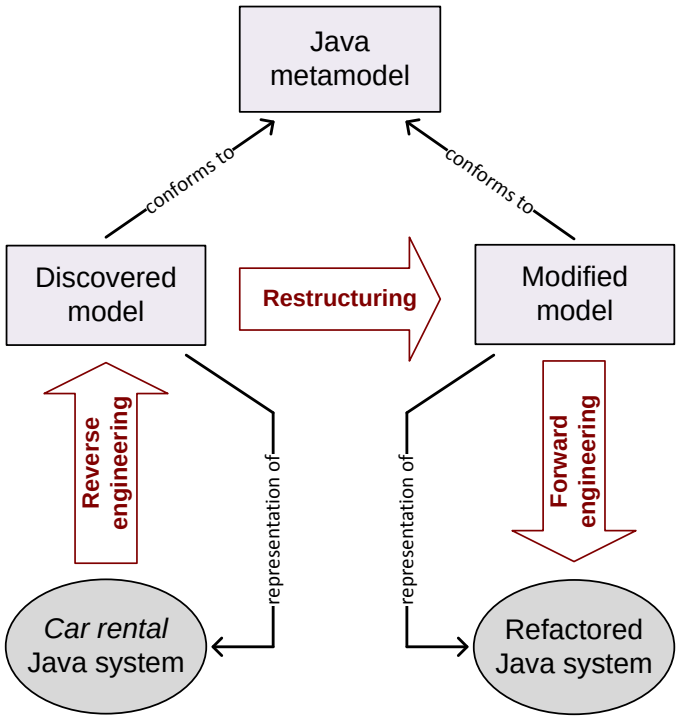

Figure 3.13 – Overall process of the model driven Java application refactoring.

The choice of Java for implementing the model transformations (rather than using a dedicated model transformation language like ATL) has been made by Mia-Software because of the Java expertise of their engineers. The MoDisco Model Browser (and Java metamodel specific customization) has been used for manual verification and testing at each step of the process and on of the several application model versions. The overall

integration of the three different steps has been realized thanks to the provided MoDisco workflow facilities.

Observations. Globally, the use of MoDisco in the context of this refactoring use case has been a success. As a result of the process, approximately 60K LOC from the whole application were concerned and so automatically refactored (Figure 3.14). An effective performance gain on the regenerated application has been observed by Mia-Software. They also noticed an overall readability improvement of the modified source code parts.

```
private static final long serialVersionUID = 1L;
public Integer connections;
public Integer shmFlag;
public Object resAverage;
public Object ciAverage;

//MIA tag : Start : user attributes of "DebugInfoType"
//MIA tag : End : user attributes of "DebugInfoType"
//Begin : Constructors of Class "DebugInfoType"
/**
* Builder DebugInfoType
*/
public DebugInfoType() {
    //MIA tag : Start : user modified constructor "DebugInfoType"
    //MIA tag : End : user modified constructor "DebugInfoType"
    connections = 0;
    shmFlag = 0;
    resAverage = 0;
    ciAverage = 0;
}
```

```
private static final long serialVersionUID = 1L;
public int connections;
public int shmFlag;
public Object resAverage;
public Object ciAverage;

//MIA tag : Start : user attributes of "DebugInfoType"
//MIA tag : End : user attributes of "DebugInfoType"
//Begin : Constructors of Class "DebugInfoType"
/**
 * Builder DebugInfoType
 */
public DebugInfoType() {
    //MIA tag : Start : user modified constructor "DebugInfoType"
    //MIA tag : End : user modified constructor "DebugInfoType"
    resAverage = 0;
    ciAverage = 0;
}
```

Figure 3.14 – Sample Java code before (top) and after (bottom) refactoring.

According to the Mia-Software engineers working on this project, discovering intermediate model-based representations of the source code really improved their comprehension of the application and also largely facilitated the elaboration of the transformations implementing the different modifications. Indeed, only 1 person/month has been required for realizing the full project, i.e. internally developing, then deploying and finally applying the MoDisco-based solution. In contrast, Mia-Software evaluated that following a semi-manual approach based on textual regular expressions (which is for them less reliable than a model-based approach) would have been more costly. It would have required at least 1 person/month simply for realizing a study phase to identify the concerned parts of the application as well as needed expressions, and then again more to set up and perform the actual refactorings themselves. According to their effort calculation scheme (study phase + 500 LOC/day/person for a manual processing), following a fully manual approach could have cost up to 7 person/months for the same project.

Due to the relatively important size of the targeted legacy system, the main problem

encountered during the process was scalability-related. This was linked to the use of the single EMF framework without other complementary (scalability) solutions at the time. A specific parameterization of the model discovery to filter out some Java packages that were not involved in the refactoring (via the corresponding Java model discoverer parameters), as well as the split of the loaded model into several derived models, helped solving this issue.

To summarize, using available MoDisco to implement the MDRE solution to this use case has allowed the company saving time and resources (and therefore reducing the project costs) for the following main reasons:

— The Java metamodel, corresponding injector and extractor were already provided for free and directly reusable. This permitted really focusing on the transformation part which is the core part in a refactoring process.
— The comprehension of the handled application and writing of the model transformation rules (in Java in that case) have been practically facilitated and accelerated by both the well-structured Java metamodel and the model navigation capabilities the MoDisco Model Browser is offering.
— The general automation of the solution (including the developed transformations) has been made easier thanks to the MoDisco integrated framework and corresponding MDRE workflow support.

Use Case 2: Code Quality Evaluation

The use of MoDisco has been integrated in the Mia-Quality solution [144] which is dedicated to the *quality* monitoring of existing applications. This software system has already been deployed several times, e.g. to verify the main product of an insurance management software provider.

Process Overview. To efficiently evaluate the quality of a given legacy application, an automated quality analysis process has to be put in place. However, all the (often heterogeneous) technologies combined in the input system have to be covered to obtain relevant results. This means that the quality solution to be designed and implemented has to be completely technology-agnostic, and to potentially deal with any kind of quality measures/metrics.

Thus, the solution needs to be parameterizable by both the measurements to be actually performed and the software to be monitored. Mia-Quality is a commercial product offering such a generic solution benefiting from the reuse and integration of several MoDisco components.

This **model driven quality evaluation process** is composed of three main steps (Figure 3.15):

1. **Measurement** by analyzing models of reverse engineered applications. These models can be potentially obtained thanks to MoDisco model discoverers (for Java, JavaEE, XML) and related transformations for instance, or via provided import capabilities compatible with other existing tools such as Checkstyle [42].
2. **Consolidation** by connecting external quality analysis tools and aggregating the data coming from these different tools in a unique quality measurement model.

This integration is realized by using a copy of the MoDisco SMM metamodel, as the pivot metamodel in the quality solution, with related model transformations.

3. **Presentation** by displaying the quality analysis results in the specified formats according to user requirements. For instance, the external tool Sonar [192] has been plugged to the solution and customized for showing these results (i.e. the measurements model) in a graphical way.

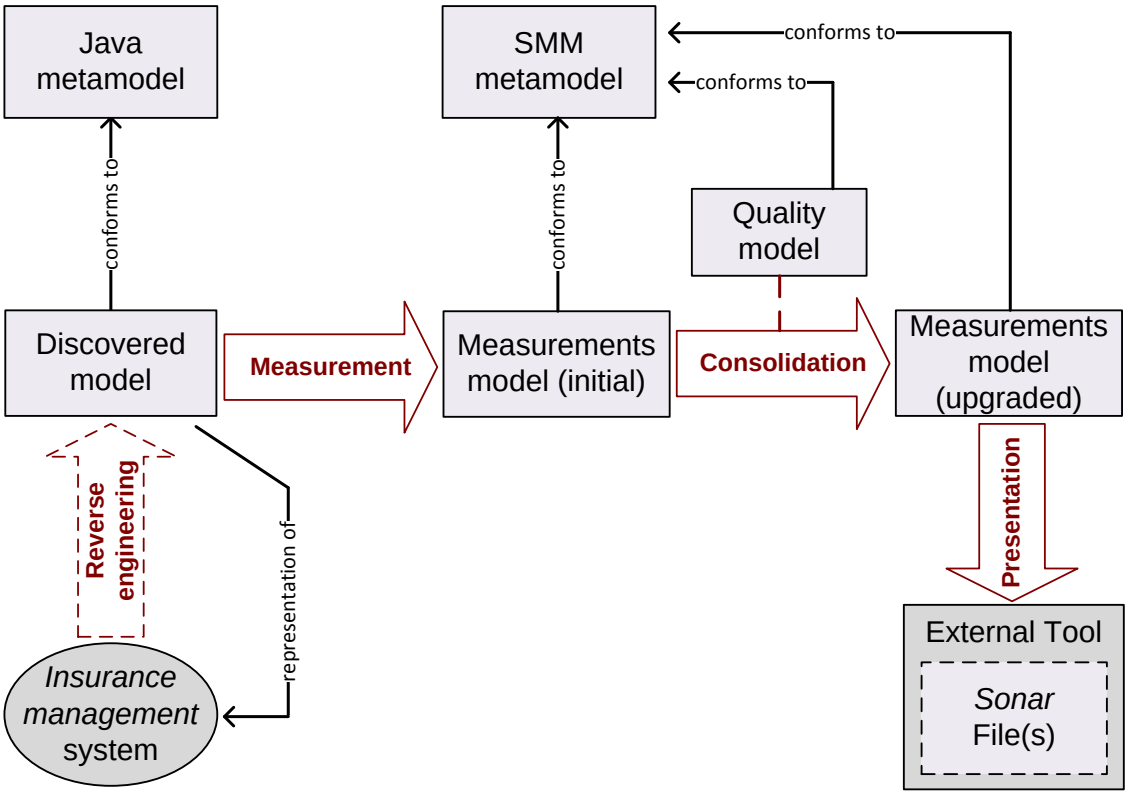

Figure 3.15 – Overall process of the model driven code quality evaluation.

The MoDisco SMM metamodel is largely used in the solution for both specifying homogeneously the measures and representing the results of their computation on the application models. The data exchanged between the three steps of the process are almost exclusively SMM models. They are treated either automatically via Java code or manually thanks to a dedicated editor (Figure 3.16). This editor is based on parts of the MoDisco Model Browser completed with additional specific customizations for the SMM metamodel.

Observations. In the context of this use case, several MoDisco components have been successfully combined as part of an automated quality evaluation solution. The built solution fits the expressed needs in terms of genericity and automation, and has also been concretely applied on the initially targeted insurance management software (as well as in other projects). Figure 3.17 shows samples of concrete final results as graphically displayed in Sonar.

Mia-Software clearly benefited from the use of MoDisco (and some of its preexisting components) in order to build faster the Mia-Quality solution. Moreover, although it is quite difficult to precisely evaluate actual benefits for end-users, Mia-Quality users also

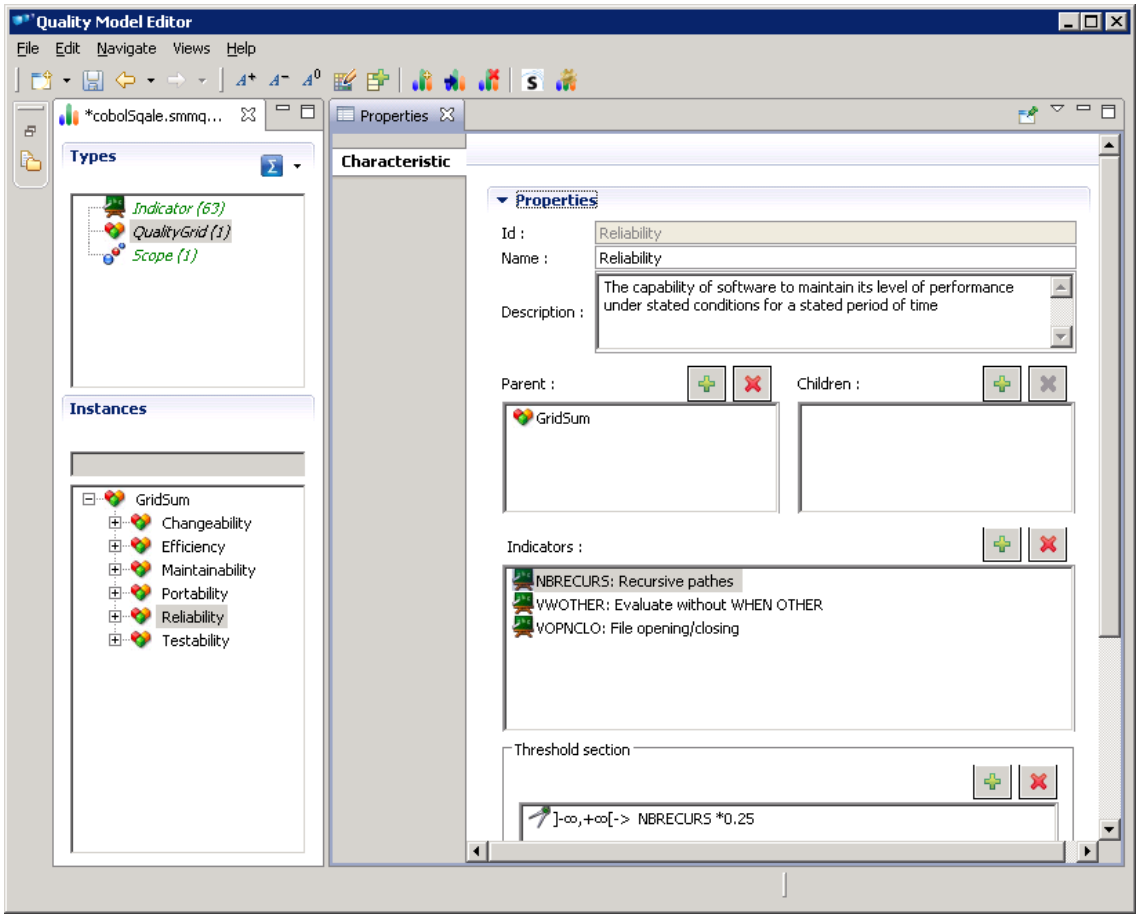

Figure 3.16 – The MoDisco-based Mia-Quality model editor.

reported a positive feedback regarding the productivity and flexibility of the tool (thanks to the use of the MoDisco SMM Metamodel and the integration of several MoDisco Model Browser graphical features).

This use case has demonstrated the ability of MoDisco to be considered not only as an integrated framework (cf. Section 3.4.2), but also as an interesting provider of automatically reusable MDRE components. Thus, their genericity and customizability largely facilitate their integration both within and with other existing solutions.

To summarize, using (parts of) MoDisco to implement the MDRE solution to this use case was valuable for the following main reasons:

— The SMM metamodel, used as both the core metamodel inside the solution and the pivot one for integration with external tools, was already provided for free and directly usable. SMM is also a recognized interoperability standard promoted by the OMG.

— The various MoDisco model discoverers can be reused to provide different kinds of input to the code quality evaluation solution (model discovery phase). It is important to note that only some importers from external tools are distributed in the commercial version so far.

— Several graphical components from the Model Browser (e.g. tree viewer, customization support) have been used as the basis for building the solution quality model editor (model understanding phase) as shown on Figure 3.16, allowing considerably reducing the required development effort.

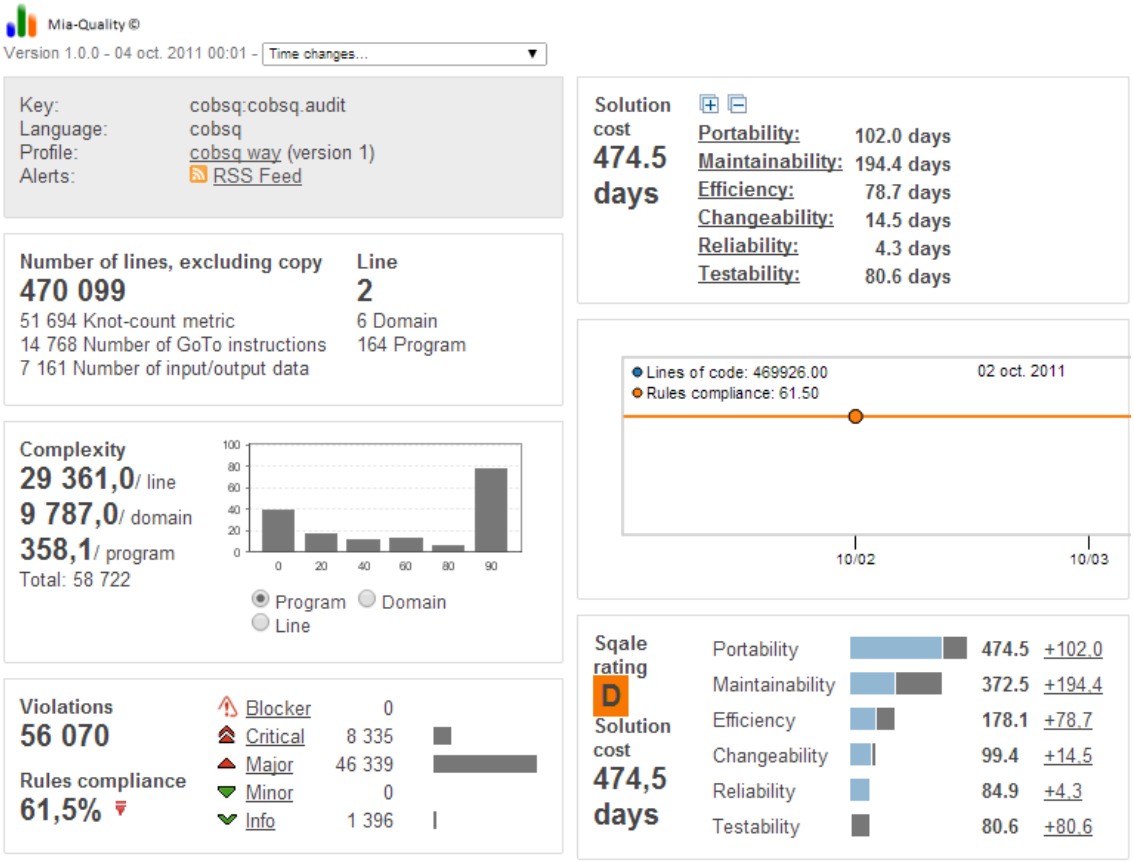

Figure 3.17 – Some quality measurements obtained as output of Mia-Quality.

3.4.3 Performance Benchmarks

Scalability is one of the MDRE main challenges as stated in Section 3.1.4. Thus, this section presents quantitative performance evaluations (cf. RQ3 in Section 3.4.1). Their objective is to highlight the ability of our approach and implementing MoDisco framework to be useful in industrial scenarios where models go far beyond the toy examples sometimes used in research papers. To this intent, we evaluated this overall performance according to several fundamental criteria: execution time, memory footprint, model size, etc. The results show that our approach is one of the suitable candidates for engineers looking for reusable components to build their own MDRE solutions.

The evaluations focus on one of the key components in any MDRE solution when facing scalability issues, i.e. the model discoverer as introduced in Section 3.2.2. In particular, we analyze the results obtained from the automated tests realized on the MoDisco Java model discoverer. Note that different implementations have already been provided in MoDisco for the Java discoverer, all based on EMF. One implementation just uses the standard EMF API while another relies in addition on the CDO framework [60] dedicated to the handling of very big models. However, the evaluations presented here only concern the standard MDE-based implementation of the Java discoverer.

To emphasize on the sufficient scalability of our approach, the experiments presented in this section have been voluntarily realized on a basic development machine. It had the following limited configuration: Quad Core processor at 2.40GHz (Intel), 4GB of RAM, x86 architecture (Windows 7 OS). The measurement components have been implemen-

ted by using the Java standard API. The used memory is computed as the difference at a given time between the total memory (*Runtime.getRuntime().totalMemory()*) and the still free one (*Runtime.getRuntime().freeMemory()*). The required time is obtained via calls to *System.getTime()* at both the beginning and end of (the part of) the process to be measured. Due to some system or Java internals, a measurement overhead can be sometimes observed. However, it can be considered as not significant in the context of the realized experiments.

Globally, the following four performance indicators have been considered by the executed evaluations:

— Size of the discovered Java models.
— Memory footprint during the discovery process.
— CPU time needed for performing this same process.
— Internal repartition of effort inside this process.

Experiment 1: Discovered Models Size

As input for this experiment, we have used several Java legacy systems (plug-ins from the Eclipse Platform in this case) of increasing sizes measured in number of LOC. Note that the first (small) systems have been considered only for evaluation purposes, MDRE and MoDisco becoming actually valuable when applied on systems of medium size or more. Also, the computation of a comment rate in the code could have allowed to evaluate a bit more precisely their actual size. However, the raw number of LOC has been retained as a sufficient indicator for the evaluation.

Figure 3.18 shows the results of evaluating the size of the discovered Java models for each plug-in. The number of model elements (in this context *objects* in the model) and the memory used on hard drive disk after XMI serialization [168] have been computed for each of them.

Eclipse Plug-ins	Lines of code	Number of model elements	XMI model size in Megabytes
org.eclipse.jdt.apt.pluggable.core	781	3,449	0.807
org.eclipse.jdt.apt.ui	2,113	10,217	2.541
org.eclipse.jdt.compiler.tool	2,195	10,187	2.476
org.eclipse.jdt.compiler.apt	6,885	29,444	7.639
org.eclipse.jdt.launching	12,172	52,205	12.801
org.eclipse.jdt.apt.core	13,854	59,270	14.723
org.eclipse.jdt.junit	14,744	66,411	16.206
org.eclipse.jdt.debug	39,411	156,374	38.432
org.eclipse.jdt.debug.ui	39,526	159,028	42.326
org.eclipse.jdt.core	278,045	1,430,345	367.828
org.eclipse.jdt.ui	325,657	1,444,753	393.959

Figure 3.18 – Benchmark on the size of discovered Java models.

As expected, both the number of model elements and the size of the serialized models grow quite fast when the input application gets larger (in LOC). Nevertheless, the discovered models size stays relatively proportional to the input legacy systems size (with a multiplier between 4 and 5 for the number of model elements, and between 0.0010 and 0.0013 for the XMI size). With a small input project of less than 800 LOC, the generated

model contains only 3,500 model elements for a size inferior to 1MB when serialized. A more consistent application of around 40,000 LOC is already represented by more than 155,000 model elements and 42MB of XMI data. Obviously, when considering a larger project (325,000 LOC), the number of model elements becomes huge (almost 1,450,000) as well as the size of the corresponding XMI file (almost 400MB).

This behavior highlights the predictable scalability issues when dealing with even larger legacy systems and their models. The number of model elements is very important because it implies that the available memory could not be sufficient in some case to actually load the full models. Thus, we could explore the possible use of optimization techniques such as the lazy loading of elements at the model manipulation time (e.g. during a transformation). The serialization (or more generally storing) size is also fundamental since it may limit the loading and saving of big models. Several alternatives (e.g. in the Eclipse world) have been proposed relying on different storage environments, and notably databases (cf. Section 3.6).

Experiment 2: Time vs. Memory Footprint of a Discovery Process

This evaluation and the next one focus on the two largest Java projects from Figure 3.18. They are actually the most relevant inputs according to the size of the applications to be normally tackled with MoDisco in realistic MDRE scenarios. Thus, the required time and memory footprint (i.e. used RAM) have been evaluated while executing full Java discovery processes on these two examples. Note that the evaluation itself is also using some memory during the discovery process. Nevertheless, this amount of memory is not significant compared to the total memory used, and so can be voluntarily ignored when analyzing the obtained results as graphically displayed on Figure 3.19.

The full discovery from the *org.eclipse.jdt.core* Java project takes more than 250,000ms (i.e. nearly 4 minutes) and needs at most 350MB in memory. The discovery from the larger *org.eclipse.jdt.ui* project has been performed in around 400,000ms (almost 7 minutes) and has used more than 500MB of memory. Considering the nature and frequency of a classical discovery process, the total execution times are in both cases reasonable according to the inputs size.

The potentially important memory footprint (notably at the end of the process) could imply scalability issues when using standard computers. However, this in-memory footprint is interestingly proportional to the time. This could allow anticipating the amount of dynamically allocated memory for a given discovery process. Such an optimization could be realized according to the input legacy application size and also the current process running time. This knowledge could be particularly useful when working on optimizing long discovery processes from very large legacy systems.

Experiment 3: Time Repartition During a Discovery Process

Finally, a last experiment on the two same examples has focused on the internal effort repartition during a full Java discovery process. The objective was to collect interesting information on the required time per different subtask inside such a discovery process. In that specific case, these subtasks are:

1. Creation of the abstract syntax tree from the Java program.

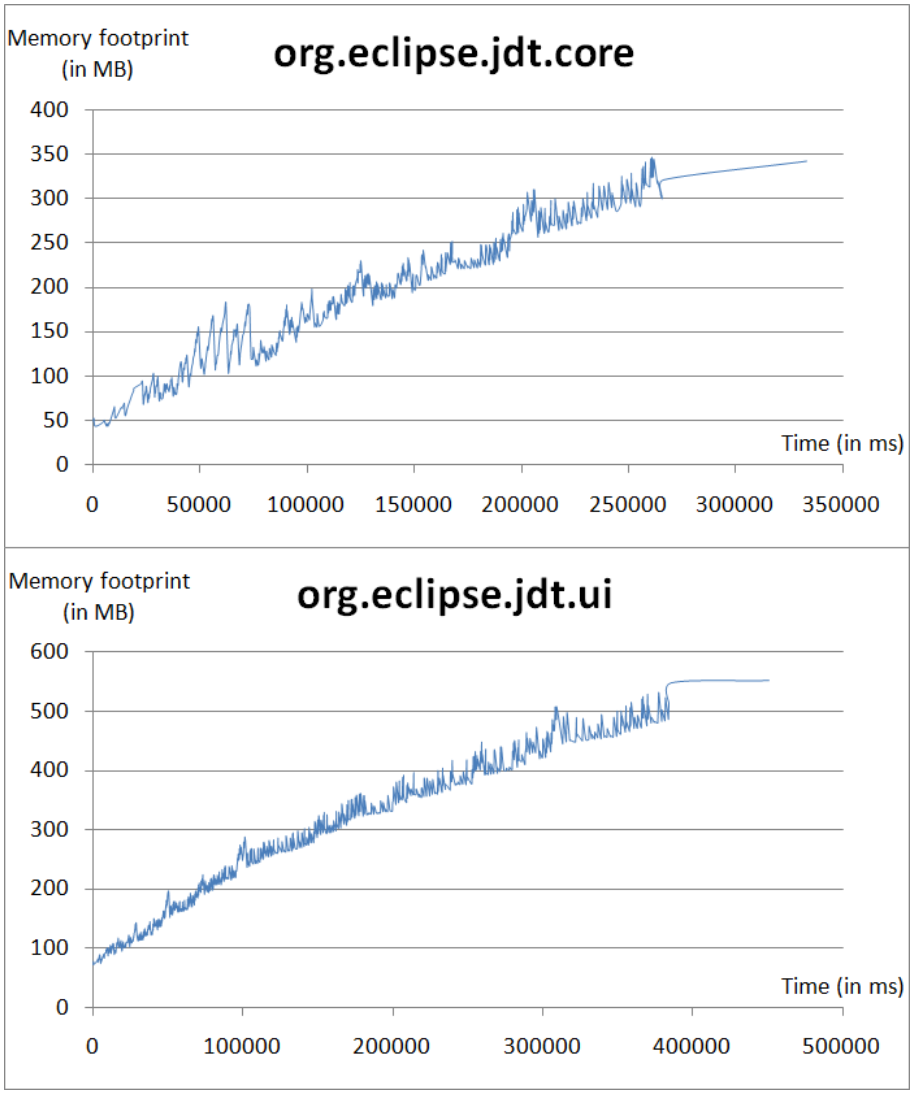

Figure 3.19 – Benchmark on the time and memory footprint of a Java discovery process.

2. Visit of this tree to generate the corresponding model elements.
3. Resolution of the references between the different created model elements.
4. Redefinition of some methods when required.
5. Serialization in XMI of the built model.

Note that the provided total discovery times are approximations, advanced reference resolutions having been voluntarily skipped. In any case, the time allocated to this reference resolution largely depends on the needed degree of details. This could allow interesting scenario-specific performance optimizations. The obtained results are graphically presented in Figure 3.20.

This evaluation generally emphasizes that the Java program abstract syntax tree creation and discovered Java models serialization are very time consuming tasks. It also denotes that the actual visit of the tree (including both the navigation and production of model elements) is relatively fast compared to them. This highlights two main discovery process parts where considerable performance gains could be obtained. This last experiment provides precious indications on where to significantly improve the overall discovery process scalability (cf. Section 3.6).

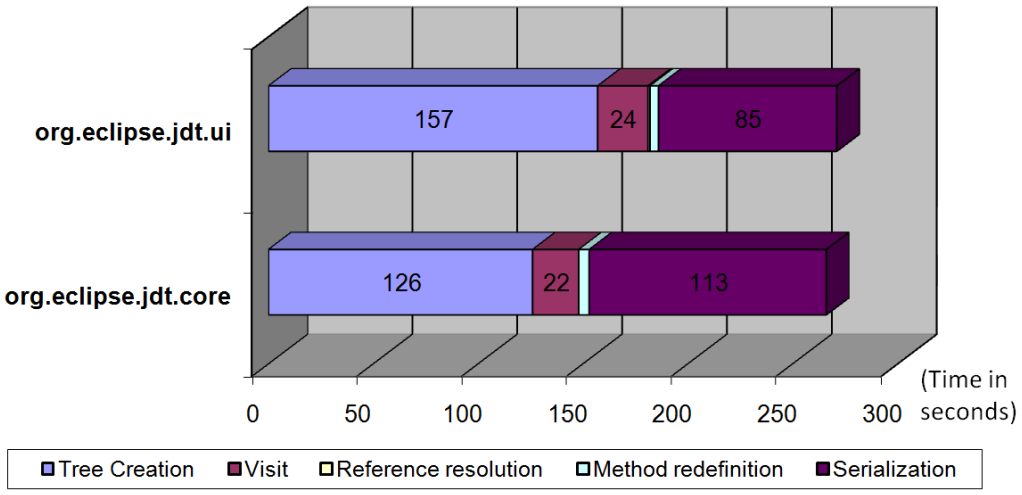

Figure 3.20 – Benchmark on the time repartition during a Java discovery process.

3.5 The FREX Component

As previously introduced in Section 3.1.4, there are different MDRE challenges to be tackled. The proposed conceptual approach and MoDisco framework described in this chapter intend to address them. However, the presented work has mainly focused on providing support for reverse engineering with an emphasis on design elements. Properly reverse engineering the runtime execution of existing software has not been explored a lot so far. More generally, while there has been a strong focus on static analysis techniques for structural aspects of software, there have been (much) less attempts to target their behavioral aspects by dynamic analysis techniques.

In this section, we introduce the fREX component as the initial result of further research work on applying our proposed approach to the reverse engineering of system executable behaviors. Instead of letting users re-implement such a feature from scratch whenever needed, we decided to contribute a first basic solution to this complex problem. This way, it acts as a complementary solution to the already provided MoDisco reverse engineering support for structural aspects. The fREX contribution is actually twofold:

1. An simple open and extensible framework that is capable of automatically generating and executing fUML models from existing applications.

2. A base mapping between UML activity/class diagrams (i.e. fUML) and the core language features of Java, putting the focus on behavioral aspects and their execution at model-level. To obtain the required runtime information, fUML comes with a dedicated Virtual Machine (VM) that has been extended to provide execution traces as a runtime model [137].

We start by motivating further the need for fREX (in Section 3.5.1 and by providing an overview of the proposed framework as another instantiation of our MDRE approach (in Section 3.5.2). Then, we describe a concrete example in order to illustrate the initial Java-to-fUML mapping that has been designed and implemented (in Section 3.5.3). We end by opening on interesting future application scenarios for our fREX framework (in Section 3.5.4).

3.5.1 Motivation

As mentioned before, there are already significant results as far as modeling structural aspects of software is concerned. However, there has been less initiatives really focusing on modeling precisely software behaviors.

Background

In UML, application behavior can be defined either interaction-oriented by using sequence diagrams or state-oriented by using state machine diagrams. The behavior triggered by such interactions and states can be represented in details by means of activity diagrams. fUML [163] corresponds to the core subset of UML that has been identified as relevant for representing software behavior with the main purpose of executing it. In particular, fUML makes explicit the semantics of both class and activity diagrams for a dedicated VM to interpret them. Thus, similarly to existing object-oriented programming languages (e.g. Java or C#), fUML provides concepts for defining classes with attributes and operations, abstract classes, multiple inheritance, enumerations as well as an extensible type system. Operation bodies are implemented by activities and via the action language provided by UML, enabling the expression of manipulations and other computations. As a consequence, fUML appears to be a potential language for capturing the behavior of source code at model-level. As it is capable to represent the behavior in executable form, it enables dynamic analysis to be carried out directly at model-level instead of code-level.

In addition to fUML, Micro-KDM [174] is also capable of representing application behavior in a language-independent way at model-level. However, it currently does not come with an explicit semantic specification and execution engine. Another possibility would be to extend other languages used for measurement and metric calculation like FAMIX [54] with an action language such as the one already provided by fUML. In all mentioned cases, more reverse engineering support is still required in order to automatically obtain relevant and valid behavioral models from already existing source code.

Generally, the elaboration of mappings between programming and modeling languages such as Java and UML is not new [81, 105, 150, 124]. However, only a few approaches [88, 98] have been considering UML activity diagrams for expressing application behavior at model-level. These approaches focus on forward engineering as they use Java as the output language and their mapping (from UML) is encoded by code generators. The base of such mappings may also be reusable in a reverse engineering context such as ours, but they would have to be complemented to express concepts such as `ControlFlow` and `ObjectFlow` that are not explicitly represented in the application code. The difference between existing approaches that deal with reverse engineering of activity diagrams from application code [129] and our approach is that their proposed tooling is strongly language and visualization-oriented, while we follow a more generic approach targeting model execution. In our case models are to be discovered solely by static analysis as we aim at obtaining a representation of the overall behavior independently from any execution scenario.

Finally, there is already a significant body of software analysis work covering different techniques and tools [46] . Existing approaches that support dynamic analysis

typically gather runtime information directly at code-level, based on which the analysis is then carried out. Many of these approaches use UML(-like) representations to capture actual analysis results in terms of models. For instance, the UML sequence diagram is often used in the context of execution trace analysis. Our approach is different as we aim at performing the full dynamic analysis at model-level, in particular on top of previously discovered fUML models. Such models are more expressive compared to program code in several respects (e.g. different kinds of relationships, precise multiplicities, explicit control flow and data flow) which is beneficial for realizing more powerful dynamic analysis tools. Analysis tools working at this model-level can directly benefit from these richer representations as well as from the large ecosystem of modeling techniques and tools.

Illustration

A practical illustration of the need for reverse engineering capabilities for executable behaviors is the ARTIST initiative. It has resulted in both an overall methodology and the related tooling aimed at providing a global model-based re-engineering approach for migrating existing software more easily to novel cloud offerings [17, 143]. Notably, this involves selecting a cloud storage solution given a set of persistence requirements derived from software implemented in a variety of programming languages. This in turn requires at least (i) to obtain a precise data model and (ii) to understand how application data is persisted and retrieved. However, statically producing a representation allowing to reason on structural aspects is not enough. On the contrary, it is highly required to dynamically analyze the behavioral aspects of the system for deriving improvements concerning non-functional aspects. Dealing with such a scenario highlighted the practical need for a dynamic/behavioral reverse engineering support, as well as the effort required to realize it separately for several different programming languages (e.g. Java or C# that were both in the scope of the project). This would imply duplicating the work, e.g. to instrument source code and produce the runtime information in terms of machine-interpretable execution traces.

Among the different paradigms and corresponding solutions available, the global MDRE approach proposed in this chapter is the one that has been applied in the ARTIST context. This has notably required to use the Java and UML [167] model discovery support provided by MoDisco in order to obtain models representing the applications in a programming language-independent way. Studying such mappings between programming and modeling languages (e.g. Java and UML) has a long tradition in both reverse and forward engineering. The work in this area focuses mainly on UML class diagram to capture structural aspects of an application, while behavioral aspects are typically expressed (mostly partially) in terms of sequence and state diagrams. With the relatively recent emergence of the fUML [163], UML activity diagrams appear to be more appropriate for capturing behavioral aspects in a way that they can be executed directly at model-level.

3.5.2 Proposed Framework

A fundamental idea of the proposed fREX framework is the central use of a common representation format for all behavioral concerns. fUML, as a subset of UML focusing

on executability aspects, plays the essential role of a pivot language in our solution.

Architecture

The fREX framework is intended to facilitate the construction of several structural and behavioral models on a given software, and this at different levels of abstraction depending on the reverse engineering needs. It follows the two-phase process as proposed by our MDRE approach:

— **Model Discovery** generates from the software artifacts and/or their executions the needed initial models representing the raw behavior of the considered software. In our present case, base fUML models are automatically discovered from Java source code.

— **Model Understanding** further analyzes the previously obtained fUML models by producing derived traces and/or models proposing different additional relevant views. In our present case, we employ a fUML VM to execute these models and test the produced traces.

The overall architecture of fREX and its current Java support is presented in Figure 3.21.

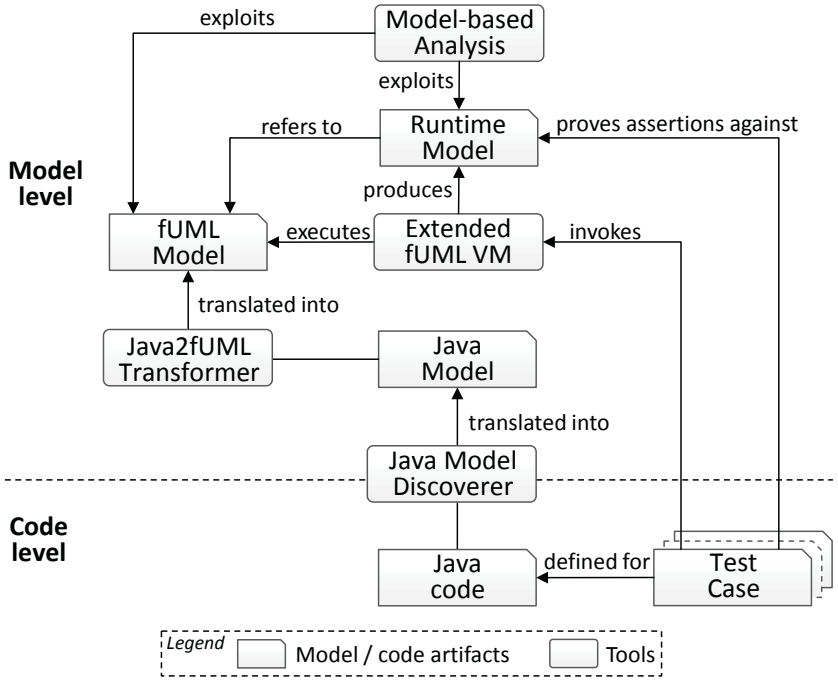

Figure 3.21 – Overall architecture of the fREX framework.

Model Discovery

Producing fUML-based representations from source code requires both overcoming different encodings and resolving language heterogeneities. Thus, instead of directly translating plain code into fUML, a two-step approach is preferable for the fUML model discovery phase. Firstly, the source code is *translated into* a code model (a Java model in the present case) using a low-level specific *model discoverer*. The obtained model conforms to a metamodel of the programming language that precisely describes its terminology and structures. Secondly, this code model is *translated into* a fUML model

that resolves language heterogeneities by relying on the correspondences between the given language (here Java) and fUML metamodels. This is implemented as a so-called *transformer*.

Model Understanding

Having obtained a proper fUML model, it can then be directly *executed* by the fUML VM in a model understanding phase. In previous work, some additional tracing support has already been integrated into an existing fUML VM [137]. In particular, a metamodel has been designed allowing to capture the runtime behavior of fUML models in terms of execution traces and to extend the fUML VM for recording execution traces as instances of this metamodel. Hence, as a result of the model execution onto the fUML VM, a runtime model is *produced* capturing execution traces *referring* to the executed fUML model. They provide information on executed activities and their actions including information about their call hierarchy, the chronological and logical order of their execution, and information on the runtime states of the model during the execution. The generated runtime model along with the previously discovered fUML model can then be *exploited* by model-based analysis techniques. These include model refinement, slicing or view generation for instance (cf. Section 3.5.4 for possible application scenarios).

In addition, in order to check the validity of the produced fUML models, we apply a test-driven approach. The base idea is to define and run unit tests for asserting that the discovered fUML models actually capture the original behavior of the Java code. Actually, we compare for a given input the result of a given fUML model execution (i.e. a runtime model) against the result of running the corresponding piece of code. Note that the code-level test cases are for now manually translated into model-level test cases implemented with Java. However, an automated translation of test cases is in principle possible if all programming language constructs needed for defining test cases are supported by the model discoverer. We apply such a test-driven approach to allow continuously validating new language correspondences that are implemented by the available transformers (e.g. the Java-to-fUML one, cf. Section 3.5.3).

Tooling Support

The current implementation of the fREX framework relies on the combined use and integration of several components from the EMP. The initial low-level Java model discovery step (from a source Java project) is automatically performed by reusing the corresponding MoDisco component. Then, the previously introduced Java-to-fUML mapping is currently implemented using AtlanMod Transformation Language (ATL). As mentioned before, the extended fUML VM developed in the Moliz project [36] is used to provide the required model execution capabilities. Finally, JUnit test cases have been implemented to ensure that the produced fUML models behave (i.e. execute) as expected. In addition to these core aspects, a couple of UI plugins providing fREX-specific contextual actions have also been implemented. They offer to users simple ways of launching the different steps of the reverse engineering process from the Eclipse workbench they are familiar with.

We checked the completeness and correctness of our implementation, concerning both

the model discovery/mapping and the model understanding/execution steps, with the following practical testing methodology:

1. Develop Java examples that use the aforementioned Java structures.
2. Discover Java models from them.
3. Transform these Java models into fUML ones.
4. Execute these fUML models and the original Java code via unit tests.
5. Compare the outputs produced by executing the fUML models and the original Java code.

The source code of the fREX implementation as well as a corresponding demo/video (applying our testing methodology on a concrete example), is publicly available [147].

Expected Benefits

The proposed fREX architecture (including notably the use of fUML as a pivot representation format) comes with several interesting benefits from a reverse engineering perspective.

— *Extensibility* is allowed from the model discovery side, as new model discovery components targeting fUML can be implemented from various kinds of software inputs. For instance, different fUML model discoverers could be built for supporting behavioral reverse engineering from both Java and C# source code.

— *Genericity* and *reusability* are permitted from the model understanding and analysis side, as existing components consuming fUML models can be reused independently from the original nature of the treated software. This way, the same execution capabilities and/or analysis transformations can be used indifferently on all fUML models.

— *Non-intrusiveness* is supported because only (fUML) models are considered for execution and analysis. Hence no modifications (e.g. for code instrumentation purposes) are required at source-level anymore, as everything can be performed at model-level (e.g. via the used fUML VM).

3.5.3 The Java-to-fUML Example

To demonstrate the capabilities of fREX for an extensively used programming language, we decided to start working on the Java case. As an example, Figure 3.22 gives an overview of the different artifacts and models considered and produced by fREX from a given piece of Java code.

Notably, some Java code and a corresponding (reverse engineered) fUML model are depicted there. Application structure and behavior are captured by a class diagram and activity diagram, respectively. An excerpt of the traces resulting from the execution of the illustrated activity (i.e. the runtime model) is shown beneath the diagrams. Due to the high complexity of a complete mapping between Java and fUML, we started by addressing a subset of Java called MiniJava [179]. Thus, we decided to voluntarily delay the treatment of some other aspects of the language. Our current Java-to-fUML mapping is inspired from initial work within the standard fUML specification which we refined, extended and implemented in terms of a Java-to-fUML model transformation.

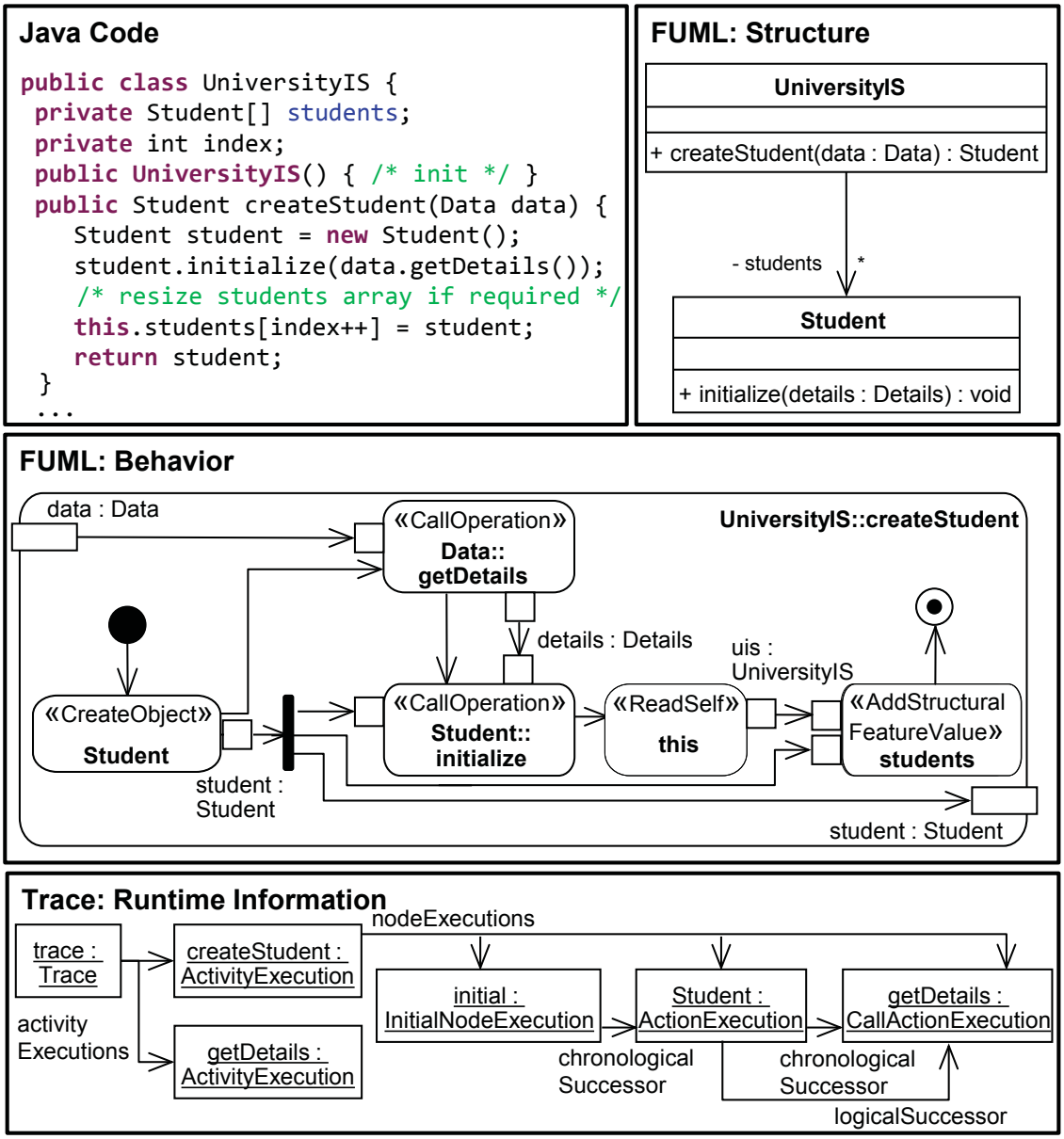

Figure 3.22 – Java code expressed and executed by means of fUML.

Table 3.3 introduces the conceptual mapping required to discover an fUML model form the Java code of Figure 3.22. It shows the rules for translating the statements of the `createStudent` method into corresponding fUML model elements. The concepts

on the left hand side of the table refer to the terminology of the Java Language Specification [102], whereas in the right hand side are corresponding concepts defined by the fUML metamodel [163]. In this present work, the focus is set on behavioral aspects by capitalizing on the structural mapping realized in JUMP [18] and by complementing it with new behavioral elements.

Java Concept	fUML Concept
MethodDeclaration md	**add** Activity a
	a.name = md.name
	a.specification = -- *infer respective Operation from structural part*
ReturnType rt	**add** ActivityParameterNode rapn
	rapn.name = "return"
	rapn.type = rt.type
	rapn.parameter = -- *infer respective Paramter from structural part*
FormalParamter fp	**add** ActivityParameterNode fapn
	fapn.name = fp.name
	fapn.type = fp.type
	fapn.paramter = -- *infer respective Paramter from structural part*
Block b	**add** InitialNode in, FinalNode fn, StructuredActivityNode san
	-- *infer control flow from b.statements*
VariableDeclaration vd, ClassInstanceCreation cic	**add** CreateObjectAction createOA
	createOA.name = vd.type.name
	createOA.classifier = vd.type
	add OutputPin op, ObjectFlow of, ForkNode fn
	of.source = op, of.target = fn
MethodInvocation mi	**add** CallOperationAction callOA
	callOA.name = mi.method.name
	callOA.operation = mi.method
	add InputPin ip, ObjectFlow of -- *for target object, e.g., student*
	add InputPin ip, ObjectFlow of **foreach**
	FormalParameter fp **in** mi.method
	-- *infer source and target of ObjectFlows*
	add OutputPin op **for** ReturnType rt **in** mi.method
Assignment a **switch**(a.leftHandSide) **case:** ArrayAccess	**add** AddStructuralFeatureValueAction asfva
	asfva.name = -- *infer name from left hand side*
	asfva.structuralFeature = -- *infer feature from left hand side*
	add InputPin ip, ObjectFlow of **fo**r a.leftHandSide
	add InputPin ip, ObjectFlow of **fo**r a.rightHandSide
	-- *infer source and target of ObjectFlows*
ThisExpression	**add** ReadSelfAction rsa
	rsa.name = "this"
	add OutputPin op

Table 3.3 – Mapping between MiniJava and fUML.

From a structural perspective, a method declared in Java corresponds to an operation in UML. In order to capture its behavioral elements at model-level, it is also mapped to an activity that is linked to the operation (see `specification` property). The name of the activity is derived from the method signature. Furthermore, formal parameters and

the return type defined by the method signature are mapped to parameter nodes of the corresponding activity. As an activity explicitly defines control nodes at which the execution starts and ends when it is invoked, those nodes, i.e. `InitialNode` and `FinalNode`, are created by default for each activity. If a `FinalNode` has been executed, the activity execution terminates. The activity also terminates if no activity node is enabled anymore. After the termination, the activity execution collects the object tokens residing on output activity parameter nodes and provides them as output (see the `student` object).

A created instance variable (see the `student` instance) is mapped to an fUML action that creates an object (i.e. `CreateObjectAction`). The action name and classifier are derived from the type (Java class) that is instantiated. Additionally, an output pin is created at which the action puts the instantiated object at runtime. The instantiated object is distributed to possibly several other actions via a fork node. It is connected to the action's output pin via an object flow edge. The latter ensures that the objects are offered to the successor activity nodes once the current node has been executed.

A method invocation is mapped to an fUML action for calling operations (i.e. `CallOperationAction`). Its main properties (i.e. `name` and `operation`) are derived from the signature of the method that is invoked. Input pins and the respective object flow edges are created for the target object of the invocation and for the values passed to the parameters of the invoked method. Also, an output pin is created if the invoked method returns a value.

A value assignment to a multi-valued Java variable (e.g. an array of students) is mapped to a named fUML action that adds a value to a structural feature (the upper value of its multiplicity is assumed to be unbounded, i.e. 0..*): i.e. `AddStructuralFeatureAction`. The latter is referenced accordingly by the action (see its `structuralFeature` property). Again, input pins and the respective object flow edges are created for the left hand side as well as the right hand side of the assignment statement.

Finally, a `ReadSelfAction` along with an output pin are created when Java *this* keyword is used to refer to the member of the current object from within an instance method. The *this* keyword may not only be used in the context of a method declaration but also a constructor declaration.

3.5.4 Possible Applications

Having fUML models that represent behavioral aspects of existing software, the way is paved for further software comprehension and analysis carried out directly at model-level. In order to give an initial impression of the applicability of our fREX framework, we consider hereafter three main families of model-based analysis techniques that could (re)use the produced fUML models.

Model Refinement

A first way of dealing with the obtained fUML models is to refine them using one or several model transformations. One of the objectives may be to insert additional information into the fUML models, possibly computed and/or coming from other models. Thus, in the proposed framework we are able to complement the initially generated fUML mo-

dels by using runtime information coming from the trace models produced by the fUML VM (cf. the one from our example in Figure 3.22). Other interesting refinements could be achieved too at fUML-level. For example, transformations could be proposed in order to explore automatically, based on model executions, the refinement of associations into bi-directional associations or compositions with more accurate multiplicity constraints in the fUML models. This requires an analysis of the execution traces to observe if changes on one of the two potential unidirectional associations are always replicated on the other, suggesting that they are indeed representing the same concept. We have already implemented a first version of such a fUML model-to-model transformation for exploration purposes.

Model Slicing

The obtained models convey many types of information that are more explicit than in source code, e.g. associations between classes as discussed before. However they may not scale well in some cases, notably when the volume of represented information becomes too large. Thus, model-based slicing techniques [4] can help in capturing only relevant parts of a larger model for a given purpose. The class diagram depicted in our example can already be considered as a slice because it shows a reduced part of the whole university information system. With the dynamic approach in our framework, slices can be produced that contain only model elements required for a specific execution, e.g. creating a student entity, facilitating the comprehension of the parts of the model behavior relevant to specific functionalities. Complementary to this, model slicing could be extended by chaining different transformations computing distinct slices. For instance, in a first step, models capturing behavioral aspects are sliced according to a given slicing criterion. Then, in a second step, the structure influenced by the execution of the sliced behavior may be obtained. The latter can be achieved by computing a model slice according to the type information of the produced objects. Additionally, these slices may be propagated again to other UML models such as architectural ones (e.g. in UML component diagrams).

View Generation

Generating different useful views on existing software is one of the major purposes in reverse engineering [38]. A view enables turning the focus on certain concerns where a pertinent viewpoint specifies the conventions for representing such a view. As our approach relies on fUML and as its parent (i.e. UML) can be considered as a multi-viewpoint language, several interesting views are naturally conceivable for our example. For instance, in order to represent high-level interactions relevant in the context of creating a student in the university information system, a dedicated view based on UML sequence diagrams may be produced using trace analysis techniques [25]. Deriving partial (and usually more abstract) representations of the software behavior would allow the right amount of information to be conveyed to each stakeholder involved in the system. Finally, we also foresee the potential application of domain-specific languages for behavioral analysis, highlighting aspects which are not straightforward to represent in pure UML models. To this intent, more generic (in the sense of metamodel-independent) model view approaches that allow relating together models which conform to different metamodels

could be reused (cf. next Chapter 4).

3.6 Conclusion

The number of software systems to be maintained, extended or generally evolved has grown considerably during the last decades and will continue to do so. In order to deal with all this legacy software, both economically and technologically speaking, reliable (semi-)automated reverse engineering solutions must be provided. To reach this objective, integrating MDE techniques in reverse engineering solutions shows promising and innovative results.

In this Chapter, we have presented a generic and extensible MDRE conceptual approach and an implementing framework named MoDisco. They intended to facilitate the elaboration of MDRE solutions actually deployable within industrial scenarios. The provided description includes the overall underlying approach, architecture, available components and the detail of its two main reverse engineering phases, namely *Model Discovery* and *Model Understanding*. Real applications of our approach and the MoDisco framework on concrete industrial use cases have also been presented as well as performance benchmark results. We have also introduced the fREX component resulting from initial further research work. Its goal is to target the reverse engineering of system executable behaviors, complementary to the reverse engineering support for system structural aspects as already provided by MoDisco.

On one hand, MoDisco as a project has developed significantly since its creation. Thanks notably to the active support of Mia-Software (Sodifrance) [190], it has grown from a research initiative to an industrialized project having a user base and regular stable releases. The project is continuously open to requests, enhancements or new contributions either by using the Eclipse infrastructure tools such as the forum, Bugzilla, etc. or by directly contacting the project team. But getting external people or new partners to actually join in such an open source project, with the long-term involvement that this implies (e.g. maintenance or release engineering tasks during several years), is not something easy as we see later in Section 5.3. As a consequence, and despite our efforts, we have observed a progressive diminution of the activity around the project since some years (e.g. in terms of number of posts on the forum or of bug submissions).

On the other hand, the conceptual approach and MoDisco framework as such has been successfully deployed or reused in different industrial MDRE scenarios (cf. Section 3.4 for some concrete examples). It has shown in practice its capabilities in terms of adaptability/portability and no loss of required information. However, we continued testing the extensibility and improving the coverage of the framework by applying it on other technologies. Some preliminary experiments around the migration of C#/.NET systems have been performed within the context of the ARTIST FP7-ICT European project focusing on software migration to the Cloud [30, 143] (cf. Section 5.2.1). Finally, some MoDisco components are being progressively externalized to facilitate their reuse in other projects (which are not necessarily dealing with reverse engineering). A remarkable example of this has been the creation of the Eclipse EMF Facet project [67] as a spin-off from MoDisco.

4

Model Federation and Comprehension

As described before in Chapter 3, when reverse engineering software systems, models that conform to different modeling languages (i.e. metamodels) are used by engineers and architects to describe the concerned system from various perspectives. This often leads to scattering of information across (possibly many) heterogeneous models, and to overlappings/redundancies that can create inconsistencies in the system description [87]. Thus, engineers may have difficulties to comprehend efficiently the complete system description by looking at all these models (and their relationships) in full detail.

This is even a major concern when dealing with systems of systems or CPS [55], e.g. especially in an industrial context where multi-disciplinary engineering takes place [85]. In such cases, models actually need to cover hardware infrastructure (i.e. physical entities such as energy grids, networks, production plants or IoT devices), deployment, usage scenarios as well as non/functional properties of the system (e.g. performance, reliability, maintainability). All these elements are also important from a general migration/-modernization perspective (cf. Section 1.2). The different involved models have strong dependencies and interconnections to the software models we are familiar with. They are generally combined altogether in order to better monitor and analyze these systems, e.g. at design time and/or runtime.

View-based approaches in software engineering have been proposed and used to tackle these issues [87]. Initially, they mostly followed a strategy of proposing a fixed set of predefined *viewpoints* to be used in different application domains or scenarios. This happens for instance with most architectural frameworks such as Zachman [219] or RM-ODP [136], each viewpoint targeting particular perspectives of the system to be considered. It offers several advantages such as improved comprehension or a more integrated and user-friendly tool support. However, these approaches generally lack the flexibility required in many scenarios. This is notably the case when the useful model views go beyond a limited set of viewpoints and may change over time.

Recent advances in MDE/Modeling have fostered the possibility of having more flex-

ible view-based approaches, based on metamodeling and model transformation techniques notably (cf. Section 2.1.2). The notion of ad hoc views computed via queries has been studied intensively in past decades. While the metaphor helps engineers to get the meaning of these concepts, similar problems such as (incremental) view updates arise. These *model view* approaches usually allow creating custom (semi-)automatically generated views over possibly heterogeneous models. Such a capability can help reducing accidental complexity in any software- and system-based processes.

There are many real-world use cases in which the direct support for such *model views* can be required. In addition, there are also a lot of scenarios in which the use of model views can be beneficial as part of a larger model-based solution to a given complex problem. From our own practical experiences within various collaborative industrial projects in the past years, we have already observed relevant applications in areas such as (meta-)model federation/integration [33], reverse engineering [143] or language maintenance and evolution [31]. However, each scenario requires a different trade-off in terms of model view capabilities. For instance, a good expressivity of the view definition and an efficient data synchronization support are fundamental to model federation or language evolution cases. In the context of reverse engineering, scalability in the computation appears to be important due to the potential handling of very large models.

In this chapter we present EMF Views (and our general survey on model view approaches), as the second main contribution of this thesis (cf. Section 1.4), that is both:

1. A generic, extensible and global model view approach to facilitate the building and handling of model-based views in various contexts.
2. A ready-to-use framework, implementing this approach as an Eclipse tooling on top of the Eclipse/EMF environment.

The rest of this chapter is structured as follows. Section 4.1 provides our extended survey detailing the current state-of-the-art and main challenges related to model view approaches in general. Section 4.2 describes the proposed conceptual approach, relying notably on a *model virtualization* backend and two DSLs for expressing viewpoints and views in different contexts. Section 4.3 presents the EMF Views technical framework implementing this approach and languages, i.e. its overall architecture and different provided features. Section 4.4 explains how we evaluated the approach and related Eclipse/EMF-based framework, via both concrete use cases and performance benchmarks. Section 4.5 concludes this chapter by summarizing the main realizations as well as their current limitations.

4.1 State of the Art and Challenges

Given the number of existing approaches, most of them only providing partial solutions, it is quite difficult to know how each approach compares to the others. From such scattered information, it is also complicated to identify which one(s) may be better suited for given needs, e.g. in a reverse engineering context as presented in Chapter 3. In this section, we contribute a detailed study of the state-of-the-art that intends to provide orientation in this area. Thus, in what follows, we start by giving some general definition related to *model view* approaches (in Section 4.1.1). We also describe what we believe to be the main characteristics of model view approaches, via a dedicated feature model we

propose (in Section 4.1.2). Then, we evaluate a selected set of relevant approaches according to this feature model (in Section 4.1.3). Note that the complete methodology used in order to perform this selection is available from our original journal publication [27]. We end this section by presenting some important challenges regarding *model view* approaches, and Model Federation and Comprehension in general (in Section 4.1.4).

4.1.1 General Definitions

In the MDE/Modeling domain, the terms *view*, *viewpoint* or *viewtype* have been used in several different ways. From the very early viewpoint approaches [214, 87] up to the ISO standard 42010 [113], various definitions for these terms have been given. In this thesis, we have decided to consider the definitions summarized in the next paragraph as particularly relevant in our MDE/Modeling context [100].

A *view* is usually a special kind of model. A view contains information that is related to and coming from other models, which can also be themselves other views. A view is always a view on something. Thus, in an engineering context, the set of physical and/or logical entities that a view represents is called a *system*. Such a system can be observed from different *viewpoints*, each of them providing different perspectives over it. The relation between views and other models is specified by various means such as (model) transformations, rules, queries, or other formalisms. As any model, a view conforms to a metamodel which is usually called *viewtype*. This viewtype can be defined a-priori, or can be sometimes deduced from the specification of the view itself. This is however not the case in general, as viewpoint/viewtype and view specifications are usually clearly separated. Such a situation can be practically observed in many of the approaches that we present later in this section.

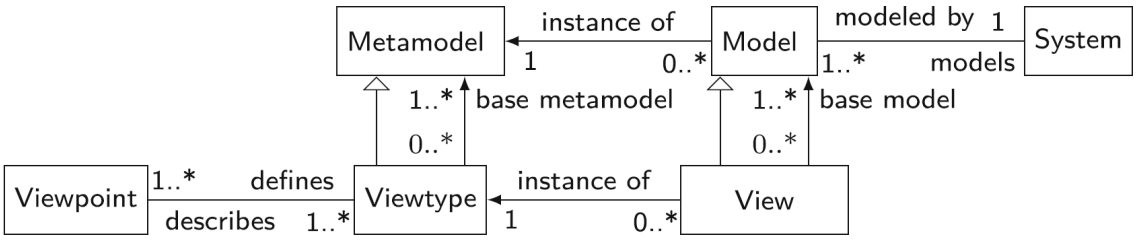

Figure 4.1 – A terminology for model view approaches.

We propose some general definitions for the most frequently encountered terms while searching for and studying solutions for views/viewpoints on models. These main terms are graphically summarized in Figure 4.1 and textually explained in the following:

— A **system** is a unit consisting of multiple interdependent components, which are designed and implemented by engineers. A system encompasses software, hardware, requirements, as well as all other artifacts created during its development process.

— A **viewpoint** is the description of a combination, partitioning and/or restriction of concerns from which systems can be observed. In our modeling context, it consists of a set of concepts coming from one or more metamodels, eventually complemented with some new interconnections between them and newly added features (cf. **viewtype** definition).

— A **viewtype** is a metamodel that describes the types of elements that can appear in a view, i.e. the formalism/language actually used. An element in a viewtype may be part of one of the base metamodels, or may be specifically defined for the viewtype. A given viewtype can be relevant for several viewpoints, and a viewpoint usually defines several viewtypes.
— A **view** is a representation of a specific system from the perspective of a given viewpoint. In our modeling context, it is an instance of a particular viewtype and consists of a set of elements coming from one or more base models. It is eventually complemented with some new interconnections between them and additional data, that are manually entered and/or computed automatically (usually via one or more model transformations).
— A **base metamodel** is a metamodel that contributed to a given viewtype definition. Depending on the approaches, a viewtype specification can possibly have one or several different base metamodels.
— A **base model** is a model that contributes to a given view. Depending on approaches and on the corresponding defined viewpoint (and related viewtypes), a view can possibly gather elements coming from one or more base models.

4.1.2 Characterization of Model View Approaches

Based on our own experiences working on/with model views in the past years, and on a deep study of the related state-of-the-art (cf. our original journal publication [27] for more details on the used methodology), we propose a feature model describing what we consider to be the main characteristics of a model view mechanism. This feature model is depicted in Figure 4.2 and will be used in Section 4.1.3 to better describe and compare the model view approaches we have identified and selected. Note that the symbols (#, @, *, etc.) used in the top diagram from this figure, at the right of some proposed features, are meant to point to the corresponding bottom sub-diagrams that detail further these features.

We have identified three main categories of features that model view mechanisms can potentially cover. The first one gathers capabilities which concern the general *type structure* of the mechanism. The two others distinguish between *design time* and *runtime* aspects of the view/viewtype specification, computation and handling. We describe each one of these categories and the sub-features they contain in the following subsections.

Type Structure

— **Metamodel/Model Arity:** A fundamental aspect of a model view mechanism is its arity at both metamodel/ and model-level. Some mechanisms allow specifying viewpoints/viewtypes over a single metamodel only, while others allow combining several metamodels together in a same viewpoint/viewtype specification. Similarly, depending on the considered mechanism, corresponding views can be computed over a single model and/or combining several distinct ones. In the latter case, the involved models can potentially conform to different metamodels (as used in the related viewpoint/viewtype specification).
— **Closedness:** In addition to arity, we can differentiate two main families of mechanisms regarding their closedness. The first one limits the definition of views as

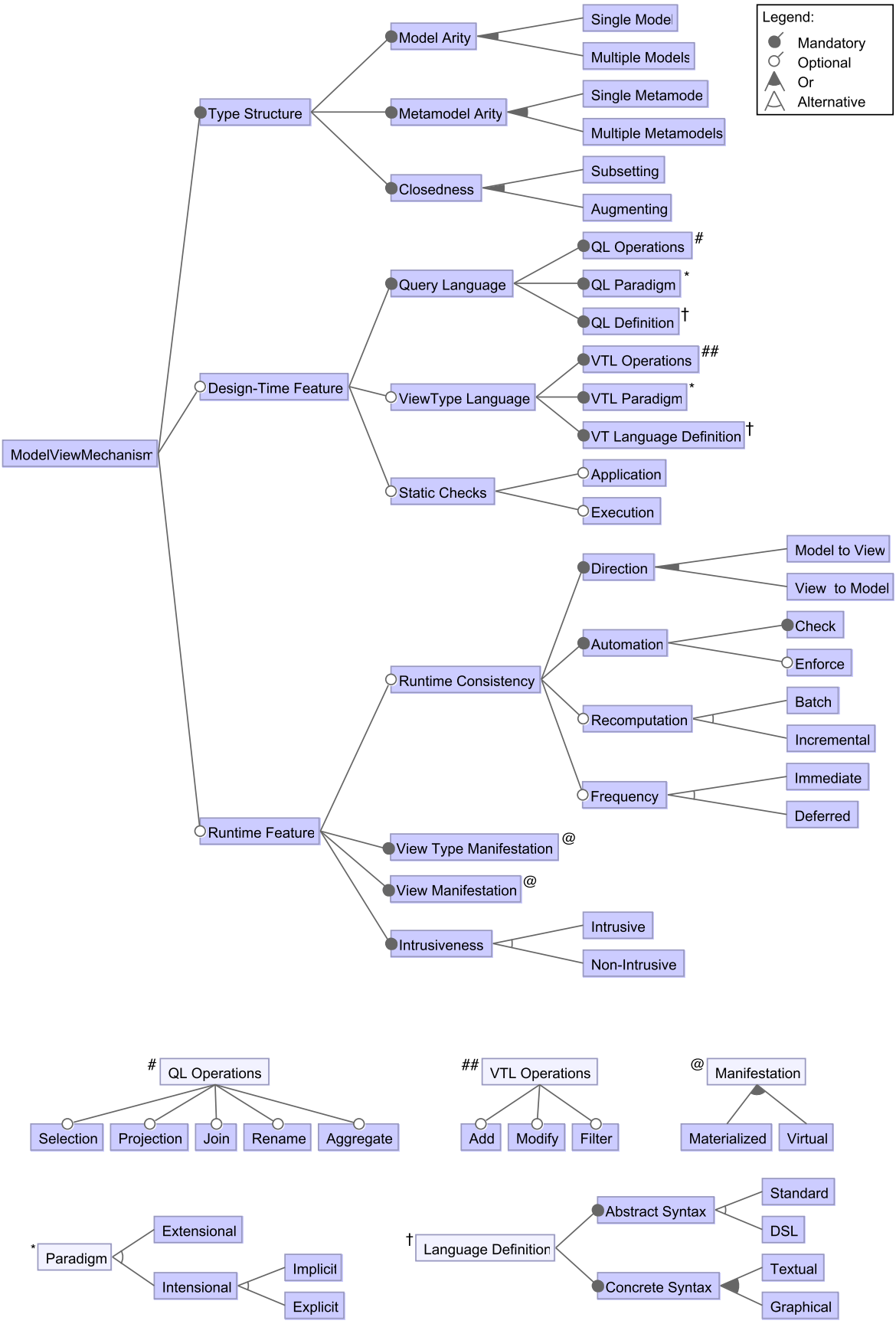

Figure 4.2 – A feature model for model view approaches.

subsets of the base model(s). The second one allows for views in which the content coming from the base model(s) can be augmented, e.g. with newly computed or manually entered data. Thus, the closedness is also related to the operations supported by the used viewtype and query languages (cf. the corresponding descriptions of these features). For instance a viewtype language can provide (or not) add operations, and a query language can come with some aggregate functions.

Design Time Features

As far as design time is concerned, we explicitly distinguish between two main kinds of languages (though they quite frequently appear mixed together in practice). The first kind of language is used to specify the viewpoints and build corresponding viewtypes, i.e. , to specify the metamodel for a new viewtype. We call it the *View Type Language*. The second kind of language is used for corresponding computation at view-level, i.e. to populate the view with the expected data. We call it the *Query Language*. We describe the properties of these language kinds in the following.

— **Viewtype Language:** It has an abstract syntax that can be either based on an already existing standard and/or general-purpose modeling language (e.g. , UML), or be implemented as a domain-specific modeling language (e.g. addressing more particularly a given set of concepts). As usual, its concrete syntax can be textual, graphical or eventually a mix of both. As said before, the viewtype definition itself can be directly connected to the used Query Language (cf. the explanation of this particular feature) or somehow independent from it.
Any given viewtype language globally follows a particular paradigm for expressing viewpoints: definitions can be made either extensionally or intensionally (and explicitly or implicitly in the latter case). A definition is considered extensional when the actual element types to be part of the corresponding view are listed exhaustively. On the contrary, it is intensional when it rather provides the properties/conditions needed for inferring the corresponding view (elements). When the viewtype definition is intensional and explicit, it typically uses a combination of core add/modify/filter operations to be applied on the base metamodels. In the case of an intensional and implicit definition, some underlying operations or conditions can be applied by default/systematically.

— **Query Language:** This same extensional vs. intentional distinction also applies to the Query Language. Indeed, queries can be defined (extensionally) by providing a fixed set of expected values or elements. They can also use (intentionally) more complex expressions, explicitly stated or implicitly called, implying further computations in order to retrieve the targeted elements.
In addition to that, and similarly to the viewtype language, a given query language can be based on a standard query language or equivalent (e.g. Structured Query Language (SQL) or OCL) or be a domain-specific query language targeting a particular domain or range of applications. The main capabilities of a given query language are provided by the operations it actually supports. Differently from the viewtype language, its operations focus on how the model-level data is selected and manipulated to be integrated into the resulting view (in a way that conform to the viewtype definition). Therefore, a first set of operations consists in reducing the scope of the original metamodel(s) by selecting and/or projecting only

some of the model elements in the produced view. A second set of operations is about complementing the view data by adding extra-information that can be derived from corresponding model elements (by join or aggregation) and/or manually added by a user (rename).

— **Static Checks:** Finally, different kinds of static checks can be performed at design time on the specified viewtypes and related queries. In some cases, they may concern the application of a given viewtype specification on the concerned base metamodel(s). In others, the pre-execution or parsing of the corresponding defined queries may be of interest. Two key properties of viewtypes arise here: (i) applicability (will the view computation ever return a non-empty set?) and (ii) executability (will the resulting view be ever consistent with the possible constraints defined at the viewtype-level?).

Runtime Features

From a runtime perspective, there are also different interesting aspects to consider for a model view mechanism. We describe them in the following.

— **Runtime Consistency:** There are several important consistency features to be considered when views are actually computed and handled. On one hand, synchronization can be supported in two main directions: from the model(s) to the view and from the view to the model(s). On the other hand, consistency verification can be more or less automated, following check or enforce strategies notably. The way the recomputation is performed can also vary. It can be made incrementally by updating only the concerned elements in the view, e.g. putting the full content of the view in this case.
The point in time when the recomputation is performed is described by the Frequency feature. Immediate recomputation means that the synchronization is triggered directly after each change to a view (or to one of its base models). Deferred computation means that synchronization is performed at defined points in time, but can span several editing steps.

— **View/Viewtype Manifestation:** There are two manifestation dimensions to be possibly taken into account: a given mechanism can provide support at viewtype-level (i.e. metamodel-level), at view-level (i.e. model-level), or at both levels of course. In all cases, the results can be actually realized differently. They can be concretely materialized, e.g. with the creation/duplication of actually new (meta)model elements, or virtual in the sense of only relying on proxies to already existing (meta)model elements. The used type of manifestation can have a direct impact on the the way the viewpoints/viewtypes and views are computed (cf. the description of the Runtime Consistency feature). For instance, synchronization can be made easier when using a virtualization approach instead of a duplication-based approach.

— **Intrusiveness:** Finally, a model view mechanism can also have different intrusiveness strategies, depending on whether or not it actually allows the alteration of the concerned base metamodels and models. Indeed some approaches do not modify the original (meta)models that can thus continue to have their own living, while some others do and directly impact the actual (meta)models content.

				Cicchetti	EMF Facet	EMF Profiles	EMF Views	Epsilon Merge	Epsilon Decoration	FacadeMetamodel	Kitalpha	ModelJoin	OpenFlexo	OSM	Sirius	TGGmv	TGGvv	VIATRA Viewers	VUML
Type Structure																			
Model Arity	Single Model			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Multiple Models						✓	✓			✓	✓	✓		✓			✓	✓
MM Arity	Single MM			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Multiple MMs						✓	✓			✓	✓	✓		✓			✓	
Closedness	Subsetting			✓			✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓
	Augmenting				✓	✓	✓		✓	✓	✓	✓			✓	✓	✓	✓	
Design Time Features														∅					
Query Language	QL Operations	Selecting		✓	✓		✓				✓	✓	✓		✓	✓	✓	✓	✓
		Projecting		✓	✓		✓				✓	✓	✓		✓	✓	✓	✓	✓
		Joining					✓	✓			✓	✓	✓		✓	✓	✓	✓	✓
		Rename					✓	✓			✓	✓	✓		✓	✓	✓	✓	✓
		Aggregation			✓		✓	✓			✓	✓	✓		✓	✓	✓	✓	
	QL Paradigm	Extensional			✓														✓
		Intensional	Implicit							✓									
			Explicit	✓	✓		✓	✓			✓	✓	✓		✓	✓	✓	✓	
	QL Definition	Abs. Syntax	Standard		✓										✓				✓
			DSL	✓			✓	✓			✓	✓	✓		✓	✓	✓	✓	
		Con. Syntax	Textual		✓		✓	✓			✓	✓	✓		✓			✓	✓
			Graphical	w									✓			✓	✓		
VT Language	VTL Operations	Add			✓	✓	✓		✓		✓		✓			✓	✓	✓	
		Modify					✓				✓	✓	✓						
		Filter					✓				✓	✓	✓						
	VTL Paradigm	Extensional		✓	✓	✓	✓	✓	✓		✓		✓		✓	✓	✓	✓	✓
		Intensional	Implicit																
			Explicit							✓		✓							
	VTL Definition	Abs. Syntax	Standard	✓		✓						✓				✓	✓		✓
			DSL		✓		✓	✓	✓	✓	✓		✓		✓			✓	
		Con. Syntax	Textual				✓	✓			✓	✓	✓					✓	
			Graphical	w	✓	✓			✓	✓			✓		✓	✓	✓		✓
Static Checks	Application					✓			✓	✓		✓	✓						
	Execution			✓			✓	✓		✓	✓		✓		✓	✓	✓	✓	✓
Runtime Features																			
Consistency	Direction	Model→View		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
		View→Model				✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Automation	Check		✓								✓		✓					✓
		Enforce				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
	Recomputation	Batch			✓		✓	✓			✓	✓			✓	✓	✓		✓
		Incremental		✓		✓			✓	✓			✓	✓		✓		✓	
	Frequency	Immediate		✓		✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	
		Deferred			✓			✓				✓							✓
VT Manifest.	Materialized			✓		∅		✓	∅	✓	✓	✓	✓	✓		✓	✓	✓	✓
	Virtual				✓	∅	✓		∅						✓				
View Manifest.	Materialized			✓		✓		✓	✓		✓	✓			✓	✓			✓
	Virtual				✓		✓			✓			✓	✓			✓	✓	
Intrusiveness	Intrusive													✓			✓		✓
	Non-Intrusive			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	

Table 4.1 – A comparison of existing model view approaches (✓=feature supported,∅=not applicable,w=Wizard).

4.1.3 Description of Model View Approaches

Applying the methodology detailed in the original journal publication [27], we selected a number of model view approaches which are particularly relevant in the context of our study. They are used as our basis for presenting the current state-of-the-art in this area. We describe each one of these approaches and summarize their main characteristics in what follows. To realize this, we notably relied on the terminology and feature model we introduced in earlier in this manuscript. Table 4.1 summarizes the overall results of our detailed study.

Cicchetti. This hybrid multi-view modeling approach [44] can be used to create sub-metamodels of existing Ecore metamodels and use them as customized view types. It is a hybrid approach in the sense that it uses the base models for the synchronization of the views (i.e. a *projective* approach [113]) and offers the creation of stand-alone models as views (i.e. a *synthetic approach* [113]).

The sub-metamodels, which are to be used as view types, are defined using a wizard

in the Eclipse development platform. These view types have to be consistent with the base metamodel in such a way that instances of the view types, i.e. the views, are also valid instances of the original metamodel. Thus, the approach is strictly closed. Synchronization mechanisms for maintaining consistency among the views are then generated automatically using a combination of Eclipse-based technologies. The mechanisms are based on model differencing. A difference metamodel is created automatically from the base metamodel and the viewtype metamodel. From these difference metamodels, model transformations are generated that synchronize the views with the models. Finally, an Eclipse plug-in is generated automatically, which provides an editor for the creation and manipulation of the views.

EMF Facet. EMF Facet [67] is an open source Eclipse/EMF-based tool, initially developed in the context of MoDisco (cf. Chapter 3.3) and then externalized. It provides a generic lightweight extension mechanism for existing EMF models. In particular, it allows to define so-called *facets* on given metamodels (i.e. Ecore models) in order to complement them with new computed types (concepts), attributes and references (properties). Modification and filtering of existing elements are not possible in the current version. The definition of a facet is stored in a specific and separate Facet model. If available for a given metamodel, a facet can then be applied on its corresponding models (e.g. within a model browser) in order to dynamically extend them at runtime. A facet can thus be seen as a view type that is specified over a single metamodel with the objective to extend it with newly computed information. When applied to given models, it allows to obtain corresponding views on them (i.e. extended models).

A core aspect of the approach is that it relies on a query abstraction framework that gives the possibility to plug any type of query (e.g. Java, OCL) to the facet so that the newly added elements can be computed automatically. In addition, another specificity of the approach is that everything is performed at runtime, meaning that the *extended* model (i.e. a view) is dynamically computed when the facet is applied on. The original model is not modified and the elements added in the view are only stored *virtually* in memory until the facet is unloaded or the model is closed.

EMF Profiles. EMF Profiles [132] is an approach and corresponding Eclipse/EMF-based prototype that provides another generic lightweight extension mechanism for EMF models. Directly inspired from what is already widely used in the UML world, it basically generalizes the concept of profiles to be used with all possible (meta)models. Thus, instead of allowing to extend only the UML metamodel, profiles can be defined on top of any Ecore model (i.e. any metamodel) and then applied on any corresponding instance (i.e. any corresponding model). A profile specifies a set of named stereotypes applicable only to selected types of elements and that come with their own additional properties. A profile can thus be seen as a viewpoint that is specified over a single metamodel with the objective to extend it with annotations on existing elements. When applied to given models, it allows to obtain corresponding views on them (i.e. extended models).

A core aspect of the approach is that it is intended to be used in a semi-automatic manner: a profile application is not computed automatically by default, instead the user triggers the application of stereotypes and many times the additional information stored in the view is added directly by the user. A given profile application is persisted as a

separate model in addition to the base model (in contrast to UML profiles). Thus, it can also be potentially computed by a model transformation if required and/or relevant in particular cases. Another interesting aspect of the approach is its capacity to define meta-profiles, i.e. profiles that are defined at meta-metamodel level. Thus, they are applicable on all models (independently from the metamodel they conform to) and allow to represent viewpoints which are directly reusable for different modeling languages.

EMF Views. EMF Views [33, 31] is an approach and corresponding Eclipse/EMF-based prototype that provides capabilities for specifying and obtaining views on top of models which potentially conform to different metamodels (cf Sections 4.2 and 4.3 for more details on this other main contribution from this manuscript).

Epsilon Merge. There is a first Epsilon-based approach [126] which provides a hybrid rule-based language (i.e. declarative rules with imperative bodies) for combining both homogeneous and heterogeneous models. As a merging approach, it allows to produce a new (merged) model out of several original models. Such a merged model can be considered as a view gathering elements coming from several different models, the merge specification in the Epsilon Merging Language (EML) thus acting as the viewpoint definition (of the merged metamodel) in this case. The overall merge process and its four consecutive steps (comparison, conformance checking, merging, reconciliation/restructuring) are actually realized thanks to the combined used of several Epsilon languages (respectively the Epsilon Comparison Language (ECL), Epsilon Validation Language (EVL) and EML for the last two steps).

A core aspect of this approach according to our present study is that it is merge-based. Thus, it covers only partially the set of capabilities generally expected when dealing with model views. For instance, basic synchronization between the original models and the generated merged model (i.e. the view) is not provided by default (even if some support can be implemented based on the merge trace model produced during the merge process).

Epsilon Decoration. There is a second Epsilon-based approach [128] that introduces a model decoration support enabling the annotation of models with additional information that is not necessarily supported by existing metamodel(s). A decorated model can be considered as a view on the original model complemented with some new manually added information. In the proposed solution, which differs from most other approaches, there is no explicit viewpoint definition and decorations are simply represented in a generic way, namely as tag/value pairs. Interestingly, the approach also proposes a bi-directional support for covering both automated decoration extraction from a previously annotated model (via its diagram) and decoration injection (or application) to a non-annotated model. This model decoration approach is implemented on top of the Epsilon framework, notably by reusing its Epsilon Model Connectivity (EMC) layer. The idea of having such a layer providing generic model loading, storing, querying and updating capabilities has been be directly inspired by existing relational database connectivity layers.

A core aspect of the approach is that it has the following two main characteristics: (i) non-intrusive as the original model is not modified and decorations are stored in a separate decorator model, and (ii) transparent from a usage perspective as decorations can be

handled manually and programmatically as if they were parts of the original model. However, the trade-off is that specific decoration injection and/or extraction transformations have to be specified for each decoration alternative (i.e. expressing a particular viewpoint).

FacadeMetamodel. The FacadeMetamodel [152] benefits from the realization that most DSLs are often just subsets of UML itself. It presents an approach to build new modeling languages by generating a kind of *facade* that looks like a pure DSL to the modeler. However, it actually uses UML in the back thus allowing full reuse of the UML infrastructure.

Given the extensibility and ease of use of this approach, the FacadeMetamodel may be used to build views on top of UML models by aliasing (renaming), refining, pruning and extending (using the UML profile mechanism) the UML language. The approach takes this customization model and generates an Ecore metamodel representing the new language/viewtype that can then be used to generate a custom user interface (e.g. in Papyrus [72]).

The new language appears as a complete pure DSL to end-users. Therefore, all modeling tools can use it transparently and it is automatically populated from the base UML models. Updates on the DSL level are propagated back to UML and stored as normal UML models. Updating those same models with standard UML tools is not recommended due to the possibility of inconsistent modifications.

Kitalpha. In the context of the PolarSys project [73], Kitalpha [78] is proposed as a framework to define model views inspired from the use of viewpoints in architectural modeling. Thus, the notion of view is very general in Kitalpha and covers the following points: abstract syntax, notations (such as icons), concrete syntax (textual and graphical), rules (e.g. check, transformation), services and tools. Furthermore, Kitalpha focuses also on reusing views by providing inheritance and aggregation for viewpoints.

A textual domain-specific language is provided to define these general building blocks of a viewpoint in Kitalpha. In addition, dedicated DSLs are provided to define the internals of the different building blocks. For instance, a language for defining the abstract syntax of viewpoints is available. With this language, metaclasses from base metamodels may be extended with more specific classes, new classes may be introduced, and additional references may be added as well.

ModelJoin. ModelJoin [34] is a DSL and tool for the creation of views on heterogeneous models (i.e. models that are instances of different metamodels). ModelJoin is a declarative language with a human-readable textual concrete syntax that bears similarities to that of SQL. The language is used to describe the desired properties of a view. It abstracts from the technical details of how views are created.

In relational databases, the table schema of the result of a query is dependent on the columns chosen by the projection operators. Transferred to modeling, this means that such a query not only defines the elements in a view, but also the kinds of elements which can be in a view, i.e. the metamodel of a view (or *view type*, cf. Section 4.1.1). Thus, the implementation of ModelJoin generates both a target metamodel and transformations that create target models from the input (base) models.

ModelJoin has been extended to support editability inside the views. For this purpose, OCL constraints that limit view editability are generated together with the views. These constraints define the possible edit operations for which the resulting views can be translated back to a valid base model. If no such translation is possible, the view may be adapted automatically with an automatic fix so that the resulting view is translatable. The prototypical implementation is based on Eclipse technologies such as Xtext/Xtend [77], and uses QVT-O [161] as the target language into which the transformations for the synchronization between base models and view are generated.

OpenFlexo. OpenFlexo [101] is a generic solution allowing to assemble and relate, without duplication, data coming from various kinds of data sources. Its main goal is to support the federation of data from heterogeneous technical spaces (EMF, XML, Web Ontology Language (OWL), Microsoft Excel, etc.) into the same conceptual space realized as a kind of virtual view. It comes with several components, including notably the Viewpoint Modeler and ViewEditor. The former is for specifying viewpoints that indicate how to mix together the different types of data. The latter intends to provide regular view visualization and editing capabilities, depending on previously specified viewpoints. To integrate these components, the solution also comes with an underlying model federation framework. This allows for homogeneous handling of data as models.

A core aspect of the approach is that it also provides the ability to define synchronized views on models, e.g. stored as EMF models. As soon as a virtual view is computed (from potentially multiple different models), the view is connected with the different base models. The other way round, there is a mechanism for indirectly connecting the different base models to the view. For instance, this allows propagating changes from one base model to other base model(s) via the common view. Furthermore, changes on the view elements can be propagated back to the corresponding base models.

OSM. The Orthographic Software Modeling (OSM) approach [10] aims at establishing views as first-class entities of the software engineering process. In the envisioned view-centric development process, all information about a system is represented in a Single Underlying Model (SUM). Even source code is treated as a special textual view. The OSM concept is based on three main principles: dynamic view generation, dimension-based view navigation, and view-oriented methods.

User-specific custom views are generated dynamically based on transformations from and to the SUM. These views are organized in independent (orthogonal) dimensions. Technically, a view is a model of its own, which also has a metamodel. Model-to-model transformations allow to create the views dynamically from the SUM. However, this requires that bi-directional transformations exist for every view type. Such transformations can provide the synchronization of the views with the SUM. In addition, edit operations can be propagated back to the SUM likewise. The complexity of a hub-and-spoke architecture like OSM is linear in terms of the number of transformations that have to be written and maintained. This notably contrasts with the quadratic number of transformations in a peer-to-peer synchronization scenario for views.

OSM also encompasses a development process with a *developer* role, who uses the generated views, and a role called *methodologist*, who creates the different view types along the orthogonal dimensions. A prototypical implementation of the OSM approach based

on KobrA has been developed [7]. It relies on UML and OCL, and offers the dimensions abstraction (defined by notions of model-driven development), variability (defined by notions of product line engineering), compositionality (defined by notions of component-based development), encapsulation (e.g., public/private) and projection (structural/operational/behavioral).

Sirius. As mentioned in Chapter 2.3, Sirius [74] is an open source Eclipse/EMF based tool that is intended to facilitate and speed up the construction of industrial graphical modeling solutions. Considering the domain metamodels which are key assets inside companies, and thus, cannot be changed or modified easily, it allows to specify different concrete representations on top of them expressing different viewpoints (for different stakeholders). Within Sirius, a given Viewpoint Specification Model (VSM) defines a set of logically organized representations which are graphical constructions to represent the actual data (i.e. the original model). Such representations can be diagrams, tables, matrices or trees. A viewpoint specification also describes how the different elements from the original metamodel(s) should be actually mapped to the various representations. These mappings can go from basic ones (e.g., one concept/one box in a diagram) to complex ones (resulting from complex query computations, e.g. defined in the Acceleo Query Language (AQL)). Thus, applying such a viewpoint, different graphical views on the same model(s) can be provided to different types of users/stakeholders.

A core aspect of the approach is the clear separation between the metamodel that is conceptual and the viewpoint that is considered as a purely representation asset (mostly in the sense of graphical representation). Another aspect is that the provided views can be made completely editable if needed, the required editing capabilities being also specified within the Viewpoint Specification Model. In this respect, Obeo Designer Team (as a commercial extension of Sirius) comes with more powerful additional features dealing with collaborative editing of such views and the management of related conflicts.

TGG-based approaches. Triple Graph Grammars (TGGs) [183] have been proposed as a means of specifying a consistency relation between two graph languages. TGGs are a well-studied formalism to define bi-directional transformations including not only the back and forth translation of models, but also the comparison of models and their synchronization. Thus, they are also applicable as a base mechanism to solve model view problems. In particular, two TGGs-based approaches have been presented in the past for defining views. Both *TGGvv (virtualized view)* and *TGGmv (materialized view)* are supported by the EMF-based, bootstrapped eMoflon model synchronization tool [134]. In the following, we summarize both approaches.

A first approach of using TGGs for model views (TGGvv) has already been proposed [114]. In this approach, the base metamodel is aligned with the viewtype by defining a correspondence model between them. Based on these three models, models conforming to the base metamodel can be filtered to yield virtual views conforming to the viewtype. The advantage of this approach is that the views are virtual in the sense that elements of the base model are simply interpreted as elements of a view. This simplifies synchronization tasks in many cases. The approach, however, requires non-trivial changes to the base metamodels and has not been fully formalized for complex metamodels or multiple views on the same base metamodel.

TGGvv has been complemented by a recent TGGs-based approach for materialized views (TGGmv) [5]. It notably allows for separate view models without requiring any changes to the base metamodels. This approach provides a formalization of View TGGs, as a restriction of existing TGGs theory to the special asymmetric case of view specification (as TGGs are symmetric in general). It shows that the chosen restrictions can be suitably exploited to enable highly efficient view synchronization.

VIATRA Viewers. The VIATRA Viewers [75] approach emerged from EMF IncQuery [200], a framework for performing incremental model querying based on the RETE algorithm. The main advantage of EMF IncQuery, compared to many other model query approaches, is that queries do not have to be entirely evaluated in case model changes happen (but only queries which are actually concerned by the changes). Of course, this capability plays also an important role for model view approaches. Thus, EMF IncQuery can be extended for dealing with model views [52]. In particular, the EMF IncQuery language is extended by so-called derivation rules which are defined with annotations to EMF IncQuery query patterns. Via the annotation of derivation rules, the viewtypes are defined and the query patterns are used to populate the views by simply reusing the EMF IncQuery support.

In addition to the views, trace models are computed between the views and the base models. These trace models are used to reason about changes of query pattern matches which allows to build synchronization support. In particular, if changes to the base models are performed, the query pattern matches are changed. This stimulates incremental updates in the view models. In recent work, the propagation of view change to the base models is also considered [185]. Finally, it has to be mentioned that VIATRA Viewers allows to build chain views, i.e. to build views on views.

VUML. VUML [148] is a UML profile that supports view-based modeling for the UML family of modeling languages. The proposed methodology includes the definition of *actors*, which all possess a unique viewpoint, and a design process that supports the actor concept. The design process is aligned with the OMG MDA approach [158]. The viewpoint models which are tied to these viewpoints are UML class diagrams. They are composed into a VUML system model. The composition algorithms have been implemented in a composition metamodel and transformations in ATL [116].

The application of a viewpoint on a specific system is called a view. VUML provides the concept of *Multiviews Components*, which consist of both default base views (that can be used by all actors) and actor-specific views (that are connected to the base view via an extension relation). The actor concept can be used to structure the views and to implement access control. The semantics of VUML are described by a metamodel and textual descriptions. The views of a MultiView Component can have dependencies between each other, which are expressed with OCL constraints.

4.1.4 General Challenges for the Community

Several interesting findings can be made from the aggregated Table 4.1 resulting from our previous evaluation of different model view solutions. There are some commonly

shared aspects: for instance, all evaluated approaches require an explicit definition of the viewtype, and do not offer the possibility of deriving it on the fly when computing the view. Also, for each feature there is at least one approach covering it, showing the variety of existing solutions. Still, most of these solutions focus on a reduced set of features. Thus, we are still missing more general solutions that can be applicable in a broad number of scenarios.

Moreover, from what we have been able to observe so far, there are currently not many proofs that the existing solutions do scale up in the context of very large models. This can be considered as an important issue in this model view area, as scalability is a key element in related challenges such as view update or incremental maintenance for instance (as further explained in the remaining of this section).

Table 4.1 presents some features that seem to be more challenging since very few approaches provide support for them. A typical missing feature is the verification support for viewpoint/view definitions. Most approaches do not support designers in the specification of the viewpoints and underlying viewtypes. For instance, they do not alert the users regarding the applicability or executability of these definitions to build actual views. The few that somehow ensure these properties are doing it more as a side-effect, because the underlying view mechanism itself is strict enough to prevent from some possible issues. Moreover, graphical languages to express queries are rarely used, while they could be useful to allow less technical users defining their own viewpoints/views (as tools like graphical query builders for databases have proved to be). All these potential features open doors for more interesting research in the future.

Beyond these mentioned features, we would like to highlight a few more research challenges worth to be investigated in the coming years within our community. We plan to address some of them in the context of our model view solution as presented in this Chapter (cf. also Section 5.4.2 for more information on possible related future work). Some of these challenges are actually well-known recurring problems in any technical space where views are used in practice (e.g. in the database domain). Some others are more specifically related to our modeling context and have been identified by studying deeper the content of Table 4.1.

— **Terminology inconsistencies**. A different vocabulary is employed within the papers from the literature we studied for this review. For instance, there has been approaches defining model composition as the combination of two successive mappings into one [141]. In other cases [41], model composition refers to a specific model integration scenario where models with running interacting features are assembled. This also causes comparisons and building on top of existing approaches to become more difficult. This is especially evident for papers targeting different levels in the modeling stack (cf. Section 2.1.2). Even papers that use the same type of techniques and share a similar conceptual goal may use a very different terminology (e.g. DSL combination vs model merging). This makes them quite often ignorant of each other.
— **View updating problem**. In the general case, fully updating a view is not always possible [138]. This will notably depend on the kind of modifications applied, and on the operators used to compute the view. Indeed, some combinations may not result in a deterministic translation of the view update into a set of modifications on the base model elements. As a simple example, imagine a model view that

displays the average value of a certain attribute from a given class. An update of this average value does not have a single way to be propagated back to the individual values from the base models: Should we proportionally increase all of them? Or rather assign the whole increment to a single one? A pragmatic solution would be to provide a uniform way to support several different model view update strategies. For instance, a listener on the model view could capture the update events and deals with them according to the instructions provided by the designer (e.g. similarly to the concept of INSTEAD-OF triggers as available in SQL). As we have observed, current solutions follow a more conservative approach where they basically restrict the changes as soon as they become complex to handle.

— **Incremental view maintenance**. For those approaches where (some of) the model view elements are automatically computed, a major problem is to incrementally update them after changes on the base models. As seen in Table 4.1, current solutions typically ignore or provide very limited support to this feature. Always completely recomputing the whole view may be too costly and/or trigger undesirable side-effects in some cases. As introduced in the previous item, specific view update strategies could be implemented to provide the needed incremental support to deal with such scenarios. As a possible technical solution, they could rely on incremental model transformation techniques [117].

— **Concrete syntax generation**. The (direct or indirect) definition of the abstract syntax of the view type is a key element in most approaches. Nevertheless, most of them do not offer explicit support to specify the concrete syntax part (as seen from Table 4.1). To make model views more easily usable by end-users, we should be able to display the view content graphically (and not just show it using default tree-like browsers). In order to achieve this, we could generate a default concrete syntax based on the concrete syntax(es) associated to the base metamodels/languages. We could also manually build one explicitly for a given viewtype. The graphical-oriented approaches proposed by solutions such as Sirius or Kitalpha are good examples to follow regarding these aspects.

— **Security aspects**. Views are typically used as a security mechanism to prevent people from accessing data they are not authorized to see and/or modify. This requires the availability of an access-control mechanism that enables designers to give read/write permissions (on specific model views) to particular categories of persons. Such a mechanism notably allows preventing them from accessing directly to the base models when not appropriate. As observed during our study, this is in general a green area for the modeling community. However, it makes sense to tackle related challenges once the specification of views/viewpoints on models comes into play. For instance, many approaches provide profiles or DSLs to annotate models with security characteristics of the system being modeled. Until now, they do not allow assigning explicit permissions to the model access itself. This could be possibly addressed by the use of model views in such contexts.

4.2 Proposed Conceptual Approach

This section presents the global model view approach we propose in order to support the building and handling of views over different heterogeneous models. This approach notably intends to potentially overcome some of the general challenges that have been

identified in previous Section 4.1.4 (cf. what follows in this section, as well as Section 5.4.2 for more future work). Quite similarly to the process we followed in the context of the first main contribution described in this manuscript (cf. Chapter 3), our objective is:

1. To identify the main steps and components commonly used in model view solutions.
2. To combine them coherently as a generic approach.

In what follows, we start by providing an overview of our approach in Section 4.2.1, explaining both the viewpoint creation phase happening at *Design Time* and the view initialization phase occurring at *Runtime*. After that, we give in Section 4.2.2 more insights on the core virtualization (weaving) metamodel our approach is based on. Then, we describe in Section 4.2.3 a couple of DSLs we defined on top of it in order to facilitate the specification of model views in two different contexts. We also present in Section 4.2.4 how we propose to integrate existing model persistence solutions within our approach in order to improve its overall scalability. Finally, we end by summarizing the main benefits of our approach in Section 4.2.5.

4.2.1 Overall Approach

From our previous analysis of the state-of-the-art in the area (cf. Section 4.1), we have been able to observe that there is no generic model view solution that can be easily adapted to a broad number of scenarios (that could benefit from the use of model views). This is particularly true when such scenarios may imply to consider many different sets of possibly heterogeneous and (very) large models. To this sense, we believe an ideal model view approach must provide the following characteristics:

— **Genericity**: the view mechanism should be applicable for all modeling languages (i.e. metamodels and models), possibly combining several of them in a single view.
— **Expressivity**: a user-friendly interface should be provided to support the view specification, e.g. via a *select-project-join*-like syntax or DSLs relevant in the targeted contexts.
— **Non-intrusiveness**: the view mechanism should be applicable without requiring to change the modeling languages used in the already existing processes or architectures to be tooled.
— **Interoperability**: a view should be perceived and possibly (re)used as any regular model, both from the user and tool perspectives.
— **Modifiability**: a view should be changeable as any regular model is (eventually depending on corresponding access control policies already in place).
— **Synchronization**: changes in base models should be directly reflected in the views, and vice versa (eventually according to some view update strategies to be defined and/or just reused).
— **Scalability**: the view creation and manipulation time should be sufficiently limited (from a usability perspective), this should also be true for the corresponding memory usage (from a technical resource perspective).

Intending to fulfill such characteristics (partially for some of them at the current state, cf. Section 4.2.5), our approach targets the definition and building of views on any possible set of interrelated models that conform to potentially different metamodels. To this

end, we have been able to generally observe two main complementary phases in (model) view solutions:

— **Viewpoint Creation at Design Time**: a specification is provided and allows to obtain a corresponding viewpoint/viewtype (according to the nature of the approach).

— **View Initialization at Runtime**: the previously obtained viewpoint/viewtype is used in order to (at least semi-)automatically compute a version of the view.

As a consequence, our generic approach quite naturally also relies on such a two-step approach. However, unlike other approaches, we propose:

1. To explicitly separate the specification of viewpoints from the realization/handling of corresponding views.
2. To apply a same building mechanism at both levels.

We decided to rely on the unification power of models and on the capabilities offered by the related model-based techniques (cf. Chapter 2). Notably, in what follows we emphasize on two particular kinds of models that are fundamental in our approach. These also complement the general terminology already presented in Section 4.1.1:

— A **virtual model** is a model whose (virtual) elements are just proxies to actual elements contained in other models. The same approach is also applicable at metamodel-level, i.e. a **virtual metamodel** is a metamodel whose (virtual) elements are just proxies to actual elements contained in other metamodels.

— A **weaving model** is a model that describes links between and/or pointing to elements coming from other different models. It conforms to a weaving metamodel that specifies the types of relation that can be represented at (weaving) model-level. Such a weaving model can also be used at metamodel-level, i.e. it can describe links between and/or pointing to elements coming from other different metamodels.

Motivating Example

Figure 4.3 provides a basic illustration of viewpoint and view, considering two easy-to-understand metamodels (named *Book* and *Publication*) as well as two corresponding models that conform to them (respectively). We want to define a viewpoint relating together these two metamodels via a new *bookChapters* relationship from any *Publication* (from the *Publication* metamodel) to associated *Chapter*s (from the *Book* metamodel). In addition, we also want to filter the *nbPages* property in these *Chapter*s (still from the *Book* metamodel). Using this viewpoint, we then want to obtain a view that combines corresponding *Book* and *Publication* models accordingly. Thanks to such a view, from any given *Publication* (e.g. *ATL in Depth* in our example) we can transparently access to the related *Chapter*s as if all these elements were actually coming from the same model, and see only the authorized properties (e.g. *title* from *Chapter* in our example).

Overview of the Approach

In order to realize the previous simple example, a model view can be considered as a set of proxy elements, which point to concrete elements from the base models referenced in the view, plus some newly added relationships between them (and eventually

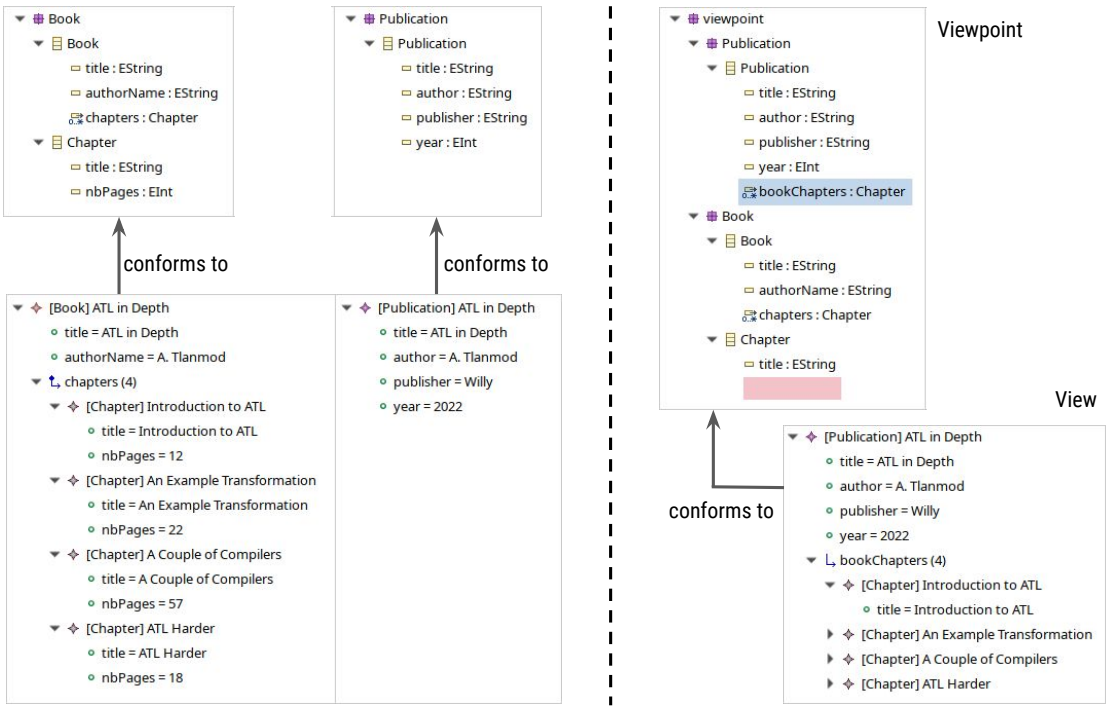

Figure 4.3 – A simple example of model viewpoint and view.

other new elements or filtered information). We base our solution on a model virtualization approach that is deployed similarly at both metamodel- and model-levels. Thus, views are actually virtual models that act transparently as regular models via proxies to these interrelated models, but do not duplicate any of the already available data. Each view "conforms to" a particular viewpoint, which has been previously specified from one or several corresponding metamodels (interconnected together) as a virtual metamodel. Interestingly, the fact that both viewpoints and views are actually virtual (meta)models behaving as normal (meta)models allows for easier viewpoint/view handling and reuse. An overview of the proposed approach is shown on Figure 4.4.

At *design time*, designers may specify a new viewpoint by choosing the concerned metamodel(s), listing the relations she/he wants to represent between them (as well as indicating how to eventually compute them at view-level, see hereafter), and identifying the concepts and properties to be selected. This required information is directly collected from the designer/architect, either manually or using a DSL (cf. Section 4.2.1). This input data is stored in a weaving model that is then used by the virtualization mechanism to obtain the actual viewpoint. Therefore, the original metamodel(s) are not modified or polluted by the viewpoint definition. This results in a virtual metamodel, representing the viewpoint, that aggregates several different metamodels according to the given specification.

Similar to the *select-project-join* operations in relational algebra, the viewpoint definition mostly specifies what types/attributes from the contributing metamodels should be part of (or, conversely, filtered out from) the view (*projection*), what conditions model elements will need to satisfy in order to appear as a result in the view query (*selection*) and how the elements from different models should be linked when computing the actual view (*join*).

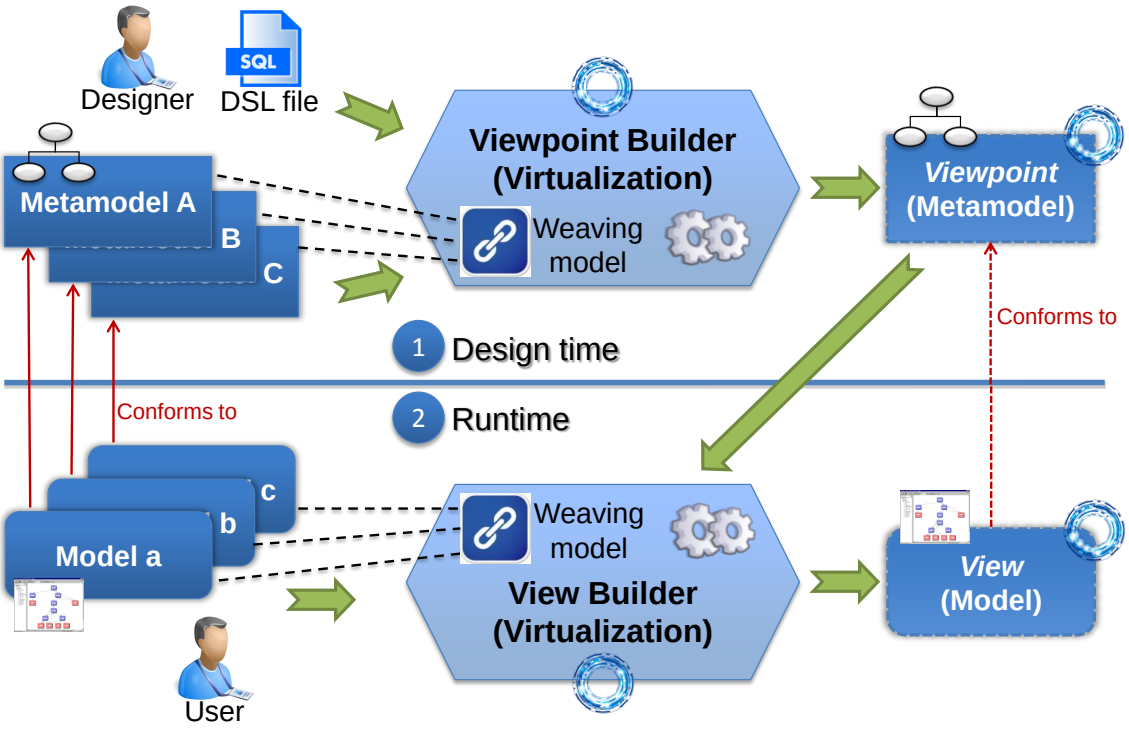

Figure 4.4 – Overview of the Model View approach.

At *runtime*, once the viewpoint is specified, the user can work on querying and handling views that "conform to" it. To obtain such a view, she/he can choose the set of input models to be used as input data for the view (and that "conform to" their respective metamodels, themselves used to create the given viewpoint). With those models and the input viewpoint, the proposed approach can build up the corresponding view. As described before, the view is represented as a virtual model. In order to create the view, new links have to be established between the underlying models. These links are computed and initialized from the rules expressing the combination of the corresponding metamodels at the viewpoint-level (though a manual modification by the user is also possible when needed) by means of a matching engine (cf. Section 4.2.1). The links are stored in a separate weaving model associated with the view, without altering the original models neither.

An important point of our approach is that a same virtualization technique has been adapted slightly differently at *design time* and *runtime*. More details on these two levels can be found in what follows in this section. Another fundamental aspect is the use of weaving models in order to create and store the viewpoint/view-specific information. Thus, our approach heavily relies on a common and generic virtualization (weaving) metamodel we propose in Section 4.2.2. From a usability perspective, it is needed to provide proper interfaces for designers/users of our solution. As a consequence, we contribute in Section 4.2.3 two DSLs covering two different applications of model views (as well as two corresponding subsets of our core virtualization metamodel). Finally, from a scalability perspective, it is required to be able to load, store and access to all the involved metamodels and models (i.e. contributing ones, weaving ones) in a efficient way. Section 4.2.4 describes how we propose to realize this by integrating the use of existing model persistence solutions.

Viewpoint Creation at Design Time

At *design time* as shown on Figure 4.5 (i.e. when specifying the viewpoint), the designer or architect first needs to provide the viewpoint definition data.

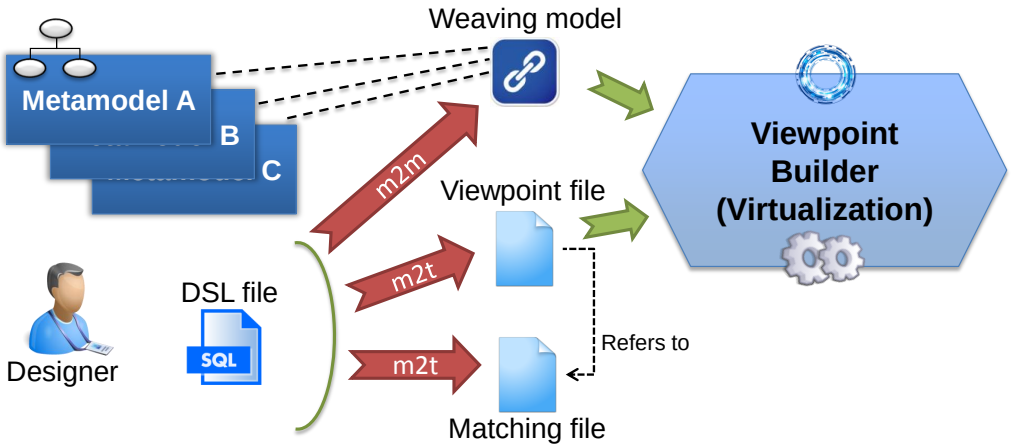

Figure 4.5 – Viewpoint creation at design time.

To this intent, he/she can use one of the provided DSLs (cf. Section 4.2.3). From the collected information, our approach can produce (by means of model-to-model or model-to-text transformation) three different artifacts:

— A **Viewpoint file** that lists general information about the viewpoint, i.e. contributing metamodel(s), weaving model, matching file.
— A **Weaving model** that contains the links between the connected elements from different metamodels, links to filtered elements, etc.
— A **Matching file** that expresses the matching rules to be applied at view-level in order to compute inter-model links (cf. Section 4.2.1).

Both these Viewpoint file and Weaving model (in addition to the contributing metamodels of course) are actually used as inputs for the virtualization mechanism to create the (virtual) metamodel representing the defined viewpoint.

View Initialization at Runtime

At *runtime* as shown on Figure 4.6 (i.e. for the view to be actually realized), the user also needs to provide some additional information.

He/she has to list the contributing models to be used to produce the view according to the chosen viewpoint. These models have to conform to the contributing metamodels of the viewpoint. From this information, our approach can produce two different artifacts:

— A **View file** that lists general information about the view, i.e. contributing models, weaving model, viewpoint definition (i.e. Viewpoint file).
— A **Weaving model** that contains the actual links between the connected elements from different models, the links to filtered elements, etc.

The weaving model can be initialized automatically using a matching engine that reads the viewpoint definition and uses the content of the referenced Matching file to decide how to join the elements in the different models. In this case, it may exist only in memory and may not be serialized as an actual file. However, this weaving model may also

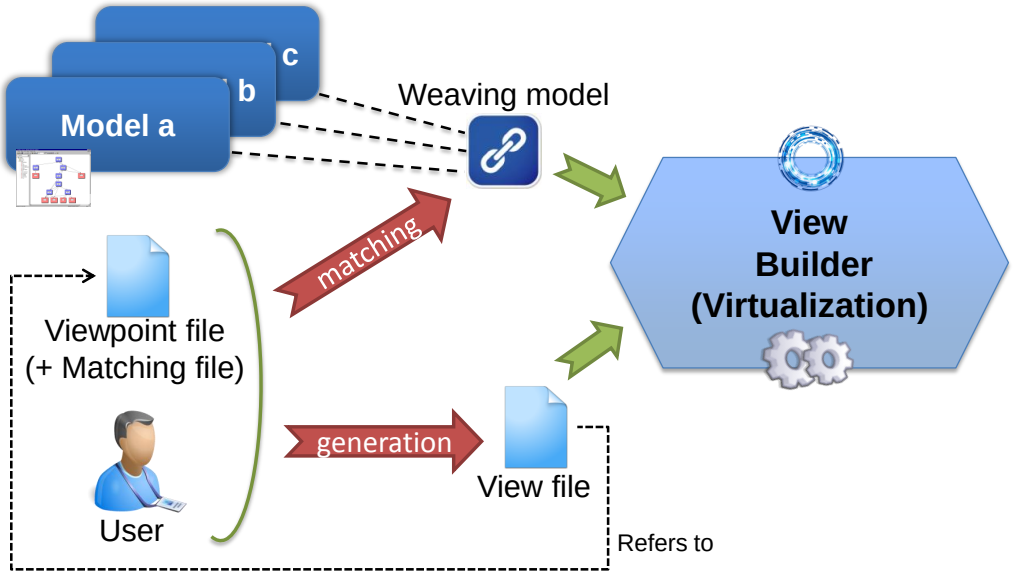

Figure 4.6 – View initialization at runtime

be completed manually or via other tools if needed (e.g. from the execution of a model transformation). Both the View file and the Weaving model (in addition to the contributing models of course) are used as inputs for the virtualization mechanism to produce the resulting view as a virtual model.

4.2.2 Core Virtualization (Weaving) Metamodel

As described before in Section 4.2.1, our approach relies on the use of weaving models in order to represent and store the required viewpoint/view-specific information. These weaving models are then consumed to produce the defined viewpoints and views as virtual metamodels and models (respectively). Thus, a dedicated metamodel is needed in order to specify the nature of the data these weaving models have to deal with. As said earlier, we intend to use a similar building mechanism at both viewpoint (i.e. metamodel) and view (i.e. model) levels. As a consequence, we made the choice of designing a generic core virtualization metamodel for our approach, as depicted in Figure 4.7 [1].

*VirtualLink*s are the core elements of our metamodel. They actually represent the viewpoint/view-specific information that cannot be obtained from the base metamodels/-models. There are four different types of *VirtualLink*:

— A *VirtualConcept* is a concept that only exists in the viewpoint/view. As any regular concept , it can inherit from other concepts (also virtual or concrete).

— A *VirtuaProperty* is a property that only exists in the viewpoint/view. It is attached to the concept (virtual or concrete) it belongs to and can be set as optional. As any regular property, it has a primitive type (integer, boolean, string, etc.).

— A *VirtuaAssociation* is an association that only exists in the viewpoint/view. As any regular association, it connects together source and target concepts (virtual or concrete). It also has a cardinality that can be determined by setting its upper

1. The complete Ecore version of this metamodel is available from https://github.com/atlanmod/emfviews/blob/master/plugins/org.atlanmod.emfviews.virtuallinks/resource/VirtualLinks.ecore

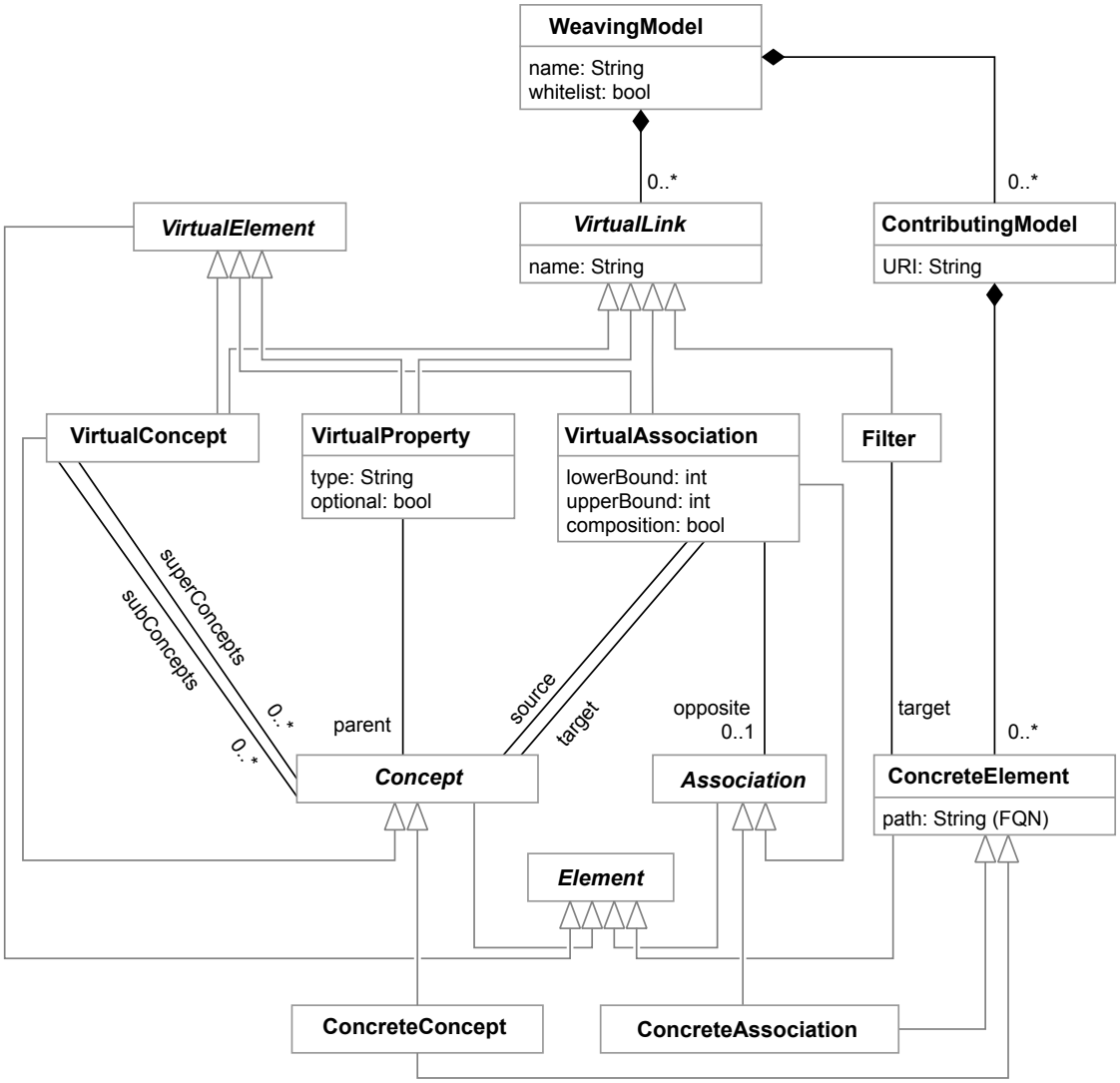

Figure 4.7 – Core virtualization metamodel of our approach.

and lower bounds. It can be a standard association or a composition one, thus introducing a containment notion between the connected concepts. Finally, it can have an opposite association that allows for bi-directional navigation between the connected concepts.

— A *Filter* is a declaration of inclusion or exclusion of concrete elements coming from the base metamodels/models (cf. the explanations on the *WeavingModel* and attached filtering strategy in the next paragraphs).

*VirtualLink*s all have a name which is used for the corresponding virtual feature name in the resulting viewpoint/view, except for *Filter*s where the name is ignored as not necessary.

*ContributingModel*s are proxies to the metamodels or models that are considered in the viewpoint or view (respectively). A *ContributingModel* aims at holding the *ConcreteElement*s that are required by the different *VirtualLink*s in the viewpoint/view. It has a URI that always refers to the corresponding metamodel namespace (even if at view/model-level).

*ConcreteElement*s are proxies to elements coming from the base metamodels/models

contributing to the viewpoint/view. They all have a path that allows pointing to and retrieving the actual metamodel/model elements. At viewpoint-level, it is the fully qualified name of the corresponding element (not including the metamodel name as already given by its container *ContributingModel*). At view-level, it is an implementation-specific identifier (e.g. in our EMF-based implementation described in Section 4.3, it is obtained by a call to the *Resource.getURIFragment* method). In addition, *ConcreteElement*s can be more precisely characterized as *ConcreteConcept*s or *ConcreteAssociation*s. The distinction is needed in these two cases (and not in the case of concrete properties) because a *VirtuaProperty* can be attached to a *ConcreteConcept* and a *VirtuaAssociation* can have a *ConcreteAssociation* as an opposite.

All the *VirtualLink*s, *ContributingModel*s and *ConcreteElement*s (via their respective *ContributingModel*) are contained in a *WeavingModel* root element. This root can have a name that is used at viewpoint level (only) in order to specify the namespace URI. The filtering strategy is also specified at viewpoint level via the *whitelist* boolean property. If the flag is set to false (default value, i.e. blacklist), the view will include all the elements coming from the contributing models unless they are explicitly filtered out. If the flag is explicitly set to true, the view will include no element unless they are explicitly filtered in.

4.2.3 Viewpoint/View Specification DSLs

We want to show in practice the genericity and expressivity of our approach (as presented in Section 4.2.1) as well as of its core virtualization metamodel (as presented in Section 4.2.2). Thus, we studied a couple of possible contexts where the use of model views can be particularly relevant:

1. The basic specification of views on models, in a similar way to what can be done in the database world.
2. The definition of metamodel extensions, as viewpoints on existing metamodels.

In what follows in this section, we describe the two DSLs resulting from this first work. These two DSLs, covering two different applications of model views, also correspond to two different subsets of our core virtualization metamodel. For instance, the latter allows to add new concepts in the built views (i.e. concepts not coming from any of the contributing models) while the former does not. For both languages, we present the main provided operations as well as their respective concrete textual syntaxes. It is interesting to note that, in each case, the semantics of the language is provided by a direct mapping to our core virtualization metamodel (abstract syntax) and related viewpoint/view building mechanism.

ViewPoint Description Language (VPDL)

In order to facilitate the definition of viewpoints and related views, we proposed a first DSL directly inspired from the very well-known SQL language. This DSL is named ViewPoint Description Language (VPDL). The choice of an SQL-like language was quite natural since SQL has already proved its relevance to deal with similar view problems in the database community. It notably also allows expressing the main needed operations in our model view context, i.e. SELECT, PROJECT, and JOIN. Moreover, considering such

a widespread language as a base language for our DSL intends to facilitate its adoption by potential future users.

Listing 1 – Grammar of the ViewPoint Definition Language (VPDL).

```
View: K_CREATE K_VIEW name=ID K_AS
  select=Select
  from=From
  where=Where?;

Select: K_SELECT features+=SelectFeature (',' features+=SelectFeature)* ','?;

SelectFeature: metamodel=[Metamodel] '.' class=[Concept] rest=SelectFeatureRest;

Concept: name=ID;

SelectFeatureRest: '.' features+=Attribute
  | features+=Relation
  | '[' features+=Feature (',' features+=Feature)* ','? ']'
  | features+=AllAttributes;

Feature: Attribute | Relation;

Attribute: name=ID;

Relation: K_JOIN metamodelRight=[Metamodel] '.' classRight=[Concept] K_AS name=ID;

AllAttributes: wildcard='.*';

From: K_FROM metamodels+=Metamodel (','metamodels+=Metamodel)* ','?;

Metamodel: nsURI=STRING K_AS name=ID;

Where: K_WHERE rules+=Rule (','rules+=Rule)* ','?;

Rule: condition=STRING K_FOR relation=[Relation];

K_CREATE: 'create' | 'CREATE';
K_VIEW: 'view' | 'VIEW';
K_AS: 'as' | 'AS';
K_FOR: 'for' | 'FOR';
K_JOIN: 'join' | 'JOIN';
K_FROM: 'from' | 'FROM';
K_WHERE: 'where' | 'WHERE';
K_SELECT: 'select' | 'SELECT';
```

Listing 1 presents the grammar of our textual DSL and highlights its four main language features [2]:

— **Create view** (mandatory): It defines the name of the viewpoint.

— **Select** (mandatory): It specifies the contributing metamodel(s)' features (i.e. properties or associations) to be included in the viewpoint, as well as the new virtual associations to be created if any (using a **Join**). To include all features of a selected concept, * is used. Note that this DSL realizes a whitelist filtering strategy (cf. Section 4.2.2).

— **From** (mandatory): It enumerates the different contributing metamodels as well as their respective aliases to be used in the remainder of the viewpoint definition.

— **Where** (optional): It expresses the rules used for populating new virtual associations by navigating the contributing models at view-level only (contrary to the three other clauses that also operate at viewpoint-level). For genericity reasons, the current version of the language simply considers rules as full strings (expressed in

2. The complete Xtext version of this grammar is available from https://raw.githubusercontent.com/atlanmod/emfviews/master/dsls/vpdl/org.atlanmod.emfviews.vpdl/src/org/atlanmod/emfviews/vpdl/Vpdl.xtext

any relevant querying/matching language). It does not parse or interpret any of these rules at VPDL-level, they are just copied into a Matching file associated to the Viewpoint file (cf. Section 4.2.1).

Listing 2 – Example of viewpoint specification in VPDL.

```
CREATE VIEW EntrepriseArchitectureExample AS

SELECT
    togaf.Requirement[statementOfRequirement, acceptanceCriteria],
    togaf.Process.isAutomated,
    togaf.Element.name,
    togaf.EnterpriseArchitecture.architectures,
    togaf.StrategicArchitecture.strategicElements,
    togaf.BusinessArchitecture.processes,

    bpmn.Definitions[name, rootElements],
    bpmn.CallableElement.name,
    bpmn.Process[isClosed, isExecutable, processType],

    reqif.ReqIFContent.specObjects,
    reqif.SpecObject.type,
    reqif.ReqIF.coreContent,
    reqif.Identifiable[desc, longName],

    togaf.Requirement JOIN reqif.SpecObject AS detailedRequirement,
    togaf.Process JOIN bpmn.Process AS detailedProcess

FROM
  'http://www.obeonetwork.org/dsl/togaf/contentfwk/9.0.0' AS togaf,
  'http://www.omg.org/spec/BPMN/20100524/MODEL-XMI'       AS bpmn,
  'http://www.omg.org/spec/ReqIF/20110401/reqif.xsd'      AS reqif

WHERE 's.name=t.name␣and␣s.isAutomated␣=␣false' FOR detailedProcess,
      't.values.exists(v␣|␣v.theValue=s.name)'  FOR detailedRequirement
```

To illustrate better VPDL, Listing 2 shows a simple example of a viewpoint specification (from the TEAP collaborative project, cf. . Section 5.2.2). It selects and aggregates some elements (*SELECT* part) from base metamodels (*FROM* part) and establishes new associations between them (*WHERE* part).

Metamodel Extension Language (MEL)

In order to make easier the definition of metamodel extension, we provided another textual DSL that offers an initial list of extension operators. This DSL is named Metamodel Extension Language (MEL). The proposed syntax is intended to be intuitive and easy-to-learn for people already familiar with (meta)modeling. Having genericity and portability in mind, it has also been defined independently from any particular metamodel or modeling framework/environment.

Listing 3 – Grammar of the Metamodel Extension Language (MEL).

```
Model:
  ('import' imports+=Metamodel)*
  'define' extensionName=ID 'extending' metamodels+=[Metamodel]
  ("," metamodels+=[Metamodel])* '{' extensions+=Extension* '}';

Metamodel: name=ID 'from' nsURI=STRING;

Extension: AddClass | ModifyVirtualClass | ModifyClass | FilterClass;

AddClass:
  'add' 'class' name=ID
  ('specializing' parents+=TargetClass ("," parents+=TargetClass)*)?
  ('supertyping' children+=TargetClass ("," children+=TargetClass)*)?;
```

```
TargetClass: ConcreteClass | VirtualClass;

ConcreteClass: metamodel=[Metamodel] '.' class=[ecore::EClass];

VirtualClass: class=[AddClass];

ModifyVirtualClass:
  'modify' 'class' class=VirtualClass '{'
    operators+=ModifyVirtualClassOperator*
  '}';

ModifyClass:
  'modify' 'class' class=ConcreteClass '{'
    operators+=ModifyConcreteClassOperator*
  '}';

ModifyVirtualClassOperator: AddAttribute | AddReference | AddConstraint;

ModifyConcreteClassOperator:
  AddAttribute | AddReference | ModifyAttribute | ModifyReference | FilterProperty |
  AddConstraint | FilterConstraint;

AddAttribute: 'add' 'property' name=ID ':' type=[ecore::EDataType]
  (cardinality=AttributeCardinality)?;

AddReference: 'add' relationType=RelationType name=ID ':' type=TargetClass
  (cardinality=ReferenceCardinality)?;

ModifyAttribute:
  'modify' 'property' property=[ecore::EAttribute] '{'
    (('name' newName=ID)?
     & ('type' type=[ecore::EDataType])?
     & ('cardinality' cardinality=AttributeCardinality)?)
  '}';

ModifyReference:
  'modify' 'association' property=[ecore::EReference] '{'
    (('name' newName=ID)?
     & ('type' type=TargetClass)?
     & ('cardinality' cardinality=ReferenceCardinality)?
     & ('relation-type' relationType=RelationType)?)
  '}';

FilterProperty: 'filter' 'property' property=[ecore::EStructuralFeature];

FilterClass: 'filter' 'class' class=ConcreteClass;

AddConstraint: 'add' 'constraint' constraint=ID value=EString;

FilterConstraint: 'filter' 'constraint' constraint=EString;

enum AttributeCardinality: MANDATORY = '1' | OPTIONAL = '0';

ReferenceCardinality hidden(ML_COMMENT):
  lowerBound=CardinalityBound '..' upperBound=CardinalityBound;

enum CardinalityBound: ZERO = '0' | ONE = '1' | STAR = '*';

enum RelationType: composition | association;

EString returns ecore::EString: STRING | ID;
```

The overall structure to declare a metamodel extension includes its name, the metamodel(s) it extends and the list of applied operators (as well as the metamodel elements they are applied to). Listing 3 presents the full grammar of our textual DSL, thus highlighting its main concepts and operations [3]:

3. The complete Xtext version of this grammar is available from https://raw.githubusercontent.com/atlanmod/emfviews/master/dsls/mel/org.atlanmod.emfviews.mel/src/org/atlanmod/emfviews/mel/Mel.xtext

- **ADD** (a new concept to the metamodel)
  - **Create** *from scratch* a completely new concept.
  - **Specialize** (subtype) a concept.
  - **Generalize** (supertype) one or several concept(s).
- **MODIFY** (an existing concept, in the metamodel or added by the extension)
  - **Add property** (or reference/association) to an existing concept.
  - **Filter property** (or reference/association) from an existing concept.
  - **Modify property** (or reference/association) of an existing concept (equivalent to Filter + Add).
  - **Add constraint** to an existing concept or one of its properties/associations.
  - **Filter constraint** from an existing concept or one of its properties/associations.
- **FILTER** (an existing concept in the metamodel). Note that this DSL realizes a blacklist filtering strategy (cf. Section 4.2.2). We are voluntarily using the term FILTER and not DELETE as we do not want to actually delete elements but rather hide them. Filtering is applied on cascade (i.e. in the case of generalizations or derived properties)

Listing 4 – Example of metamodel extension specification in MEL.

```
import TOGAF from 'TOGAFmetamodelURI'

define EnterpriseArchitectureExtension extending TOGAF {

  add class FinancialBusinessArchitecture specializing TOGAF.BusinessArchitecture

  modify class FinancialBusinessArchitecture {
    add property currency : String
    add association financialbusinessprocesses : TOGAF.Process 0..*
    filter property name
  }

  filter class TOGAF.StrategicArchitecture

}
```

Listing 4 shows a simple metamodel extension illustrating the defined concrete syntax (using an example from the MoNoGe collaborative project, cf. . Section 5.2.2). It extends a base metamodel by adding a new concept (*add class* part) and then modifying it (i.e. adding a property/association and filtering an inherited property, cf. *modify class* part) as well as by filtering another existing concept (*filter class* part).

4.2.4 Integration With Model Persistence Solutions

As said in Section 4.2.1, scalability is a fundamental challenge to be addressed as far as model view solutions are concerned. In this section, we describe our general approach to support scalable model views on heterogeneous model sources [32]. The overall objective is to be able to build views that do scale up in practice: views that are built on top of several models where some are too large to be loaded, handled and stored only in memory (e.g. using the base EMF features such as XML/XMI serialization). This is made possible notably by relying on models that can be persisted and manipulated, when necessary, using appropriate database backends. The general approach we consider is depicted in Figure 4.8.

A *Modeling Framework* is usually composed of two main parts: a *Core* component

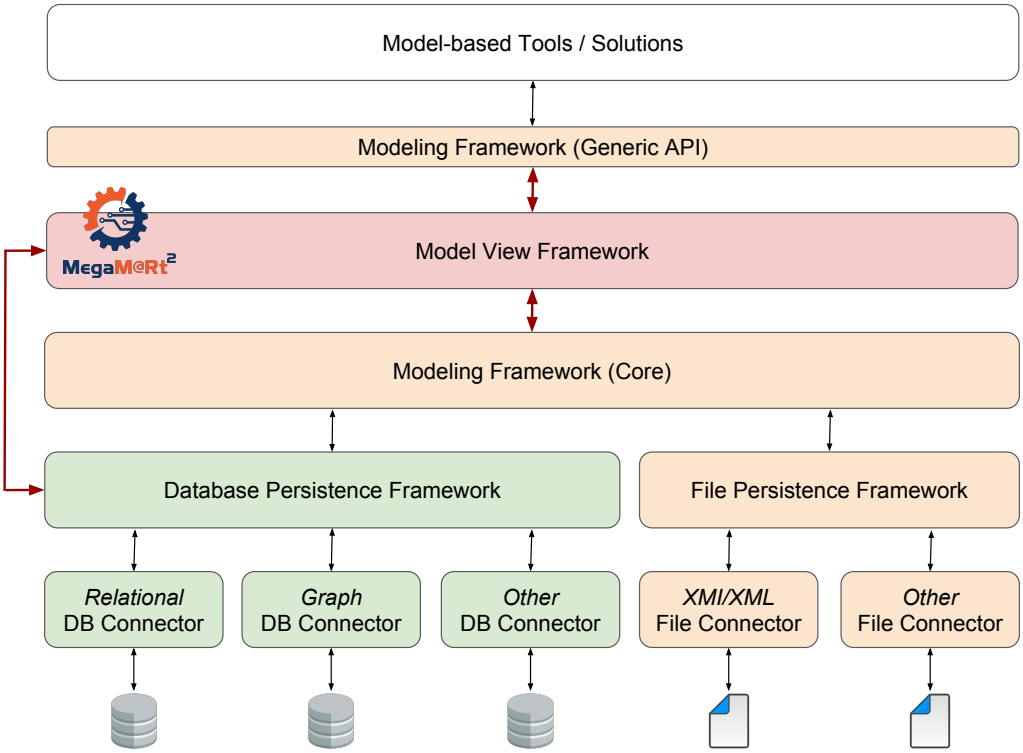

Figure 4.8 – A conceptual approach for integrating model view and model persistence capabilities.

providing the inner behavior (i.e. model manipulation facilities) and a *Generic API* as the interface provided externally for (re)use by *Model-based Tools*. The *Modeling Framework* also often provides a base *File Persistence Framework* relying on the local file system, coming with some file import/export capabilities in different serialization formats.

Database Persistence Frameworks have been proposed to connect the modeling framework to databases of various kinds (relational-based, graph-based, etc.). These solutions are typically used to store large models with a reduced memory footprint.

In the general case, the *Model View Framework* must be correctly integrated with the *Modeling Framework* and comply with its *Generic API*. This allows client applications to query views as regular models. Moreover, for model views to scale with large models, the *Model View Framework* has to leverage the characteristics of the *Database Persistence Frameworks*. This notably requires various refinements and optimizations from both sides. The next subsections describe the important goals we have identified in order to realize such a scalable integration.

Building Views on Heterogeneous Model Sources

This goal is of primary importance in the context of our generic approach and is a prerequisite to the three subsequent ones. As said earlier, most (if not all) modeling frameworks provide a default file serialization support, usually relying on XML-based format(s). However, they are very often not supporting other data sources. This is notably the case when needed to load/store models from/into different kinds of databases

(e.g. relational, graph). Such databases can be existing ones, e.g. handled by external applications, or can be created just for the sake of modeling.

Model view approaches generally rely on the model persistence support provided by their underlying modeling framework. Thus, they usually lack of support for scalable model persistence solutions, e.g. relying on databases. As a consequence, it is required to perform the integration of the model view framework with such database persistence framework(s). This way, depending on the nature of the contributing models, different persistence backends can be selected and combined in the context of a same view.

Such an integration can be performed in different ways. In some cases, it can be realized indirectly. The considered modeling framework can be first refined to be able to use the database persistence framework. Then, the view framework can simply rely on the general interface of the modeling framework in order to access transparently the underlying database resources. In some other situations, a direct connection can be required between the model view framework and the persistence framework. This notably allows implementing particular optimizations that could not be realized if passing by the modeling framework.

Persisting the View Information in a Scalable Way

Depending on the viewpoint/view specification, additional data can also be required in order to be able to fully compute it. For instance, this is the case when a given view provides new relationships between elements coming from different models, or when it adds new properties to elements from one of the involved models.

When initializing such a view, this view-specific information has to be obtained in some way. One possibility is to compute it at runtime when loading the view. This can be based on the data available from the contributing models and on some predefined queries executed on top of them. Another option is to collect it from an existing data source or a dedicated additional model. Such a model can come from manual inputs from the view users. It can also be the result of the running of an external application. In either cases, the view mechanism has to be able to retrieve the appropriate information in order to build the view.

Related scalability problems can occur when this view-specific data is too large to be handled correctly by the default support from the used modeling framework. Indeed, depending on the nature of the view, this extra data can be even larger than some of the contributing models themselves. In these cases, it is required to be able to persist a view-specific model (storing this data) by using a more scalable database persistence framework. Adopting such a strategy can reduce significantly the memory footprint of given views, thus allowing to manipulate views that could not be loaded otherwise.

Optimizing the View Loading and Element Access

With very large views, some operations can rapidly become heavy in terms of execution time and memory consumption. This can even go to a point where the view is not really usable anymore. For example, this is the case when the response time is too long (e.g. when the user navigates the view) or when the view simply ends by crashing. The

situation is notably critical during the process of initializing/loading the view, as it can require a significant number of model element accesses.

Relevant performance gains can be obtained by applying various lazy loading techniques at different levels. In the general case, any hit to an actual model element has to be delayed as much as possible and must only occurs when strictly needed. Such optimizations also concern accesses to both the various contributing models (cf. Section 4.2.4) and the view-specific elements (cf. Section 4.2.4). Ideally, all these accesses must be delayed without impacting the overall correctness of the view.

Moreover, depending on the used persistence framework(s), the model view framework can be refined differently. For given model element accesses, the view framework can directly benefit from specific capabilities provided by a database type (e.g. graph). For instance, the view framework can leverage the database API to turn full traversals of models into more selective requests, as traversals are time- and memory-intensive for large models.

Optimizing the View Querying

Once a view has been correctly created and loaded (cf. Section 4.2.4), it can be navigated and queried as any regular model according to the needs of the engineering activity it supports. As said earlier, the view framework usually relies on a generic interface provided by a modeling framework and shared between different tools from a same ecosystem. This way, it also natively supports the execution of queries defined in languages supported by this modeling framework.

However, when implementing this in practice, performance issues can arise. For instance, some models can be serialized in standard XML-based files while others can be stored in databases (cf. Section 4.2.4). In this situation, using the default querying support might not take advantage of backend-specific optimizations. Thus, more elaborated schemes have to be considered.

Base operations (e.g. `allInstances` in OCL) can be costly to execute with the default behavior of the modeling framework. For better efficiency, such operations could be delegated to the various persistence frameworks used in a view. This is illustrated in Figure 4.9 where an OCL query navigates a given view and returns the number of design requirements that are impacted by a specified runtime event, captured as a log from an execution trace. This query can be optimized by delegating the `allInstances` call to the database persistence framework that contains the related elements, thus bypassing the default (less efficient) implementation of the modeling framework.

More generally, large performance gains are possible by splitting a query (on a given view) into a request plan that is better suited to the underlying persistence frameworks. This requires the model view framework to be able to split any query, delegate its execution and collect its results by leveraging the specificities of the different persistence backends.

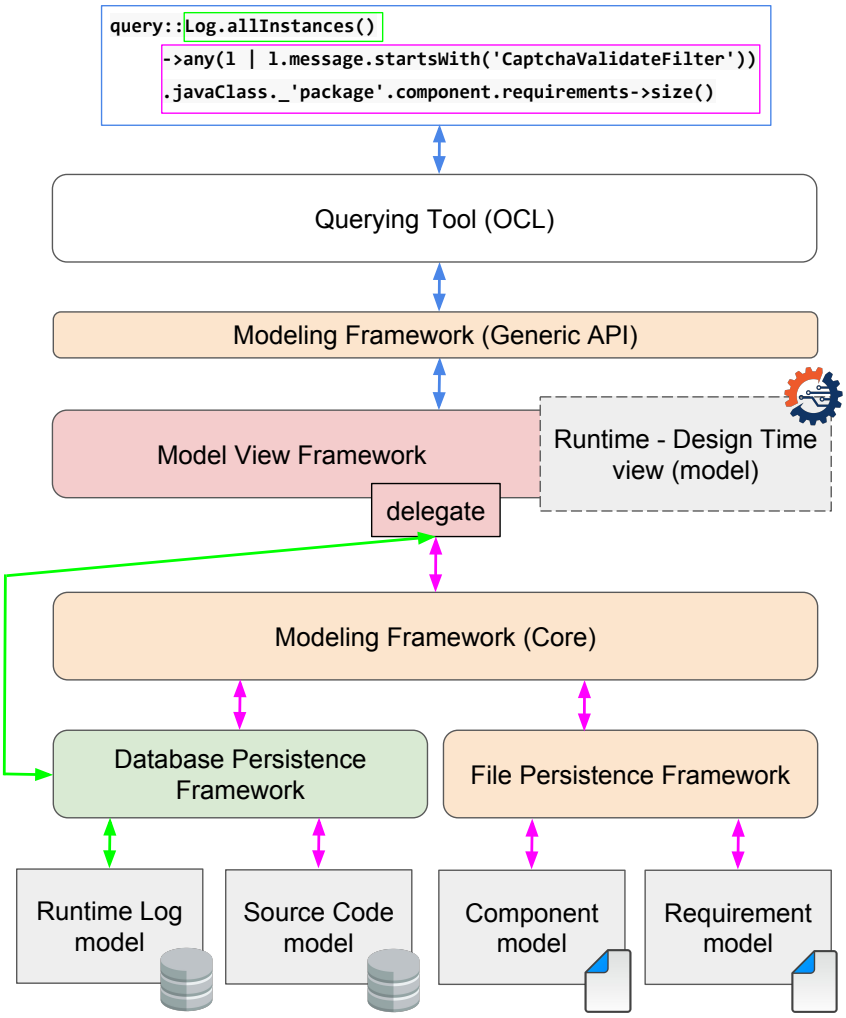

Figure 4.9 – Optimizing model view querying by delegating to model persistence backends.

4.2.5 Main Benefits

To the best of our knowledge, none of the available model view approaches fully satisfies all the characteristics introduced in Section 4.2.1. There is always a trade-off between the offered capabilities and some of these properties, such as scalability or synchronization more particularly. Moreover, no approach currently appears to provide both inter-model view support and the expected expressivity in terms of viewpoint/view definition. In what follows, we summarize how our approach intends to tackle these properties and also indicate to which degree of completeness or coverage this can be done.

Genericity can be ensured as any metamodels or models can be considered to build viewpoints and views respectively. Our proposed approach is completely modeling language-agnostic and allows combining several models (that possibly conform to different metamodels) into a single view.

The provided DSLs and our core virtualization metamodel also allow to offer an interesting level of *expressivity*. We have been able to observe that this is already sufficient in some commonly encountered contexts (cf. Section 4.2.3). However, more DSLs could still be built on top of our approach in order to target different purposes and eventually

identify a few missing elements in our core metamodel.

Non-intrusiveness can be naturally achieved in our solution since the base metamodels/models contributing to the viewpoints/views are not modified at all. This is a direct consequence of the model virtualization approach, and its underlying proxy mechanism (i.e. original model elements are only accessed but never duplicated), we directly rely on at both viewpoint and view levels.

As mentioned earlier, *interoperability* (at least under a same modeling environment) can be guaranteed as model views can be used wherever regular models can be. For example, this is the case of model transformations or code generators for which they can be used as inputs. However, it is important to note that the current implementation (cf. Section 4.3) mostly works in a read-only mode: views cannot be fully updated, as explained in the next paragraph.

Indeed, *modifiability* is only partially supported in the current implementation, even though it could be fully supported by our conceptual approach. Thus, only changes in attribute values from the view are automatically propagated back to the original models so far. However, since views and actual models do share the same real instances via a proxy mechanism, *synchronization* at view-level is directly obtained by default.

In the case of *scalability*, the most suitable way of measuring it is via empirical experiments on model data sets of large sizes. To this intent, more detailed benchmarks have been performed and are presented later in this manuscript (cf. Section 4.4). We have notably been able to benefit from the integration approach with model persistence solutions, as introduced before in Section 4.2.4.

4.3 The EMF VIEWS framework

This section describes the EMF Views framework implementing the generic model view approach described in previous Section 4.2. The goal of EMF Views is to facilitate the creation and handling of viewpoints/views on top of possibly heterogeneous metamodels/models. In what follows, we start by providing an overview of the EMF Views implementation in Section 4.3.1. Then, we give more information on the currently available tooling support in Section 4.3.2. Finally, we present the practical integration we have realized with database backends such as NeoEMF [49] and CDO [60] in Section 4.3.3.

4.3.1 Implementation Overview

EMF Views has been implemented on top of several existing technologies, reused and/or refined when needed, coming from the lively open source ecosystem around Eclipse and its well-known modeling framework EMF (cf. Section 2.3). Thanks to EMF notably, we have been able to directly rely on general model creation and handling capabilities. Based on that, the current implementation of EMF Views as depicted in Figure 4.10 and Figure 4.11 mainly consists in the following components:

— *Viewpoint and View builders* as described in Section 4.2.1, implemented as Eclipse plugins in Java and offering a standard integration with the Eclipse workbench and its different components (workspace explorer, editors and viewers, etc.).

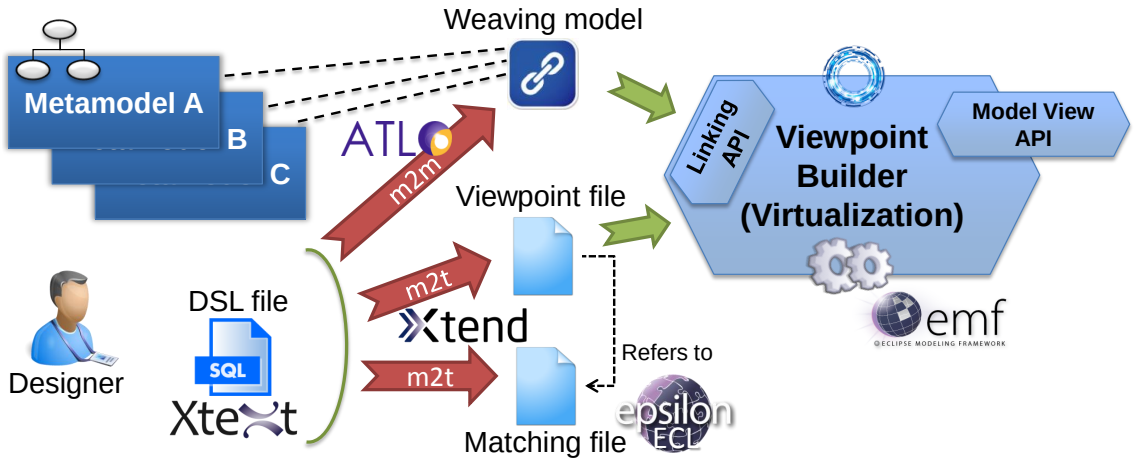

Figure 4.10 – Current implementation of EMF Views (design time).

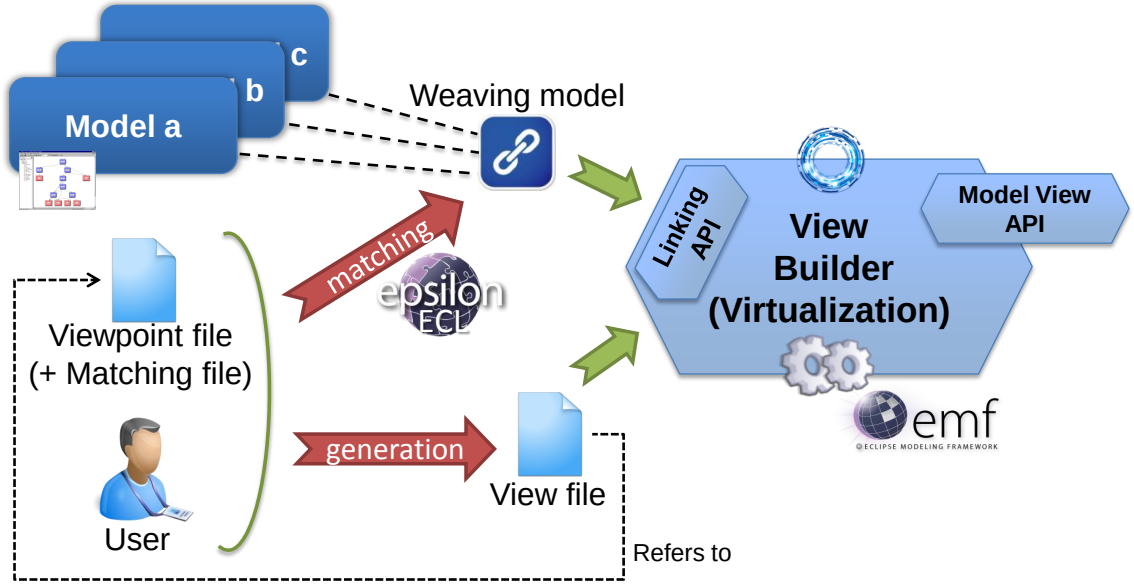

Figure 4.11 – Current implementation of EMF Views (runtime).

- A *Model View API*, deriving from the EMF standard model access API for interoperability/integration purposes, that supports model virtualization in order to be able to handle viewpoints and views transparently as any regular EMF metamodels and models (respectively).
- A *Linking API* that has for main role to initiate inter-model virtual associations when required, and has notably been connected to the mapping engine of ECL [125] in order to automatically compute such links at view-level. Note that the implementation of this API is generic and other matching languages could be plugged in to replace ECL whenever relevant.

— *DSLs* as described in Section 4.2.3, developed with Xtext [77] and then integrated using Xtend [76] model-to-text and ATL [116] model-to-model transformations. For user convenience, the DSL notably comes with a proper editor as well as syntax highlighting and some content-assist (e.g. displaying the classes or attributes/references which are applicable in the context of a given metamodel or class).

The open source tool as well as its complete source code, update sites, the related

documentation and some screencasts are all available online [11].

4.3.2 Tooling Support

As mentioned before in Section 4.3.1, the EMF Views tooling is well integrated within the Eclipse workbench. It comes as a set of Eclipse plugins offering the previously presented EMF Views components. Thus, the use of the provided features is completely transparent for the users that are already familiar with Eclipse and its graphical interface. A concrete example of the deployment and application of EMF Views inside Eclipse can be seen from Figure 4.12.

Indeed, once the EMF Views plugins installed, users can directly specify their own viewpoint definition using the textual editor coming along with VPDL. For instance, in the screenshot from Figure 4.12, a given user created a viewpoint definition that refers to the TOGAF, BPMN and REQIF metamodels as loaded into its workbench's EMF metamodel registry.

Then, from this *threeModelComposition* viewpoint definition in VPDL, EMF Views automatically generated the corresponding Viewpoint file (*.eviewpoint*), Weaving model (*.xmi*) and Matching file (*.ecl*) (cf. Section 4.2.1 for more details on the underlying approach). It also allowed selecting the models that actually contributed to a corresponding *threeModelComposition* view. In this case, the concerned models are stored in the 3 *.xmi* files from the *models* folder.

Based on this, users can create different View files (*.eview*) listing the actual contributing models and pointing to the corresponding Viewpoint file. EMF Views can then automatically initialize an associated Weaving model based on the matching information provided at Viewpoint-level by the Matching file (if available). In this case, this Weaving model is only loaded in memory and not serialized in a file (cf. Section 4.2.1 for more explanations). When directly opening this View file with the generic MoDisco Model Browser (cf. Section 3.3.2), the view behaves and can be navigated as any regular EMF model.

In Figure 4.12 we can see that, from the *Booking Trip* process in the TOGAF model, we can now access to the corresponding process in the BPMN model as if they were both part of the same model. The same is also true for the TOGAF *Requirement* elements to the corresponding REQIF *SpecObject* elements (i.e. equivalent to a process in the REQIF terminology). In both cases, the navigation can possibly continue back and forth between the 3 models interconnected in the view.

4.3.3 Integration With Model Persistence Solutions

We have concretely implemented the conceptual integration approach described in Section 4.2.4 by relying on EMF as our core Modeling Framework and EMF Views as our Model View Framework. Concerning the Persistence Frameworks, we used both NeoEMF [49] and CDO [60] as supporting graph and relational database backends (respectively). In what follows, we detail how all these technical solutions have been combined together in practice to address the four different key points identified in our conceptual approach. Our current implementation is freely available from a particular branch of the

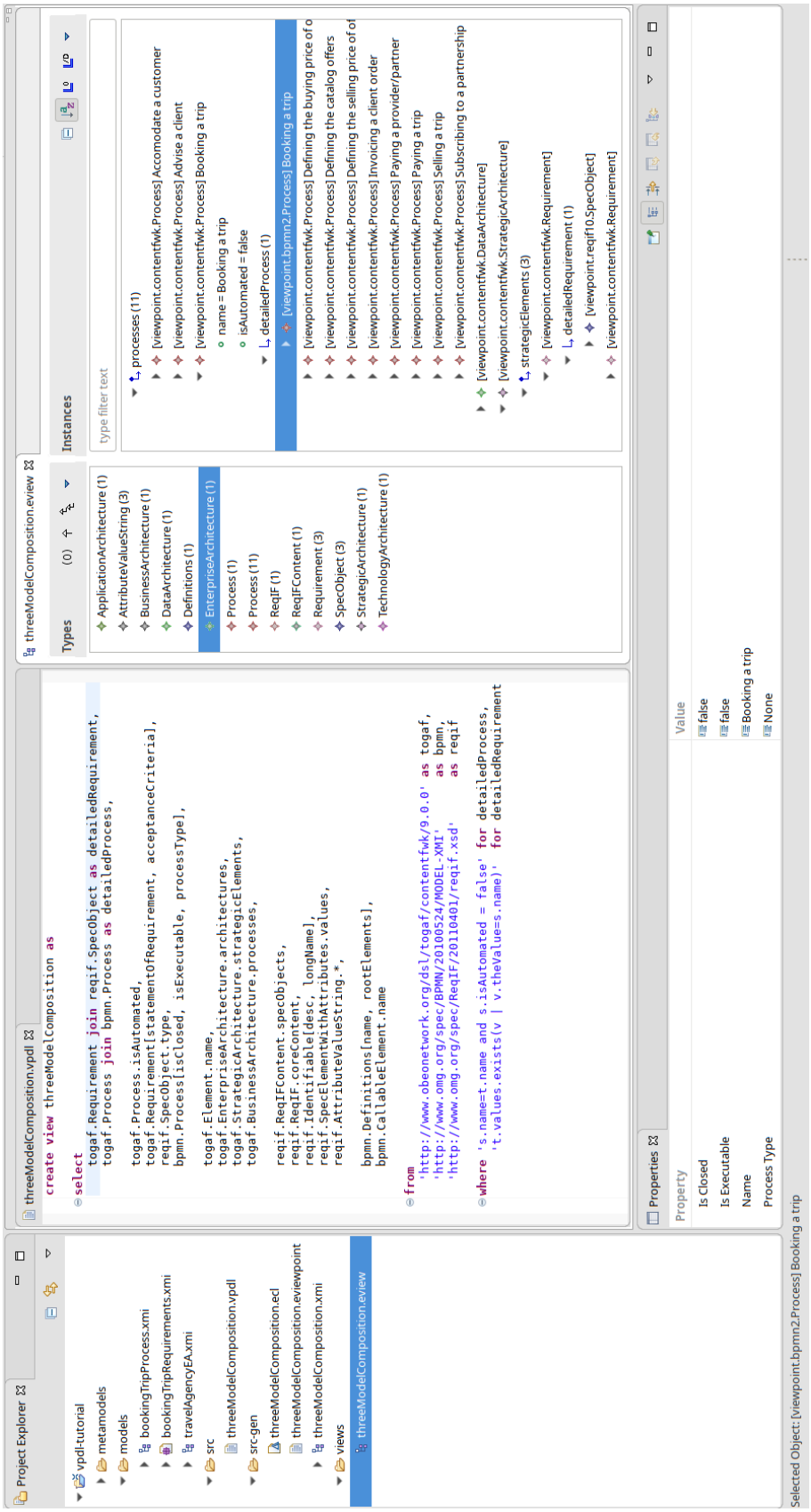

Figure 4.12 – Screenshot of an Eclipse workbench with EMF Views installed.

EMF Views source code repository [4].

Building Views with EMF Views on Models stored in NeoEMF and CDO

CDO and NeoEMF for Model Persistence CDO [60] is a model persistence framework designed to handle large EMF models by relying on a client-server repository structure. A CDO application can connect to a CDO server using a specialized interface, and a dedicated implementation of the EMF API is provided to manipulate the model. CDO is based on a lazy-loading mechanism and supports transactions, access control policies as well as concurrent model editing. Its default implementation uses a relational database connector to serialize models into SQL compatible databases. However, the modular architecture of the frameworks can be extended to support different data storage solutions (even if, in practice, only relational connectors are used and regularly maintained).

NeoEMF [49] is a complementary model persistence framework that relies on the scalable nature of NoSQL databases to store and manipulate large models. NeoEMF supports three model-to-database mappings, i.e. graph, key-value and column stores. Each one of them is adapted to a specific modeling scenario, such as atomic element accesses (key-value) or complex navigations (graph). As other persistence solutions, NeoEMF provides a *lazy-loading* mechanism that allows to obtain significant gains in terms of performances.

Since CDO and NeoEMF are two of the main actors in the field of scalable model persistence, we chose to rely on them in our implementation.

Integration Since EMF Views, NeoEMF and CDO are all part of the EMF ecosystem, integrating them together is a relatively straightforward task since they all implement the same EMF model handling API. It is mostly a matter of telling EMF Views how to retrieve and load the right model resources. However, CDO and NeoEMF resources require platform-specific initialization code (such as specific URI schemes, *resource factory* implementations and data store configurations) that had to be integrated into EMF Views. Note that this code is also available from the EMF Views/NeoEMF integration repository (cf. the URL indicated earlier in this section). Once loaded, all the model resources are navigated through the standard modeling API. This way, the persistence frameworks transparently delegate the operations to the databases in a scalable manner

Persisting the View Information with NeoEMF

As explained before in Section 4.2.1, EMF Views uses a *weaving model* that represent the view-specific information. This model can potentially contain entries for many elements coming from the different contributing models. Thus, it can get as large or even larger (depending on the view) than the contributing models themselves. In order to improve the scalability of our approach on large-scale views, we chose to persist this model using NeoEMF instead of using the default XMI serialization.

Since the weaving model is also defined as a standard EMF model, its migration to NeoEMF has been done quite transparently by changing the model serialization behavior

4. https://github.com/atlanmod/emfviews/tree/integrate-neoemf

(and initializing the corresponding database backend). Persisting the weaving model in NeoEMF allows us to handle views that cannot fit in memory otherwise.

Optimizing the View Loading and Element Access in EMF Views

When dealing with large database resources, many EMF API operations having little to no overhead with small in-memory resources now potentially bear high costs in execution time and memory consumption. So we have to pay extra attention to minimize the impact of such operations. For instance, checking whether a reference has any contents can be done by calling the *EList.isEmpty* operation. A naive implementation of this operation compares the size of the collection against zero, where getting the size is an $O(n)$ operation. On small in-memory resources, n is small and a call to the *isEmpty* operation triggers no issue. On large database resources, n is large and the overhead of hitting the database can become a bottleneck. A better implementation of *isEmpty* rather checks if at least one element exists, and thus exits early when this is not the case. Similarly, getting a given element of a multi-valued reference by using the *EList.get* operation can be costly. This is the case when the implementation first builds a list containing all the elements of the reference, regardless of the index requested. If, instead, the implementation navigates to the index and looks no further, then we make less hits to the database and the operation has a minimal cost.

We significantly improved EMF Views for large model resources by following these ideas. As introduced in Section 4.2.4, one of our key tenets was: delay actual hits to the resource as much as possible.

Another important improvement of EMF Views concerned the view loading process. As said previously, weaving models can be large depending on the number (and contents) of virtual associations. Previously, EMF Views eagerly populated these virtual associations when loading the view. Each virtual association thus delayed loading the view further: for larger weaving models, this meant several seconds or even minutes. Here again, the optimization lies in laziness: delaying work that can be done later. In this case, we have to populate virtual associations only when they are first accessed. If some virtual associations are never looked up then we never have to load them from the weaving model, thus avoiding the loading cost. Making this change to EMF Views enabled loading views with large weaving models with no overhead in terms of time.

A third point of optimization was to tweak the way the data is stored into the graph database handled by NeoEMF. In some cases, large models can be (very) flat ones (e.g. log files). They can contain a top-level element holding a large collection of sub-elements (some of which have also children). We have observed that this flat structure was reducing the performances of NeoEMF. To solve this issue, we developed a new mapping from model to graph for NeoEMF, using in-database linked lists. This mapping, dedicated to large collections, allowed us to speed up the creation of the runtime log model and the access to its elements by a factor of 30.

Optimizing the View Querying in OCL

Since views are regular EMF models, querying tools like OCL [160] or transformation tools like ATL [59] can be applied transparently on views (regardless of the underlying

persistence framework used by the contributing models). However, relying only on the EMF API can have a cost in terms of performances. For example, it has been widely observed that some base operations, like `allInstances` in OCL, can be quite costly to execute naively using the EMF API. On the contrary, persistence backends may provide more efficient ways to execute such operations. For example, NeoEMF resources expose a `getAllInstances` method which can compute the set of instances of a given classifier. This method is around 40 times faster than using the EMF API directly.

We extended the standard OCL interpreter in order to specialize some operations according to the data store they target. In the following paragraph we detail our implementation of the `allInstances` operation, but other native operation implementations can be easily defined to enhance query computation performances. However, our implementation still needs a generic operation delegation mechanism that, along with the support for other query languages, is currently left for future work.

The OCL API provides a way to customize the behavior of the `allInstances` operation through a *Model Manager* (or *Extents Map* in the legacy implementation). We define a custom extents map that allows to specialize the `allInstances` call according to the concrete data stores used in a given view. When instances of a classifier are looked up, the extents map redirects the call to the view that fetches instances—using native database calls—from each contributing model and them combines these instances as the result. Compared to the standard OCL implementation that iterates the entire model to match elements of a given type, this approach benefits from the low-level optimizations of the databases (such as built-in indexes and caches).

4.4 Evaluation

This section explains how we have evaluated the proposed conceptual approach and its implementing EMF Views framework. In order to provide a relevant evaluation covering some of the general challenges introduced in Section 4.1.4, this has been made using quantitative ways. In Section 3.4.1, we start by summarizing the research questions / challenges we intended to evaluate. Then, Section 4.4.2 introduces a practical use case involving the use of both large and heterogeneous models in a same view. Section 4.4.3 presents the main objectives we are trying to achieve with this particular use case. Section 4.4.4 describes the general process we followed, while Section 4.4.5 details the performed benchmarks as well as the obtained results.

4.4.1 Research Questions (RQs)

The evaluation described in what follows in this section was performed to quantitatively assess the relevance and usefulness of our approach when applied to realistic scenarios. More specifically, we aimed to answer the following research questions and underlying Model View challenges (cf. Section 4.1.4):

1. *RQ5 - Scalability*. Are we able to use the EMF Views conceptual approach and/or its implementing components in the context of large-scale scenarios? To evaluate this, we have worked on dedicated scalability benchmarks intending to measure

different scalability aspects of our solution (cf. Section 4.4.5).

2. *RQ6 - View maintenance*. Using the EMF Views conceptual approach and/or its implementing components, are we able to ensure a regular synchronization from the contributing models to the built views? We have not explicitly evaluated this so far. However, this capability is ensured by default as provided by the model virtualization approach which EMF Views relies on (cf. Section 4.2.1).
3. *RQ7 - View update*. Using the EMF Views conceptual approach and/or its implementing components, are we able to ensure a regular synchronization from the built views to the contributing models? We have not explicitly evaluated this so far. The current EMF Views implementation (cf. Section 4.3) already provides some base update capabilities (e.g. when a given attribute value is updated in a view, this change can be reflected back to the corresponding contributing model). However, a lot of work remains to be done in order to be able to support more complex kinds of updates.

Of course the presented evaluation could still be extended in the context of future work, e.g. to cover (many) more different Model View scenarios as well as to consider even (much) larger contributing models in views. We could notably try to evaluate more precisely the view maintenance and update features (to be) provided by EMF Views. Nevertheless, we believe the current evaluation already allows providing some interesting insights on the actual capabilities of our approach and its current implementation in a realistic context.

4.4.2 Practical Use Case

From the MegaM@Rt2 collaborative project we are involved in (cf. Section 5.2.1), and notably from the description of the industrial requirements and case studies, we have been able to extract a general scenario that concretely illustrates the need for scalable model views and their persistence. Thus, let us consider the realization of a runtime $\leftrightarrow$ design time feedback loop via a view gathering 4 different models covering both runtime and design time aspects of a given system in the project. As shown on Figure 4.13, this view relies on a runtime log model (that conforms to a simple trace metamodel), a Java code model (that conforms to the Java metamodel from MoDisco [29]), a component model (that conforms to OMG UML [167]) and a requirement model (that conforms to OMG ReqIF [162]).

On the one hand, the runtime log model and (to a lesser extent) the Java model can be considered as runtime models. They can potentially be very large, especially the runtime log model which represents actual system execution traces. Thus, a typical solution to store and access them in a scalable way is to rely on database model persistence frameworks. The used technical solution then depends on the nature of the model, its access/handling scenario or the required features.

On the other hand, the component model and requirement model can be considered as design models. They are generally of a reasonable size compared to the runtime ones, because they are very often manually specified. Hence, they can be handled by standard modeling frameworks relying on in-memory constructs and/or XML-based files.

A concrete example of the view from Figure 4.13 is given in Figure 4.14. By using

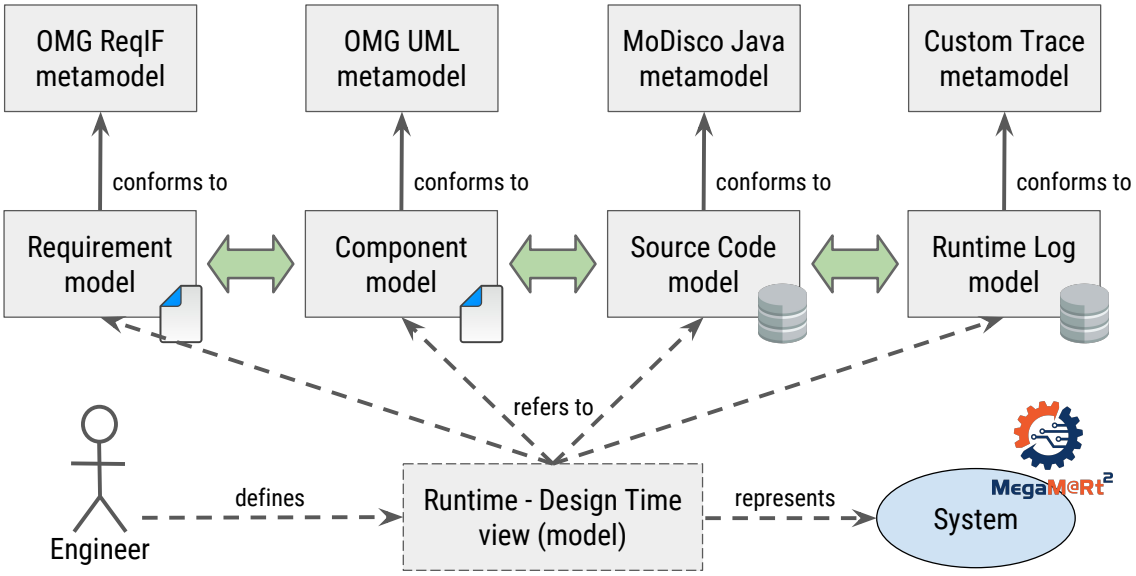

Figure 4.13 – Running use case from the MegaM@Rt2 industrial project.

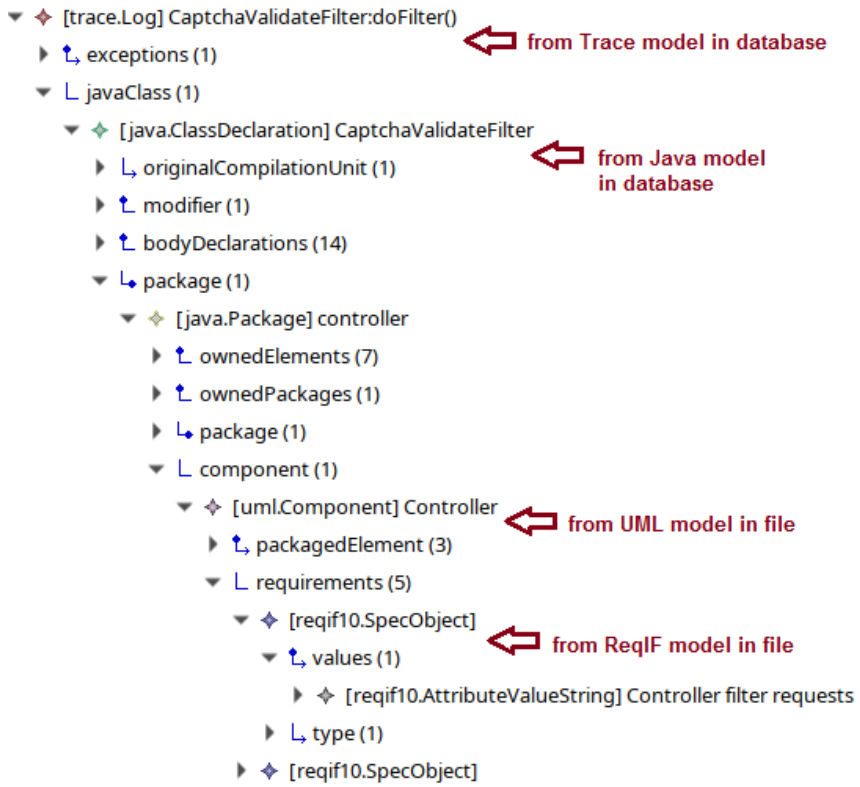

Figure 4.14 – Concrete example of a view in MegaM@Rt2 (based on the use case from Figure 4.13).

this view, an engineer can navigate transparently within and between the four contributing models as if they were all part of the same single model. Thus, from a particular runtime information collected at system execution (here a *trace.Log* element), one can move back to the originating Java code instructions (here *java.ClassDeclaration* elements). One can then follow links to the components (here *uml.Component* elements) the code implements, and up to the actual requirements these components fulfill (here

reqif10.SpecObject elements).

Such a view combining different models can also be queried as any regular model, in order to extract relevant data out of it. For example, one can obtain all the requirements that are related to a given execution trace (runtime to design time traceability). Or, the other way around, one can get all the execution traces that correspond to a particular requirement (design time to runtime traceability). We could imagine many other similar queries also relevant in the context of MegaM@Rt2, according to different needs of the industrial partners.

To summarize, the main benefit of using a view is to collect in a transparent way information that is spread among different models. Without such a view, the engineer has to query the different models one by one and then aggregate the obtained results by herself. This includes recreating the mappings between related elements from different models in the view. Instead, using a view, queries traversing several contributing models (such as the queries mentioned previously) can be expressed and computed naturally as if dealing with a single model.

4.4.3 Objectives

In the MegaM@Rt2 context, it is not sufficient to be able to build model views: the view mechanism must scale up when aggregating very large models provided by the industrial partners. More generally, the need for scalable modeling solutions has been observed in several industrial contexts [210, 109], and is also recognized as a long-term challenge from a research perspective [127]. However, existing model view solutions do not handle large and very large models well, if at all (cf. Section 4.2.1).

With this experiment, we target the building, handling and querying of scalable model views over heterogeneous model resources (cf. RQ5 in Section 4.4.1). We consider here heterogeneity in terms of both the used modeling languages (i.e. metamodels) and the underlying persistence solutions. As introduced before in Section 4.2.4 and Section 4.3.3, we intend to evaluate the four following challenges:

1. Refining the model view framework to model resources using different persistence solutions.
2. Persisting any view-specific information in a scalable way.
3. Loading views and accessing view elements with a reasonably low overhead.
4. Querying views efficiently, e.g. by leveraging persistence-specific optimizations.

At the time of writing, we have successfully tackled (1), (2) and (3), and partially addressed (4). The details are provided in what follows.

4.4.4 Process

To evaluate our integration approach and its current implementation, we applied them in the context of the use case introduced earlier (cf. Section 4.4.2). We focused on measuring the *time overhead* of our current implementation, because it directly impacts the interactive user experience (as opposed to batch processing). However, as dealing with on-disk resources is inevitably (one to two orders of magnitude) slower than dealing with

in-memory resources, matching the speed of in-memory resources is not a realistic goal. Thus, we rather insisted on the asymptotic behavior of our approach and on gains made by our optimizations. For reproducibility, the complete source code of the performed benchmarks (including the models and views) as well as more detailed results are available online [5] [6].

All the benchmarks have been realized on a laptop with an i7-7600 (2.80GHz) processor, 32GB of RAM, and M.2 PCIe SSD, using OpenJDK 64-bit 1.8.0. We built two versions of the same view answering to the MegaM@Rt2 use case.

The first version is fully file-based: all four contributing models and the view-specific information (i.e. the weaving model) are serialized using standard EMF-XMI. Thus, once loaded, the view resides fully in memory.

The second version demonstrates our capability to build views over heterogeneous model resources. It uses a mix of file-based and database resources as contributing models. More precisely, the Runtime Log model and the weaving model are persisted in a Neo4j graph database handled by NeoEMF, using our mapping developed for optimizing flat models (cf. Section 4.3.3). The Java Source Code model is persisted in a relational database handled by CDO. Only the two remaining models, namely the UML Component model and the ReqIF Requirement model, are serialized as XMI files handled by the standard EMF implementation.

Furthermore, we also evaluated the scalability of both versions of the view. To this intent, for each version we considered different sizes for the Runtime Log model, going from 10^1 to 10^6 elements. This way, we have been able to measure the performance of the view creation, loading and querying up to large-scale models, as required in our MegaM@Rt2 context.

4.4.5 Scalability Benchmarks

Benchmark 1: Creating the View-specific Information

The first benchmark evaluates the creation of the view-specific data, stored as a weaving model, that is needed by the view in order to be loaded. This benchmark notably measures the overhead of navigating and populating such databases resources, compared to in-memory EMF resources.

The weaving model contains the new (virtual) links between the different models composing the view. In Figure 4.14, we created three virtual links: (1) we connect a given execution Log to the Java *ClassDeclaration* that emitted it; (2) we relate the Java Package this *ClassDeclaration* is part of to the UML Component that represents it at design level; (3) we link this UML Component to the corresponding ReqIF *SpecObject*, i.e. the requirement the UML Component is supposed to support.

As a consequence, creating the weaving model implies checking different matches between two elements coming from two contributing models. For large models, such as the Runtime Log one from the MegaM@Rt2 scenario, these matches can be very

5. https://github.com/atlanmod/scalable-views-heterogeneous-models
6. http://remodd.org/content/towards-scalable-views-heterogeneous-model-resources

Size	XMI	Hetero.	Overhead
10^1	0.001	0.051	38
10^2	0.001	0.028	23
10^3	0.001	0.058	52
10^4	0.004	0.293	81
10^5	0.098	1.778	18
10^6	8.460	22.556	3

Table 4.2 – Time (in minutes) to create the view-specific models (weaving models).

Size	XMI	Hetero.	Overhead
10^1	0.011	0.704	64
10^2	0.036	1.828	51
10^3	0.285	13.284	47
10^4	2.799	130.736	47
10^5	28.112	1316.776	47
10^6	282.994	13263.500	47

Table 4.3 – Size (in megabytes) of the view-specific models (weaving models) on disk.

numerous. In these cases, the whole matching process can take a significant amount of time. Table 4.2 compares the time it takes to create the weaving model using the two versions of our view, while Table 4.3 compares the sizes of the persisted weaving models on disk. The *Size* column in both tables refers to the number of log elements in the Runtime Log model.

A first observation is that, while models stored in databases are, as expected, slower to create than models serialized in XMI, the overhead for the heterogeneous views diminishes when models get larger. This is possibly indicating a better asymptotic performance. A second observation is that the weaving models persisted in databases are overall 50 times larger than the ones persisted in XMI, and this factor is constant across model sizes. The large size of the persistence format used by the database backend can be explained by the creation of many indexes and logs when initializing the resource. Even though the heterogeneous resources are larger on disk, the compromise is made in favor of faster lookup as we will see in next two benchmarks.

Benchmark 2: Loading the View

In the second benchmark, we evaluate both the loading of a view and the iteration over all its contents. Again, we perform this on the two versions of the view (full XMI vs. databases + XMI). This benchmark measures the overhead of accessing the content of the different models contributing to the view. Table 4.4 compares the time it takes to load the two versions of the view, while Table 4.5 compares the time required to iterate over the full content of the view.

A first point is that loading the heterogeneous view takes a relatively low and constant time (between 1 and 4 seconds), regardless of the size of the Runtime Log model (i.e. the largest one) contributing to the view. For the largest model size, it takes four times longer to load the first view compared to the similar heterogeneous view which uses our appro-

Size	XMI	Hetero.	Overhead
10^1	0.788	2.265	2.87
10^2	0.257	0.870	3.39
10^3	0.245	0.750	3.06
10^4	0.389	0.811	2.08
10^5	0.921	2.482	2.69
10^6	12.214	3.006	0.25

Table 4.4 – Time (in seconds) to load the view.

Size	XMI	Hetero.	Overhead
10^1	1.468	5.049	3
10^2	0.641	3.029	5
10^3	0.469	2.222	5
10^4	0.623	2.833	5
10^5	0.948	6.795	7
10^6	1.946	82.323	42

Table 4.5 – Time (in seconds) to iterate over the full content of the view.

ach. This difference can be explained by the lazy loading of our approach, where most of the actual loading takes place when navigating model elements.

When iterating over the full content of the view, the overhead remains relatively small, but slightly increases as the Runtime Log model gets larger. For the largest size, the heterogeneous view is 42 times slower to navigate, which is around the expected speed difference between RAM and disk. This large increase in time can be partly explained by the data model of the underlying database, for which exhaustive iteration is a very costly operation due to numerous loads/unloads between database and memory. While a full iteration scenario may not be very common in practice, it is a useful reminder that the choice of data representation can have a strong impact on performance.

Benchmark 3: Querying the View

In the third benchmark, we measure the time it took to successfully run three different OCL queries on top of our view. The considered OCL queries are the following:

1.
```
Log.allInstances()->size()
```

2.
```
Log.allInstances()
->any(l| l.message.startsWith('CaptchaValidateFilter'))
.javaClass._'package'.component.requirements
->size()
```

3.
```
SpecObject.allInstances()
->any(r| r.values->selectByType(AttributeValueString)
     ->exists(v| v.theValue.startsWith('Controller')))
.components->collect(c| c.javaPackages)
->collect(p| p.ownedElements)->selectByType(ClassDeclaration)
->collect(c| c.traces)->size()
```

Size	XMI	XMI (Opt.)	Hetero.	Hetero. (Opt.)
10^1	1.083	0.049	5.655	4.722
10^2	0.683	0.025	2.694	3.000
10^3	0.477	0.019	2.087	1.744
10^4	0.433	0.022	2.707	1.905
10^5	0.872	0.576	7.777	2.309
10^6	5.485	0.752	85.030	14.544

Table 4.6 – Time (in seconds) to run the OCL query (1).

Size	XMI	XMI (Opt.)	Hetero.	Hetero. (Opt.)
10^1	1.093	0.116	5.075	4.043
10^2	0.711	0.031	3.478	2.563
10^3	0.527	0.033	3.256	2.087
10^4	0.570	0.068	5.336	3.173
10^5	1.050	0.241	16.604	9.626
10^6	6.294	4.008	178.444	114.733

Table 4.7 – Time (in seconds) to run the OCL query (2).

The first query simply counts all the instances of `Log` elements in the view, and thus only accesses the Runtime Log model via the view. The other two traverse the complete view, i.e. they access to elements from all of the four contributing models.

Table 4.6 compares the time it takes to execute query (1) on the two versions of our view. Tables 4.7 and 4.8 do the same for queries (2) and (3), respectively. In these three tables, the two additional *(Opt.)* columns refer to optimized views that use the custom extents map we described in Section 4.3.3.

One observation is that, on the heterogeneous view, the queries can be 11 to 30 times slower than on the XMI view, which is still lower than the expected speed difference between RAM and disk. The optimized versions are a 2- to 6-fold improvement, which brings the overhead down to 3 to 30 times slower, as the optimizations also benefit the XMI view. The effect of the specialization of the *allInstances* operation on the Runtime Log model stored in database is the most evident on the last line of Table 4.6. For the other two queries, the improvement is lower, but still significant. Some further gains may lie in fully specializing the queries into backend-specific request plans, as proposed in Section 4.2.4.

Size	XMI	XMI (Opt.)	Hetero.	Hetero. (Opt.)
10^1	1.133	0.186	4.473	3.860
10^2	0.664	0.026	2.575	2.270
10^3	0.688	0.046	2.148	1.869
10^4	0.691	0.248	5.138	3.147
10^5	1.757	0.857	18.518	12.843
10^6	12.722	7.647	251.451	154.621

Table 4.8 – Time (in seconds) to run the OCL query (3).

4.5 Conclusion

When (reverse) engineering complex systems, many different models are used to represent various system aspects. These models are often heterogeneous in terms of modeling language, provenance, number or scale. Because of this situation, the information relevant to engineers is usually split into several interrelated models. As a consequence, model view approaches are required in order to properly federate such models together. The objective is to provide global views improving the comprehension of the system under study (possibly from different perspectives).

In this Chapter, we first described an extended state-of-the-art on existing model view solutions (coming from both the academic and industrial worlds). The obtained results can be considered as a first step in the direction of clarifying the terminology and situation in terms of views/viewpoints on models. Then, we presented a generic model view conceptual approach and a corresponding implementation framework named EMF Views. They intend to facilitate the specification, creation and handling of viewpoints and views over possible heterogeneous large metamodels and models (respectively). The provided description notably includes the overall underlying approach, a core virtualization metamodel, a couple of connected DSLs and an integration approach with existing persistence solutions. We also presented a realistic application of our approach and the EMF Views framework on a practical use case. In addition, we discussed the results of different related scalability benchmarks.

EMF Views, as a project and technical solution, is relatively far from being as mature as our MoDisco contribution to MDRE (cf. Chapter 3). We are still at an early phase, even if a significant development and consolidation effort has been realized in the context of various collaborative projects (cf. Section 5.2). Nevertheless, all the related resources (source code, examples, documentation) have already been made available in open source. Thanks to this, we hope to trigger more interest from the community on both our model view approach and the associated technical solution in the coming years.

A positive signal is that we have already been able to deploy and apply our conceptual approach and EMF Views framework in the context of a concrete use case coming from real industrial needs (cf. 4.4.2). It has shown in practice that it can actually scale up if a sufficient research and development effort is made. It has also proved its capabilities in terms of genericity, expressivity, non-intrusiveness or interoperability with other existing solutions. However there are still many open challenges, for instance as far as view modifiability and synchronization are concerned. This gives room for more interesting research on these topics in the future.

5

Conclusion

5.1 Summary

In this manuscript, we have drawn the basis of a modeling infrastructure that aims at improving the support for software reverse engineering and comprehension in the context of possibly complex and heterogeneous systems. More specifically, we have proposed and described a couple of complementary and reusable model-based approaches. These two model-based approaches (and their respective implementation frameworks) allow dealing with model discovery, model understanding and model federation (via model views) activities in a scalable way.

Firstly, we have presented MoDisco as a generic and extensible MDRE approach intending to facilitate the elaboration of MDRE solutions in many different contexts. Our approach is realized as a ready-to-use framework that is also an official Eclipse project on top of the Eclipse/EMF modeling environment. The framework comes with a set of MDRE components that can be taken, reused and assembled as wished in order to build a final solution. We have shown the relevance and applicability of MoDisco on different real industrial scenarios. It notably allows discovering different kinds of models (mostly structural ones at first, but also behavioral ones thanks to latest research work on fREX) out of already existing software systems. The obtained sets of various and varied discovered models can then be reused for further model federation and comprehension, e.g. using EMF Views as mentioned in the next paragraph.

Secondly, we have described EMF Views as a complementary generic and extensible approach intending to specify, build and manipulate views over sets of existing (and potentially large) models. It is also realized as a dedicated framework implemented on top of the Eclipse/EMF modeling environment. We have shown the relevance and scalability of EMF Views by applying it in the context of a realistic scenario coming from an industrial collaborative project. The framework notably comes with a model virtualization backend that allows, in a given view, referring to different contributing models in a trans-

parent and non-intrusive way. It can be used to define and obtain views federating models that conform to any metamodel(s). Such views can then be navigated and/or queried, as any regular models, in order to get a better comprehension of the modeled systems. In addition, EMF VIEWS also provides a couple of DSLs (in its current version) intending to facilitate the specification of viewpoints and views.

All the presented solutions have been implemented as sets of open source Eclipse plugins released under the EPL [66] license, and are fully available online.

5.2 Impact of the Results

The research work presented all along this manuscript has been conducted mostly within different collaborative projects, involving both academic partners and industrial ones (ranging from large international companies to local Small and Medium-sized Enterprises (SMEs)). As a consequence, the proposed contributions are deeply anchored in real world concerns. In what follows, we introduce 4 of our collaborative projects that are particularly relevant in the context of this manuscript and in which we have been (or still are) strongly involved. For each one of them, we provide a short presentation of the project and its scope, as well as a summarized description of our specific realizations in this project. Notably, we insist on the direct relation with the research contributions proposed in the manuscript. Section 5.2.1 gives an overview of 2 European projects while Section 5.2.2 briefly presents 2 French national projects.

5.2.1 European Collaborative Projects

The ARTIST FP7 Project

The Advanced Software-based Service Provisioning and Migration of Legacy Software (ARTIST) collaborative project [1] is an industrially-supported European initiative (FP7) that started in October 2012 and finished in September 2015 (for a whole duration of 3 years). It was coordinated by ATOS Spain and is consortium was composed of 9 other partners: TECNALIA (Spain), Inria/us (France), Fraunhofer (Germany), TU Wien (Austria), Engineering (Italy), ICCS (Greece), Sparx Systems (Austria), ATC (Greece) and Spikes (Belgium). ARTIST aimed at proposing a software modernization approach based on model-based techniques in order to automate the reverse and forward engineering of legacy applications to the Cloud [143, 30]. To prove the validity of the proposed solution, the project also comes with 4 case studies covering a significant variety of technological environments (Java vs. .NET) and of application domains (environmental hazards, news publication, business process management, public administration).

One of the main expected contributions of ARTIST was notably to provide a MDRE approach and supporting tooling. This has naturally allowed us working on applying our MDRE approach and MoDisco framework as well as pursuing further research around it. More specifically, in this ARTIST context, we were able to really consolidate and stabilize our MDRE conceptual approach as presented in Section 3.2. Moreover, we also had the chance to push further our work on the model discovery of some behavioral aspects of

1. www.artist-project.eu/

existing systems (complementary to the support for their structural aspects). Indeed, this was expressed as an important need for the industrial partners in the project. This directly led us to the research work realized around fREX as described in Section 3.5.

The MegaM@Rt2 ECSEL-H2020 Project

The MegaM@Rt2 collaborative project [2] is a large and industrially-supported European initiative (ECSEL-H2020) that officially started on April 2017 for a planned duration of 3 years. It relies on a wide consortium composed of 27 partners from 6 different national clusters (Sweden, France, Spain, Italy, Finland & Czech Republic) including large companies such as Atos, Thales, Nokia, Volvo and Bombardier. MegaM@Rt2 aims at incorporating methods and tools in order to develop a continuous system engineering and validation approach that can be practically deployed in various industrial domains [1]. To prove the validity of the proposed solution, the project also comes with 9 case studies covering a large variety of potential application areas: aeronautics, railway, warehouse, telecommunication, etc. MegaM@Rt2 deliverables reporting the project's progresses have already been produced during the first year and a half of the project, and new ones are currently being developed for future releases.

One of the main expected contributions of MegaM@Rt2 is notably to propose a runtime $\leftrightarrow$ design time feedback loop that is (re)usable in these different contexts. In order to realize this, scalable model-based methods and tools are being considered to improve the productivity, quality and predictability of the concerned large complex industrial systems. Notably, in this project we propose to refine, extend and apply our model view approach presented in Section 4.2 in order to implement the previously mentioned feedback loop. The overall idea is to specify, build and then query/navigate views federating together very large models representing both runtime and design time aspects of the studied systems. In this context, we have already been able to improve significantly the performances of the EMF Views tooling, as described in Section 4.3 and Section 4.4.

5.2.2 National Collaborative Projects

The TEAP FUI 13 Project

The TOGAF Enterprise Architecture Platform (TEAP) collaborative project was a joint 3-year French collaboration between Capgemini (IT Consulting), DCNS (Naval defense and energy), Obeo (Software company specialized in MDE) and our research team. Its main objective was to provide an enhanced support for the governance of EA. In this context, DCNS identified the need for specializing their EA toolkit to their particular case. They wanted to complement TOGAF [196] in order to include both business process information and requirement specifications [33]. Besides the need for integration within their overall EA solution, DCNS wanted to be able to interconnect the corresponding models with the TOGAF models and to provide partial views depending on some DCNS member profiles (with limited access to given model elements, e.g. for security reasons). In order to do so, they had to specify several viewpoints notably linking EA, requirement and business process metamodels altogether.

2. http://megamart2-ecsel.eu/

The main expected contribution of TEAP was to propose a model federation mechanism allowing to combine heterogeneous models in order to facilitate their overall comprehension by engineers. Thus, in this context, we were able to conduct the initial research and first developments on our model view approach (cf. Section 4.2) and implementing EMF Views framework (cf. Section 4.3). This effort was then continued within the MoNoGe project as introduced in what follows. Associated to this initial work, we notably designed and developed a first version of VPDL as described in Section 4.2.3. It is important to notice that this language is still evolving, and has been enhanced recently in the context of the previously introduced MegaM@Rt2 project (cf. Section 5.2.1).

The MoNoGe FUI 15 Project

The MoNoGe collaborative project was a joint 3-and-a-half-year French collaboration between SOFTEAM (Tool Provider and IT Consulting), DCNS (Naval defense and energy), the LIP6 laboratory (Academic Research) and our research team. Its main objective was to provide a generic lightweight metamodel extension approach to be deployed in industrial environments where rapid and efficient adaptations of the used modeling tools are required [31]. The term *lightweight* was important because the proposed mechanism could not require any kind of model migration or transformation process. Moreover, metamodel extension definitions had to be shareable between different modeling tools supporting the proposed mechanism. In practice, DCNS wanted to extend their reference EA (meta)model with data coming from other EA standards. They also wanted to be able to exchange these metamodel extensions between different modeling tools they use as part of their tool set.

The main expected contribution of MoNoGe was to propose a metamodel extension mechanism and related language that could be applied on any already existing metamodel(s). As a consequence, we considered this project as a direct follow-up of the previously introduced TEAP project. It notably allowed us continuing our research effort around our model view approach (cf. Section 4.2) and related EMF Views framework (cf. Section 4.3). Associated to this work, we notably designed and developed a first version of MEL as described in Section 4.2.3. This was an interesting experience that helped us ensuring the genericity of our solution on a different application scenario (than the TEAP one). At the same time, this allowed us to refine our core virtualization metamodel accordingly (cf. Section 4.2.2).

5.3 Lessons Learned

Even in favorable contexts such as the collaborative projects introduced in Section 5.2, there is no silver bullet when it comes to create, design, develop and promote successful model-based approaches and tools. In our research team, we have a quite long-term experience in this area. From my personal perspective, this has been illustrated in practice by the two technical solutions resulting from the research work presented in this manuscript. These experiences with MoDisco and EMF Views have allowed us to identify several key factors that could also be relevant in the context of other software research and development projects (i.e. not only model-based ones). We share and detail them further in what follows:

— **Using an open source license**. The global choice of adopting an open source approach for publishing our research tools, managing them and communicating around the different project results has been fundamental. It simplifies a lot all common actions among the partners, notably concerning legal aspects (such as the intellectual property) or the transparent exploitation and dissemination of the results (especially as a research team). The fact that the selected open source license, the EPL [66] in our case, allows commercial adaptations and redistributions of the software is also a win-win situation. Indeed, it can help to convince companies to participate and contribute to the open source code base while allowing them to later on adapt the components for their commercial use.
Open source is gaining more and more credibility, particularly in the industrial world. But there is sometimes reluctance, e.g. in some big companies, to really integrate and invest on this relatively new business model (compared to the traditional proprietary model). In this context, innovative SMEs can be easier to convince as open source provides some kind of flexibility that is quite interesting for such lighter organizations.

— **Relying on a reference framework (i.e. EMF)**. For a tool to be stable and reliable, it must be built on solid ground. Thus, being based on an already well-established and recognized framework is a guarantee of a certain quality level. Moreover, due to the shared use of a common framework, this also helps targeting a more widespread audience and facilitate the interoperability with already existing solutions. In the context of MoDisco and EMF Views, the choice of using EMF as the reference modeling framework has been quite obvious since EMF was and is still today the most widely use (open source) framework of this type.
However, while capitalizing on the numerous benefits brought by EMF, we also inherit from his drawbacks. Currently, EMF still has some scalability issues when dealing with very big models and these are open spaces for research and experiments. We have already faced some of these problems in the case of EMF Views for instance, cf. Chapter 4. But globally, such drawbacks are balanced by the advantages of relying on a mature common framework like EMF.

— **Following structured development processes**. When building complex solutions, a well-defined development process (milestones, bug tracking, version control, tests, coding style guidelines, etc.) is recommended.
In the case of MoDisco, being part of Eclipse (with its well-structured procedures [61]) really helped in this matter. We were not following all the best practices at the beginning, but the growing complexity of the project and increasing number of users forced us to adopt them. Also, the released code being open source, the work of the development team had to be better and the reliability of the built solution improved. In addition, the Eclipse Simultaneous Release yearly cycle forced the project team to release and update tool versions (and related documentation) on a regular basis. However deploying such a process can be quite heavy and so generally cannot be supported by a research group alone, making the partnership with a company a requirement (this is what we did in the context of MoDisco, cf. the next item on this topic).
In the case of EMF Views, we have not yet been as far as what we did with MoDisco. This situation mostly comes from limitations in terms of available development resources from our side. This is also reinforced by the fact that we have not yet been able to find a partner company to collaborate with us on this

solution. However, even if EMF Views is not an Eclipse official project (contrary to MoDisco), we try to reach a certain level of structure and organization by following common best practices (development in branches, use of pull requests, regular commits, continuous integration, documentation, etc.).

— **Integrating a widely recognized community**. Open source itself is not enough to attract new users as many open source projects are half-dead, of relatively poor quality or adopted by a very limited set of users. Instead, being an official project within an acknowledged community can give a lot more visibility. For instance, the Eclipse brand is well-known and considered as a guarantee of high-quality by many practitioners.

Thus, the effective integration of MoDisco into such an industrially recognized project/community had a very beneficial impact over the general perception of the tool by IT professionals. We recommend to try identifying such a well-known community in its domain to benefit from the visibility and interactions with its members. However, this is a bidirectional effort: the research team also needs to strongly invest on the community, which we still have not been able to do in the case of EMF Views.

During many years, we regularly attended the Eclipse conferences and tried to participate to and use other Eclipse projects beyond ours. For different reasons (e.g. due to other time-consuming activities and to more limited resources), we had to lower down this dissemination effort since a couple of years. We observed that this directly impacted the activity level around the MoDisco project (which also started to decrease progressively) as well as the dissemination level of EMF Views (which is not as good as it could be).

— **Collaborating with an industrial partner.** Companies often consider too risky to use research prototypes due to their lack of proper user interface, documentation, completeness, support, etc. The problem is that research groups generally have limited resources that cannot be devoted to work on such non-core research activities. As a consequence, research prototypes very often remain simple proofs-of-concept. Because of this situation, research groups can miss the opportunity of having a larger tool user base, along with its associated benefits (e.g. empirical validation, feedback, visibility, collaboration opportunities).

To avoid this situation for MoDisco, we had the possibility to partner with a technology provider (Mia-Software [190], part of the Sodifrance Group) to industrialize MoDisco. This allowed us to focus on doing research while staying sure to work on actually relevant industrial challenges. Instead, the industrial partner took over the traditional software development and maintenance tasks. This strengthened its presence and visibility within the Eclipse community and surrounding open source market, and has also been used at the time to create a business network (e.g. for providing training or special consulting services).

Such a partnership has worked well so far in the case of MoDisco, even though it is not easy to maintain in the long run. However, we have still not achieved this as far as EMF Views is concerned (notably because of the different reasons mentioned in the previous items). This highlights the real difficulty of finding the appropriate company, setting up the proper process and, more than all, getting the required initial funding to initiate and then support such a collaboration during several years.

— **Being supported and rewarded by a host laboratory or institution for the**

technology transfer effort. It is necessary to benefit from dedicated resources/structures, offered by the hosting institution, in order to help the research team during such an industrialization process. Indeed, this process requires significant (and non-core research) additional effort and knowledge in legal, financial, logistic or commercial aspects, which are not competences always found in research. As an example in our case, IMT Atlantique (formerly Ecole des Mines de Nantes) [3], the LS2N [4] and previously Inria [5] have been providing some base support to their different research teams via *Innovation and Technology Transfer* entities.
However, despite of this, the required extra effort is not always rewarded at its real value from a research career perspective, highlighting the more global problem of current evaluation criteria in research organizations. Consequently (and mostly due to time or resource limitations), many research teams voluntarily ignore this aspect of what should also be part of their normal working activity.
In the case of MoDisco (mostly) and EMF Views (to a much lower extent so far), we think the realized technology transfer effort has been somehow beneficial in terms of how our research is globally perceived by our host laboratory and institutions. This notably allowed us to be identified and advertized as a research group having strong interactions with the industrial world, and notably with the European and French/regional ecosystems. However, at more individual levels, this has not been directly rewarded in terms of internal financial support (e.g. to develop further research work) or academic recognition (e.g. to access to other responsibilities) for instance.

To summarize, from a research perspective, the main challenge at the end is always finding the right balance between the fundamental nature of the research activity and the expected (and evaluated) results in terms of scientific publications, innovative conceptual solutions, corresponding prototypes, etc. In the short-term, spending a lot of effort on technical developments may seem counterproductive compared to more immediate results that can be obtained if focusing only on publishing scientific papers. However, in the medium- or long-term, a successful open source tool may be one of the biggest assets a research team may produce, which is particularly true in an engineering domain such as Software Engineering. Later on, this can notably be the source of many benefits for the team like getting more interesting contacts, collaboration opportunities (meaning more projects/contracts) and so potential available resources for continuously exploring different research lines.

5.4 Perspectives and Future Work

The study of the state-of-the-art in terms of research (cf. Section 3.1 and Section 4.1), as well as our past and present experiences of collaboration with industrial partners (cf. Section 5.2), have shown that there is a real need for generic model-based solutions dealing with software reverse engineering and comprehension problems. We have also seen that developing and promoting such solutions is not a trivial activity and requires a significant effort (cf. Section 5.3).

3. http://www.imt-atlantique.fr/en
4. https://www.ls2n.fr/?lang=en
5. https://www.inria.fr/en/centre/rennes

Anyway, there are many interesting (and open) research challenges that could still be addressed in the future related to these topics of MDRE and model federation/comprehension via model views. In this last section of the manuscript, we present some perspectives and associated future work that could be explored regarding the two conceptual approaches and technical solutions we proposed (cf. Section 5.4.1 and Section 5.4.2, respectively). Finally, we end by proposing a higher-level vision over possible future applications and related challenges in this Software Engineering area (cf. Section 5.4.3).

5.4.1 Model Driven Reverse Engineering

The future work around our proposed MDRE approach and implementing framework MoDisco includes the following:

- **Scalability improvement.** As far as MoDisco is concerned, a significant effort has already been devoted to the general scalability challenge, notably from the *Model Discovery* perspective. The objective is to efficiently deal with the very large models typically obtained when reverse engineering large code bases. The current version of MoDisco has already proved to work on real projects of small and medium size (according to an industrial scale). However, more work has to be done to improve the performance when tackling very large systems.
  - **Parallelization.** For instance, we could rely on various existing parallelization techniques such as the ones that are related to model transformation [198] as particularly relevant in our case. We could directly benefit from them when model-to-model transformations are required in the MDRE process, but also adapt them to the context of initial model discovery (text-to-model). The general execution time could be significantly reduced because some transformation/discovery rules/patterns could be executed in parallel when possible (instead of being systematically executed sequentially).
  - **Lazy loading.** Another relevant technique for improving the overall scalability of our solution is lazy loading. It consists in minimizing the in-memory footprint by loading a model element only when accessed or requested (e.g. by a model discoverer or following model transformation). For example, solutions like NeoEMF [49] provide such a capability for efficiently loading/storing models from/into various kinds of databases. Some related experiments have already been performed, reusing the MoDisco Java metamodel and discoverer in order to evaluate the use of advanced prefetching and caching techniques on various Java models [50]. They could be used to refine the current version of these MoDisco components

- **Reverse engineering of system behavioral aspects.** In addition to the base support for structural aspects of systems, the initial research work performed around fREX (and the corresponding Java-to-fUML mapping) has revealed interesting findings.
  - **fUML extension for Java (and others).** From a fUML perspective, several aspects of the Java language are currently challenging to be represented such as dynamic dispatching, generics (for classes and interfaces), exceptions and assertions, external libraries or corresponding reflection aspects. These con-

cepts, or equivalent ones in other programming languages, are currently not directly supported by fUML. Thus, we could explore how fUML may be extended to provide a more complete set of concepts to be used to map more programming language concepts directly to fUML. For instance, the Java-to-fUML mapping and implementing model transformation still need to be improved to support more and more (behavioral) aspects of the complete Java language (and not be restricted to MiniJava anymore).

- **Support for additional languages.** From a more general perspective, the current version of fREX only comes with a single language support for Java via our proposed Java-to-fUML mapping. For validation purposes, it would be very interesting to enlarge the scope of the framework by covering another widely used object-oriented language, such as C# or C++ for instance. For the sake of completeness, the study may also be extended to a few non object-oriented programming languages (e.g. C, or even COBOL or FORTRAN as quite frequently found by our industrial partners in modernization projects). The capability to reverse engineer multiple programs written with different languages into a single fUML model (i.e. at a same abstraction level) is another relevant aspect that could be studied as well.
- **Dynamic program analysis.** In parallel to the work around fREX, a PhD thesis recently started in our group on the topic of extending our MDRE approach for supporting dynamic program analysis. The initial objective was to retrieve runtime information out of system execution traces (produced via automated code instrumentation). Then, this data can be related to structural models of this same system (discovered thanks to MoDisco) in order to perform some impact analysis. The proposed architecture and first obtained results are promising [131]. However, interesting challenges also appeared in terms of general scalability (cf. also the previously related item in this section) or used code instrumentation techniques.

- **Reverse engineering of system mixed aspects.** When reverse engineering existing software systems, some aspects to be better comprehend actually require to consider both structural (static) and behavioral (dynamic) information. This usually implies addressing complex problems because of complementary dimensions to be properly recovered and then related together.
  - **Graphical User Interfaces.** GUIs have an important structural part (e.g. a combination of widgets) but are also strongly characterized by how they are supposed to behave and react at runtime (e.g. caught events and triggered actions). Moreover, depending on the used language or framework as well as on the quality of the source code, this distinction is not always easy to appreciate. Thus, reverse engineering such GUIs in a fully automated way is quite challenging. Currently, our solution focuses on generic components and does not provide any features specific to GUIs. To overcome this, we could extend our approach by relying on some already existing model-based work in this area [177, 96].
  - **Non-Functional Properties.** While most of the MDRE approaches intend to address the reverse engineering of functional properties (i.e. what the systems do or provide as features), non-functional properties (e.g. performance, quality, security, usability, etc.) appear to be more complicated to tackle in a sys-

tematic way. To this intent, an existing approach proposes to reverse engineer goal models from legacy code (notably from Java code, by exploiting relevant information such as used names or available comments for instance) [218]. Such goal models are supposed to capture the initial requirements of the system stakeholders, and so can then be used to (partially) derive some system non-functional properties. This proposed reverse engineering process could be adapted and refined in the context of our MDRE approach.

- **Generic integration.** In addition to the two particular cases from the two previous items, it could be interesting to try generalizing the problem. Thus, we could work on extending our generic approach to also provide support for better integrating the reverse engineering of such different system aspects. Moreover, the needed information could come from various interrelated source artifacts instead of single ones (e.g. the business logic of an application can be scattered between the actual database and the forms built on top of it, instead of coming from a single source code file). This requires to be able to combine the different obtained models in a certain way in order to represent the complete logic. To this intent, the ongoing work around our generic model view approach and implementing framework EMF Views appears to be particularly relevant (cf. also next Section 5.4.2).

- **Additional technical developments.** Finally, there are also some more technical challenges that could be addressed in the future related to MoDisco.
  - **Ready-to-use sets of components.** The current version of the MoDisco framework provides a set of generic reusable components that have to be combined in different ways (eventually also with other external components) in order to support particular MDRE scenarios. We could extend the framework by proposing in addition some pre-configured sets of components addressing common needs in terms of reverse engineering (e.g. chaining a given model discoverer with related model transformations and other components, such as what we did with fREX for instance).
  - **Programming style.** The current version of the Java model discoverer produces Java models that do not contain any specific information related to the coding styles used within the original source code. However, it does preserve element names, comments as well as some base formatting data. Such data, along with an additional *code style* model provided by the users (or even automatically initialized from various code samples) for instance, could be reused in order to configure the code generators [176]. This could be particularly interesting when automatically migrating to new technologies while style keeping a given company style for coding.

5.4.2 Model Federation and Comprehension

The future work around our proposed model view approach and implementing framework EMF Views includes the following:

- **Scalability improvement.** As far as EMF Views is concerned, a significant effort has already been devoted to improving the general scalability of the solution in

the context of views federating several large and heterogeneous models. Notably, we obtained interesting results when computing/initializing the view-specific information and loading corresponding views by using several database backends. However, there is still room for interesting improvements.

- **Transformation techniques.** In the next steps of our work, we will push further our experiments on view querying by also testing well-known model transformation tools on our views, such as ATL [116] or VIATRA [203]. We could reuse information available from the view in order to delegate parts of the transformation computations directly to the underlying database backends. We plan to do this by integrating scalable query/transformation approaches related to these transformation languages, such as Gremlin-ATL [48] for instance.
- **Integration with other environments.** Another possible work is to extend our approach to additional model querying environments, and then check the possible impacts in terms of scalability for the overall solution. For instance, we could experiment on using our model view solution with Epsilon [69], as they both rely on EMF. Epsilon provides its own language for handling/querying models, named Epsilon Object Language (EOL), and notably supports many common OCL operations.
- **Incrementality.** Moreover, we could also study how incremental querying techniques could be integrated in our approach (in addition to the reuse of some already realized optimizations, at OCL-level for instance [212]). For instance, we could experiment on combining our view solution with EMF-IncQuery that provides capabilities for incremental pattern matching [16]. This way, we could potentially obtain significant performance gains when executing several consecutive queries over a same view.

- **Advanced features.** Our approach and implementing tool already provide the required base capabilities for defining, building and handling model views. However, EMF Views could be extended in different ways in order to support more advanced model view features. To this intent, there are several interesting and relevant directions in which we could experiment in the future. Note that some of these research lines (e.g. view update or validation) could directly benefit from a more formal specification of the different operations we provide (cf. the description of our core virtualization metamodel and provided DSLs in Chapter 4). This is already work in progress from our side.
  - **View update strategies.** As a next line of work, we could experiment on an enhanced model view mechanism that would deal (at least partially in a first attempt) with the well-known view update problem [139]. Quite recent work allowed to provide some model view update capabilities in the particular (transformation-based) context of the ModelJoin tool [35]. However, the current EMF Views implementation is limited to basic updates only (e.g. when an attribute value is modified in the view, the corresponding contributing model can be updated accordingly). In the same vein, we could extend it by proposing some base update strategies to be selected (and eventually combined) by the engineers specifying or using the viewpoints/views.
  - **View validation support.** Some more advanced validation support for the specified viewpoints and views is also required. The current version of EMF

Views already comes with syntactic validation at viewpoint/view definition-level (when using the provided DSLs). However, there is no further validation as far as the semantics of the underlying operators is concerned. For instance, we should be able to ensure that views federating several different models are not breaking any constraints that could be expressed at individual model-level.

  - **Alternative DSLs for views.** Our proposed approach/tooling already provides a couple of DSLs for specifying model viewpoints/views. We made the choice of offering textual languages as they are usually well-suited to software engineers or developers. However, we could explore other alternative languages for view definition and manipulation, notably graphical ones, to evaluate which one would offer a better usability depending on the context. To go further, we could also experiment on translating and/or composing such view definitions when expressed using different languages.
  - **View snapshot persistence.** Our model view approach has been designed to be lightweight and as non-intrusive as possible. Thus, the data coming from the various contributing models is never duplicated and only accessed via in-memory proxies. In addition to this default behavior, we could also offer the option to persist complete snapshots of model views at given points in time. This could be relevant in case users want to export them to other tools (e.g. to external modeling frameworks), or to avoid a costly recomputation when not strictly needed.

- **Additional technical developments.** Finally, there are also some more technical challenges that could be addressed in the future related to EMF Views.
  - **New practical applications from the industry.** In the context of the MegaM@Rt2 project (cf. Section 5.2.1), we already applied our model view approach to support a runtime-to-design time feedback loop at software level. In the future we want to apply EMF Views in the context of other MegaM@Rt2 use cases, but this time to also cover aspects outside of the pure software world. For example, we already have plans to build views in scenarios involving CPSs. The objective would be to trace the architectural models of an industrial system (including a software part, but also a hardware one) with runtime models representing the configuration and running of corresponding physical entities.
  - **Integration within Modeling tools.** The current version of EMF Views is distributed as a set of Eclipse plugins that notably rely on EMF as its underlying modeling framework. Thus, it can already be easily installed in an Eclipse workbench and work along with other existing EMF-based components. As of today, the EMF Views open source solution is designed and developed in an independent way. However, in the medium or longer term, we could envision a potential deeper integration within larger EMF-based modeling tools (such as Papyrus [72] for instance). This could be relevant in the context of our research work on several of the previously mentioned topics (e.g. scalability, view update or validation). More generally, this could also help promoting the use EMF Views and disseminate it outside of the academic world.

5.4.3 Overview

As already mentioned in Section 2.1.3, software systems and particularly CPSs (i.e. involving both software and hardware elements communicating together) are becoming more and more present/important within the industry. This is being reinforced by several ongoing and industrially-supported initiatives under the *Industry 4.0* [97] or *Industrie du Futur* [93] umbrellas (for instance). With the current trends on Cloud Computing [6], Fog Computing and Internet of Things (IoT) [23] or Big Data and Machine Learning [213], we can envision (many) more and more complex industrial software systems to be developed, maintained and then evolved in the future. As shown in Figure 5.1, this is not coming without great challenges that also directly concern reverse engineering and comprehension activities as treated all along this manuscript.

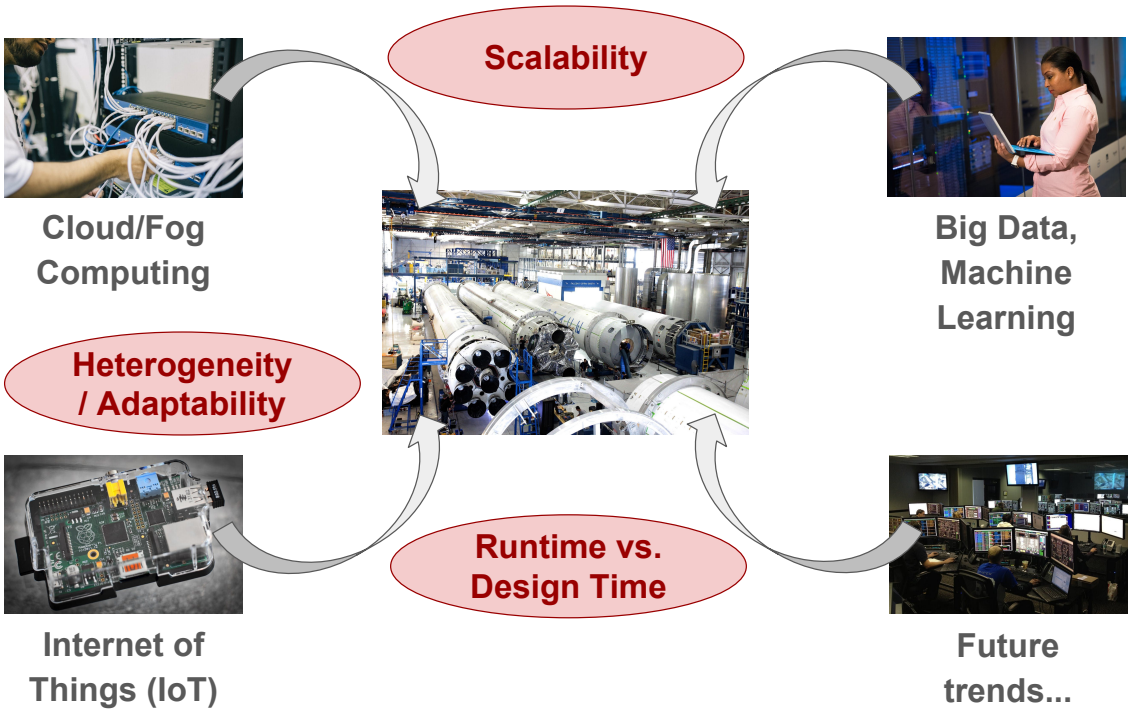

Figure 5.1 – More and more software systems to be developed, maintained and evolved in the future (e.g. Industry 4.0).

The deployment of large Cloud infrastructures inside factories, implying the handling and analysis of possibly huge amounts of data, makes the supporting systems even bigger and more complex to handle. Thus, scalability will be again critical when maintaining and migrating both these supporting systems and their related data. In addition, multiple types of devices (e.g. captors/sensors, actuators) interoperating via local heterogeneous (Fog) networks are more and more frequently encountered both inside and outside factories. The reverse engineering and comprehension support for the corresponding systems will have to be able to adapt in order to cope with this new dimension of heterogeneity. Finally, the physical parts of these industrial CPSs make *temporality* a key property to be taken into account. Indeed, at any moment in time, what happens at the hardware level at runtime is critical and has to be correctly reflected (and handled) at the software level. As a consequence, when maintaining and evolving such systems, capabilities to deal with their behavioral (runtime) aspects will be strongly relevant and needed.

Glossary

ADM Architecture Driven Modernization. 30, 37, 40, 50, 52

AQL Acceleo Query Language. 93

ASTM Abstract Syntax Tree Metamodel. 50, 52

ATL AtlanMod Transformation Language. 73, 118

BPM Business Process Management. 30

BPMN Business Process Model and Notation. 30, 32, 41

CDO Connected Data Objects. 31, 65, 115, 117, 123

CIM Computation Independent Model. 28

CPS Cyber-Physical System. 18, 27, 81, 140, 141

DSL Domain Specific Language. 27, 32, 45, 82, 91, 95, 96, 97, 100, 101, 104, 105, 106, 108, 112, 114, 127, 129, 139, 140

EA Enterprise Architecture. 30, 131, 132

EBNF Extended Backus-Naur Form. 24, 25

ECL Epsilon Comparison Language. 90, 114

Ecore Eclipse Modeling Framework (EMF) (meta)metamodel. 24, 32, 88, 89, 91

EMC Epsilon Model Connectivity. 90

EMF Eclipse Modeling Framework. 16, 17, 23, 30, 31, 32, 33, 36, 51, 52, 61, 65, 82, 89, 90, 92, 93, 108, 113, 115, 117, 118, 123, 129, 133, 139, 140

EML Epsilon Merging Language. 90

EMOF Essential MOF. 32

EMP Eclipse Modeling Project. 31, 32, 51, 73

EOL Epsilon Object Language. 139

EPL Eclipse Public License. 30, 58, 130, 132

EVL Epsilon Validation Language. 90

fUML Foundational UML Subset. 17, 69, 70, 71, 72, 73, 74, 77, 78, 136, 137

HOT Higher-Order Transformation. 26

IDE Integrated Development Environment. 31

IoT Internet of Things. 140

JavaEE Java Platform, Enterprise Edition. 53, 55, 62

JDT Eclipse Java Development Tools. 31, 54

JSP JavaServer Pages. 53, 55

KDM Knowledge Discovery Metamodel. 50, 52, 54

LOC Lines of Code. 58, 61, 66

MDA Model Driven Architecture. 28, 29, 30, 32, 33, 94

MDE Model Driven Engineering. 13, 15, 16, 18, 20, 23, 24, 25, 27, 28, 33, 36, 37, 39, 40, 41, 43, 45, 46, 48, 49, 50, 65, 79, 81, 83, 131

MDRE Model Driven Reverse Engineering. 15, 16, 17, 20, 30, 32, 36, 37, 39, 40, 41, 42, 43, 44, 48, 49, 50, 51, 53, 55, 56, 57, 58, 62, 64, 65, 66, 67, 68, 69, 71, 72, 79, 127, 129, 130, 135, 136, 137, 138

MEL Metamodel Extension Language. 106, 132

MOF MetaObject Facility. 24, 29, 30, 32

MOF2Text MOF Model to Text Transformation Language. 30

OCL Object Constraint Language. 29, 32, 53, 57, 86, 89, 91, 92, 94, 111, 118, 119, 139

OMG Object Management Group. 28, 29, 30, 32, 33, 37, 40, 50, 52, 64, 94

OSGI Open Services Gateway Initiative. 31

OSM Orthographic Software Modeling. 92

OWL Web Ontology Language. 92

PDE Plug-in Development Environment. 31

PIM Platform Independent Model. 28

PSM Platform Specific Model. 28, 30

QVT Query/View/Transformation. 30

RCP Rich Client Platform. 31

SMEs Small and Medium-sized Enterprises. 130, 133

SMM Structured Metrics Metamodel. 50, 52, 62, 63, 64

SQL Structured Query Language. 86, 91, 95, 104, 117

SUM Single Underlying Model. 92

SysML Systems Modeling Language. 30, 32, 41

TGGs Triple Graph Grammars. 93

UAF Unified Architecture Framework. 30

UML Unified Modeling Language. 15, 17, 25, 30, 32, 39, 41, 52, 53, 70, 71, 76, 78, 86, 89, 91, 92, 94

VM Virtual Machine. 69, 70, 72, 73, 74, 77

VPDL ViewPoint Description Language. 104, 105, 106, 115, 131

VSM Viewpoint Specification Model. 93

W3C World Wide Web Consortium. 55

XMI XML Metadata Interchange. 29, 31, 32, 39, 66, 68, 108, 117, 123, 124, 126

XML EXtensible Markup Language. 24, 25, 39, 53, 55, 62, 92, 108, 109

Contents

1 Introduction and Context **13**

1.1 Introduction 13

1.2 Problem Statement 14

1.3 Global Approach 16

1.4 Proposed Contributions 16

1.5 Thesis Context 18

1.6 Scientific Production 19

1.7 Outline 20

2 Background **23**

2.1 Modeling and Model Driven Engineering (MDE) 23

2.1.1 General Definition 23

2.1.2 Core Concepts 24

2.1.3 Challenges 27

2.2 Modeling Standards and Techniques 28

2.2.1 OMG's Model Driven Architecture (MDA) 28

2.2.2 Related Standard Specifications 29

2.3 Modeling in/with Eclipse 30

2.3.1 The Eclipse Open Source Platform 31

2.3.2 The Eclipse Modeling Project (EMP) 31

2.3.3 The Eclipse Modeling Framework (EMF) 32

2.4 Conclusion 33

3 Model Driven Reverse Engineering **35**

3.1 State of the Art and Challenges 36

3.1.1 Overview 37

3.1.2 Specific Reverse Engineering Solutions 39

3.1.3 Generic Reverse Engineering Platforms and Frameworks 40
3.1.4 Challenges 41
3.2 Proposed Conceptual Approach 42
3.2.1 Overall Approach 42
3.2.2 Model Discovery 45
3.2.3 Model Understanding 47
3.2.4 Main Benefits 48
3.3 The MoDisco framework 50
3.3.1 Project Overview 50
3.3.2 Infrastructure Layer 52
3.3.3 Technology Layer 54
3.3.4 Use Cases Layer 56
3.3.5 Extending MoDisco 56
3.4 Evaluation 57
3.4.1 Research Questions (RQs) 57
3.4.2 MDRE Concrete Use Cases 58
3.4.3 Performance Benchmarks 65
3.5 The fREX Component 69
3.5.1 Motivation 70
3.5.2 Proposed Framework 71
3.5.3 The Java-to-fUML Example 74
3.5.4 Possible Applications 77
3.6 Conclusion 79

4 Model Federation and Comprehension 81
4.1 State of the Art and Challenges 82
4.1.1 General Definitions 83
4.1.2 Characterization of Model View Approaches 84
4.1.3 Description of Model View Approaches 88
4.1.4 General Challenges for the Community 94
4.2 Proposed Conceptual Approach 96
4.2.1 Overall Approach 97
4.2.2 Core Virtualization (Weaving) Metamodel 102
4.2.3 Viewpoint/View Specification DSLs 104
4.2.4 Integration With Model Persistence Solutions 108

4.2.5 Main Benefits . . . 112
4.3 The EMF VIEWS framework . . . 113
4.3.1 Implementation Overview . . . 113
4.3.2 Tooling Support . . . 115
4.3.3 Integration With Model Persistence Solutions . . . 115
4.4 Evaluation . . . 119
4.4.1 Research Questions (RQs) . . . 119
4.4.2 Practical Use Case . . . 120
4.4.3 Objectives . . . 122
4.4.4 Process . . . 122
4.4.5 Scalability Benchmarks . . . 123
4.5 Conclusion . . . 127

5 Conclusion **129**
5.1 Summary . . . 129
5.2 Impact of the Results . . . 130
5.2.1 European Collaborative Projects . . . 130
5.2.2 National Collaborative Projects . . . 131
5.3 Lessons Learned . . . 132
5.4 Perspectives and Future Work . . . 135
5.4.1 Model Driven Reverse Engineering . . . 136
5.4.2 Model Federation and Comprehension . . . 138
5.4.3 Overview . . . 141

List of Tables

3.1 An overview of existing MDRE approaches - Specific solutions. 38

3.2 An overview of existing MDRE approaches - General-purpose solutions . 38

3.3 Mapping between MiniJava and fUML. 76

4.1 A comparison of existing model view approaches (✓=feature supported,∅=not applicable,w=Wizard). 88

4.2 Time (in minutes) to create the view-specific models (weaving models). . 124

4.3 Size (in megabytes) of the view-specific models (weaving models) on disk. 124

4.4 Time (in seconds) to load the view. 125

4.5 Time (in seconds) to iterate over the full content of the view. 125

4.6 Time (in seconds) to run the OCL query (1). 126

4.7 Time (in seconds) to run the OCL query (2). 126

4.8 Time (in seconds) to run the OCL query (3). 126

List of Figures

1 Un écosystème basé sur les modèles pour la rétro-ingénierie et compréhension des systèmes logiciel. . . . 9

1.1 The three phases of a software modernization and/or migration project. . . 14

1.2 An ecosystem for the model-based reverse engineering and comprehension of existing Software systems. . . . 17

2.1 System, model and metamodel. . . . 24

2.2 The three-level Modeling stack and similar structuring in the EBNF & XML technical spaces. . . . 25

2.3 Model-to-model transformation. . . . 26

2.4 Model-to-text Transformation, also commonly known as Code Generation 26

2.5 MDA vision: CIM, PIM and PSM. . . . 29

2.6 Simplified version of the Ecore metametamodel from EMF. . . . 33

3.1 MDRE framework architecture. . . . 44

3.2 General principle of Model Discovery. . . . 45

3.3 Two-step approach for Model Discovery. . . . 46

3.4 General principle of Model Understanding. . . . 47

3.5 A J2EE/Java example of a Model Understanding phase. . . . 49

3.6 Overview of the Eclipse MoDisco project. . . . 51

3.7 Architecture of the MoDisco framework. . . . 52

3.8 The generic MoDisco Model Browser, customized for the UML metamodel. 54

3.9 Example of a discovered Java model opened in the MoDisco Model Browser. . . . 55

3.10 The generic MoDisco Workflow, used for a sample Java refactoring process. 56

3.11 Java model before (left) and after (right) refactoring, using the MoDisco Model Browser to show the effects of the first refactoring. . . . 59

3.12 An example of Java application refactoring rule for type replacement. . . . 60

3.13 Overall process of the model driven Java application refactoring. . . . 60

3.14 Sample Java code before (top) and after (bottom) refactoring. 61
3.15 Overall process of the model driven code quality evaluation. 63
3.16 The MoDisco-based Mia-Quality model editor. 64
3.17 Some quality measurements obtained as output of Mia-Quality. 65
3.18 Benchmark on the size of discovered Java models. 66
3.19 Benchmark on the time and memory footprint of a Java discovery process. 68
3.20 Benchmark on the time repartition during a Java discovery process. 69
3.21 Overall architecture of the fREX framework. 72
3.22 Java code expressed and executed by means of fUML. 75

4.1 A terminology for model view approaches. 83
4.2 A feature model for model view approaches. 85
4.3 A simple example of model viewpoint and view. 99
4.4 Overview of the Model View approach. 100
4.5 Viewpoint creation at design time. 101
4.6 View initialization at runtime 102
4.7 Core virtualization metamodel of our approach. 103
4.8 A conceptual approach for integrating model view and model persistence capabilities. 109
4.9 Optimizing model view querying by delegating to model persistence backends. 112
4.10 Current implementation of EMF Views (design time). 114
4.11 Current implementation of EMF Views (runtime). 114
4.12 Screenshot of an Eclipse workbench with EMF Views installed. 116
4.13 Running use case from the MegaM@Rt2 industrial project. 121
4.14 Concrete example of a view in MegaM@Rt2 (based on the use case from Figure 4.13). 121

5.1 More and more software systems to be developed, maintained and evolved in the future (e.g. Industry 4.0). 141

Bibliography

[1] W. Afzal, H. Bruneliere, D. Di Ruscio, A. Sadovykh, S. Mazzini, E. Cariou, D. Truscan, J. Cabot, A. Gómez, J. Gorroñogoitia, L. Pomante, and P. Smrz. The MegaM@Rt2 ECSEL project: MegaModelling at Runtime - Scalable Model-based Framework for Continuous Development and Runtime Validation of Complex Systems. *Microprocessors and Microsystems*, 61:86–95, 2018. 18, 131

[2] Z. Al-Shara, F. Alvares, H. Bruneliere, J. Lejeune, C. Prud Homme, and T. Ledoux. CoMe4ACloud: An End-to-End Framework for Autonomic Cloud Systems. *Future Generation Computer Systems*, 86:339–354, 2018. 18

[3] A. Alnusair and T. Zhao. Towards a Model-driven Approach for Reverse Engineering Design Patterns. In *2nd International Workshop on Transforming and Weaving Ontologies in MDE (TWOMDE 2009), co-located with the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS2009)*, volume 531, page 16, 2009. 38, 39

[4] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt. State-based Model Slicing: A Survey. *ACM Computing Surveys (CSUR)*, 45(4):53, 2013. 78

[5] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr. Efficient Model Synchronization with View Triple Graph Grammars. In *10th European Conference on Modelling Foundations and Applications (ECMFA 2014)*, pages 1–17. Springer, 2014. 94

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010. 141

[7] C. Atkinson, P. Bostan, D. Brenner, G. Falcone, M. Gutheil, O. Hummel, M. Juhasz, and D. Stoll. Modeling Components and Component-Based Systems in KobrA. In *The Common Component Modeling Example*, pages 54–84. Springer, 2008. 93

[8] C. Atkinson and T. Kühne. The Essence of Multilevel Metamodeling. In *4th International Conference on the Unified Modeling Language (UML'01)*, pages 19–33. Springer, 2001. 24

[9] C. Atkinson and T. Kuhne. Model-Driven Development: a Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. 24

[10] C. Atkinson, D. Stoll, and P. Bostan. Orthographic Software Modeling: a Practical Approach to View-based Development. In *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2008)*, pages 206–219. Springer, 2008. 92

[11] Atlanmod.org. EMF Views, 2018. URL: `http://www.atlanmod.org/emfviews/`. 115

[12] F. Barbier, S. Eveillard, K. Youbi, O. Guitton, A. Perrier, and E. Cariou. Model-driven Reverse Engineering of COBOL-based Applications. In *Information Systems Transformation*, pages 283–299. Elsevier, 2010. 38, 39

[13] L. Baresi and M. Pezzè. A Toolbox for Automating Visual Software Engineering. In *International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*, pages 189–202. Springer, 2002. 37

[14] I. D. Baxter and M. Mehlich. Reverse Engineering is Reverse Forward Engineering. In *4th Working Conference on Reverse Engineering (WCRE 1997)*, pages 104–113. IEEE, 1997. 6, 14

[15] K. Bennett. Legacy Systems: Coping with Success. *IEEE Software*, 12(1):19–23, 1995. 5, 13

[16] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental Evaluation of Model Queries over EMF models. In *ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 76–90. Springer, 2010. 139

[17] A. Bergmayr, H. Bruneliere, J. L. C. Izquierdo, J. Gorronogoitia, G. Kousiouris, D. Kyriazis, P. Langer, A. Menychtas, L. Orue-Echevarria, C. Pezuela, et al. Migrating Legacy Software to the Cloud with ARTIST. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 465–468. IEEE, 2013. 71

[18] A. Bergmayr, M. Grossniklaus, M. Wimmer, and G. Kappel. JUMP - From Java Annotations to UML Profiles. In *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, pages 552–568. Springer, 2014. 76

[19] J. Bézivin. On the Unification Power of Models. *Software & Systems Modeling*, 4(2):171–188, 2005. 5, 14, 24

[20] J. Bézivin. Model Driven Engineering: An Emerging Technical Space. In *Generative and Transformational Techniques in Software Engineering*, pages 36–64. Springer, 2006. 23, 24

[21] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001. 28

[22] G. Blair, N. Bencomo, and R. B. France. Models@ Run. Time. *IEEE Computer*, 42(10), 2009. 28

[23] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. Fog Computing: A Platform for Internet of Things and Analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014. 141

[24] M. Brambilla, J. Cabot, and M. Wimmer. Model-driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*, 3(1):1–207, 2017. 6, 14, 23, 27, 28

[25] L. C. Briand, Y. Labiche, and J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006. 38, 39, 78

[26] H. Bruneliere, Z. Al-Shara, F. Alvares, J. Lejeune, and T. Ledoux. A Model-based Architecture for Autonomic and Heterogeneous Cloud Systems. In *8h International Conference on Cloud Computing and Services Science (CLOSER 2018)*, 2018. 18

[27] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer. A Feature-based Survey of Model View Approaches. *Software & Systems Modeling*, pages 1–22, 2017. 7, 15, 18, 83, 84, 88

[28] H. Bruneliere, J. Cabot, C. Clasen, F. Jouault, and J. Bézivin. Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In *6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, pages 32–47. Springer, 2010. 18

[29] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A Model Driven Reverse Engineering Framework. *Information and Software Technology*, 56(8):1012–1032, 2014. 32, 38, 120

[30] H. Bruneliere, J. Cabot, J. L. C. Izquierdo, L. Orue-Echevarria, O. Strauss, and M. Wimmer. Software Modernization Revisited: Challenges and Prospects. *Computer*, 48(8):76–80, 2015. 18, 79, 130

[31] H. Bruneliere, J. Garcia, P. Desfray, D. E. Khelladi, R. Hebig, R. Bendraou, and J. Cabot. On Lightweight Metamodel Extension to Support Modeling Tools Agility. In *11th European Conference on Modelling Foundations and Applications (ECMFA 2015)*, pages 62–74. Springer, 2015. 82, 90, 132

[32] H. Bruneliere, F. Marchand de Kerchove, G. Daniel, and J. Cabot. Towards Scalable Model Views on Heterogeneous Model Resources. In *ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*, pages 334–344. Springer, 2018. 108

[33] H. Bruneliere, J. G. Perez, M. Wimmer, and J. Cabot. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In *The 34th International Conference on Conceptual Modeling (ER 2015)*, pages 317–325. Springer, 2015. 82, 90, 131

[34] E. Burger, J. Henss, M. Küster, S. Kruse, and L. Happe. View-based Model-driven Software Development with ModelJoin. *Software & Systems Modeling*, 15(2):473–496, 2016. 91

[35] E. Burger and O. Schneider. Translatability and Translation of Updated Views in ModelJoin. In *International Conference on Theory and Practice of Model Transformations (ICMT 2016)*, pages 55–69. Springer, 2016. 139

[36] Business Informatics Group - TU Wien. Moliz project, 2018. URL: `http://www.modelexecution.org`. 73

[37] G. Canfora, A. Cimitile, and M. Munro. RE2: Reverse-Engineering and Reuse Re-Engineering. *Journal of Software Maintenance: Research and Practice*, 6(2):53–72, 1994. 36

[38] G. Canfora, M. Di Penta, and L. Cerulo. Achievements and Challenges in Software Reverse Engineering. *Communications of the ACM*, 54(4):142–151, 2011. 6, 15, 35, 78

[39] J. Canovas and J. Molina. An Architecture-driven Modernization Tool for Calculating Metrics. *IEEE Software*, 27(4):37–43, 2010. 46

[40] H. Chapman and P. A. Hall. *Software Reuse and Reverse Engineering in Practice*. Chapman and Hall, Ltd. London, UK, 1992. 35

[41] M. Chechik, S. Nejati, and M. Sabetzadeh. A Relationship-based Approach to Model Integration. *Innovations in Systems and Software Engineering*, 8(1):3–18, 2012. 95

[42] Checkstyle. Checkstyle development tool, 2018. URL: `http://checkstyle.sourceforge.net`. 62

[43] E. J. Chikofsky and J. H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990. 6, 14, 35

[44] A. Cicchetti, F. Ciccozzi, and T. Leveque. A Hybrid Approach for Multi-view Modeling. *Electronic Communications of the EASST*, 50, 2011. 88

[45] M. Clavreul, O. Barais, and J.-M. Jézéquel. Integrating Legacy Systems with MDE. In *ACM/IEEE 32nd International Conference on Software Engineering (ICSE 2010)*, volume 2, pages 69–78. IEEE, 2010. 38, 39

[46] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009. 70

[47] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006. 25

[48] G. Daniel, F. Jouault, G. Sunyé, and J. Cabot. Gremlin-ATL: a Scalable Model Transformation Framework. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, pages 462–472. IEEE, 2017. 139

[49] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot. NeoEMF: a Multi-database Model Persistence Framework for Very Large Models. *Science of Computer Programming*, 149:9–14, 2017. 113, 115, 117, 136

[50] G. Daniel, G. Sunyé, and J. Cabot. PrefetchML: a Framework for Prefetching and Caching Models. In *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*, pages 318–328. ACM, 2016. 136

[51] J. Davis. GME: The Generic Modeling Environment. In *18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, pages 82–83. ACM, 2003. 37

[52] C. Debreceni, Á. Horváth, Á. Hegedüs, Z. Ujhelyi, I. Ráth, and D. Varró. Query-driven Incremental Synchronization of View Models. In *2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO 2014)*, page 31. ACM, 2014. 94

[53] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible Architecture Ccnformance Assessment with ConQAT. In *ACM/IEEE 32nd International Conference on Software Engineering (ICSE 2010)*, volume 2, pages 247–250. IEEE, 2010. 38, 39

[54] S. Demeyer, S. Ducasse, and E. Tichelaar. Why FAMIX and not UML? UML Shortcomings for Coping with Round-trip Engineering. In *2nd International Conference on the Unified Modeling Language (UML'99)*, pages 28–30. Springer, 1999. 70

[55] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling Cyber–Physical Systems. *Proceedings of the IEEE*, 100(1):13–28, 2012. 28, 81

[56] R. Drath, A. Luder, J. Peschke, and L. Hundt. AutomationML - The Glue for Seamless Automation Engineering. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008)*, pages 616–623. IEEE, 2008. 25

[57] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO - Generic Understanding of Programs - An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2):47–56, 2002. 38, 39

[58] Eclipse Foundation. Acceleo, 2018. URL: https://www.eclipse.org/acceleo/. 31, 55, 57

[59] Eclipse Foundation. ATL Transformation Language (ATL), 2018. URL: https://www.eclipse.org/atl/. 31, 57, 118

[60] Eclipse Foundation. Connected Data Objects (CDO), 2018. URL: https://www.eclipse.org/cdo/. 31, 65, 113, 115, 117

[61] Eclipse Foundation. Eclipse and the Eclipse Foundation, 2018. URL: http://www.eclipse.org/org/. 30, 133

[62] Eclipse Foundation. Eclipse Java Development Tools (JDT), 2018. URL: https://www.eclipse.org/jdt/. 54

[63] Eclipse Foundation. Eclipse Modeling Framework (EMF), 2018. URL: https://www.eclipse.org/modeling/emf/. 32, 57

[64] Eclipse Foundation. Eclipse Modeling Project, 2018. URL: https://www.eclipse.org/modeling/. 31

[65] Eclipse Foundation. Eclipse Project and Platform, 2018. URL: https://www.eclipse.org/eclipse/. 31

[66] Eclipse Foundation. Eclipse Public License (EPL) - v2.0, 2018. URL: https://www.eclipse.org/legal/epl-v20.html. 30, 58, 130, 133

[67] Eclipse Foundation. EMF Facet, 2018. URL: https://www.eclipse.org/facet/. 51, 53, 79, 89

[68] Eclipse Foundation. EMFStore, 2018. URL: https://www.eclipse.org/emfstore/. 31

[69] Eclipse Foundation. Epsilon, 2018. URL: https://www.eclipse.org/epsilon/. 32, 139

[70] Eclipse Foundation. Help - Eclipse Platform, 2018. URL: https://www.help.eclipse.org. 57

[71] Eclipse Foundation. MoDisco, 2018. URL: https://www.eclipse.org/MoDisco/. 32, 50

[72] Eclipse Foundation. Papyrus, 2018. URL: https://www.eclipse.org/papyrus/. 32, 91, 140

[73] Eclipse Foundation. Polarsys, Open Source Solutions for Embedded Systems, 2018. URL: https://www.polarsys.org. 91

[74] Eclipse Foundation. Sirius, 2018. URL: https://www.eclipse.org/sirius/. 32, 93

[75] Eclipse Foundation. VIATRA Viewers, 2018. URL: https://www.eclipse.org/viatra/documentation/addons.html. 94

[76] Eclipse Foundation. Xtend, 2018. URL: https://www.eclipse.org/xtend/. 114

[77] Eclipse Foundation. Xtext, 2018. URL: https://www.eclipse.org/Xtext/. 32, 92, 114

[78] Eclipse Foundation - Polarsys. Kitalpha, 2018. URL: https://www.polarsys.org/projects/polarsys.kitalpha. 91

[79] E. Eilam. *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, 2011. 36

[80] K. El Emam and A. G. Koru. A Replicated Survey of IT Software Project Failures. *IEEE Software*, 25(5), 2008. 5, 13

[81] G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML Collaboration Diagrams and their Transformation to Java. In *2nd International Conference on the Unified Modeling Language (UML'99)*, pages 473–488. Springer, 1999. 70

[82] J. A. Estefan et al. Survey of model-based systems engineering (MBSE) methodologies. *Incose MBSE Focus Group*, 25(8):1–12, 2007. 23

[83] J.-M. Favre. Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. IBFI, 2005. 37

[84] L. Favre. *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution*. IGI Global - Premier Reference Source, 2010. 37

[85] S. Feldmann, M. Wimmer, K. Kernschmidt, and B. Vogel-Heuser. A Comprehensive Approach for Managing Inter-model Inconsistencies in Automated Production Systems Engineering. In *16th International Conference on Automation Science and Engineering (CASE 2016)*, pages 1120–1127. IEEE, 2016. 81

[86] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus - Reverse Engineering Tool and Schema for C++. In *International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE, 2002. 38, 39

[87] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering (SEKE)*, 2(1):31–57, 1992. 81, 83

[88] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *International Workshop on Theory and Application of Graph Transformations (TAGT'98)*, pages 296–309. Springer, 1998. 70

[89] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel. Model-driven Engineering for Software Migration in a Large Industrial Context. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, pages 482–497. Springer, 2007. 38, 39

[90] M. Fowler. *Domain-specific Languages*. Pearson Education, 2010. 27

[91] P. Fradet, D. Le Métayer, and M. Périn. Consistency Checking for Multiple View Software Architectures. In *Software Engineering - ESEC/FSE'99*, pages 410–428. Springer, 1999. 38, 40

[92] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE Computer Society, 2007. 27

[93] French public industrial association. Alliance Industrie du Futur, 2018. URL: http://www.industrie-dufutur.org. 28, 141

[94] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. AUTOSAR–A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, page 5, 2009. 25

[95] K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Melo, and J. M. Soto. White-box Modernization of Legacy Applications. In *6th International Conference on Model and Data Engineering (MEDI 2016)*, pages 274–287. Springer, 2016. 38, 41

[96] K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Viera, F. Melo, and J. M. Soto. White-box Modernization of Legacy Applications: the Oracle Forms Case Study. *Computer Standards & Interfaces*, 57:110–122, 2018. 38, 41, 137

[97] German public industrial association. Plattform Industry 4.0, 2018. URL: http://www.plattform-i40.de. 28, 141

[98] D. Gessenharter and M. Rauscher. Code Generation for UML2 Activity Diagrams. In *7th European Conference on Modelling Foundations and Applications (ECMFA 2011)*, pages 205–220. Springer, 2011. 70

[99] T. Girba. *The Moose Book*. Self Published, 2010. 37, 38, 40

[100] T. Goldschmidt, S. Becker, and E. Burger. Towards a Tool-Oriented Taxonomy of View-Based Modelling. In *Modellierung*, volume 201, pages 59–74, 2012. 83

[101] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, and C. Guychard. Addressing Modularity for Heterogeneous Multi-model Systems Using Model Federation. In *Companion of the 15th International Conference on Modularity (Modularity 2016)*, pages 206–211. ACM, 2016. 92

[102] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition (Java Series)*. Addison-Wesley Professional, 2014. 76

[103] R. C. Gronback. *Eclipse Modeling Project: a Domain-Specific Language (DSL) Toolkit*. Pearson Education, 2009. 31

[104] O. Gruber, B. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The Eclipse 3.0 Platform: Adopting OSGi Technology. *IBM Systems Journal*, 44(2):289–299, 2005. 31

[105] W. Harrison, C. Barton, and M. Raghavachari. Mapping UML Designs to Java. In *15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*. ACM, 2000. 70

[106] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the Gap between Modelling and Java. In *International Conference on Software Language Engineering (SLE 2009)*, pages 374–383. Springer, 2009. 38, 39

[107] J. Herrington. *Code Generation in Action*. Manning Publications Co., 2003. 6, 14, 25

[108] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. Feature-based Classification of Bidirectional Transformation Approaches. *Software & Systems Modeling*, 15(3):907–928, 2016. 26

[109] J. Hutchinson, J. Whittle, and M. Rouncefield. Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors that Lead to Success or Failure. *Science of Computer Programming*, 89:144–161, 2014. 122

[110] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In *ACM/IEEE 33rd International Conference on Software Engineering (ICSE 2011)*, pages 471–480. IEEE, 2011. 6, 14, 27

[111] IBM. Rational software architect (rsa), 2018. URL: https://www.ibm.com/developerworks/downloads/r/architect/. 38, 41

[112] International Organization for Standardization (ISO). Extended Backus-Naur Form (EBNF), ISO/IEC 14977:1996, 2018. URL: https://www.iso.org/standard/26153.html. 24

[113] ISO/IEC/IEEE. Standard 42010:2011, Systems and Software Engineering - Architecture Description, 2018. URL: https://www.iso.org/standard/50508.html. 83, 88

[114] J. Jakob and A. Schürr. View Creation of Meta Models by Using Modified Triple Graph Grammars. *Electronic Notes in Theoretical Computer Science*, 211:181–190, 2008. 93

[115] J.-M. Jézéquel, O. Barais, and F. Fleurey. Model Driven Language Engineering with Kermeta. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 201–221. Springer, 2009. 48

[116] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. 31, 48, 94, 114, 139

[117] F. Jouault and M. Tisi. Towards Incremental Execution of ATL Transformations. In *3rd International Conference on Theory and Practice of Model Transformations (ICMT 2010)*, pages 123–137. Springer, 2010. 96

[118] R. Kazman, S. G. Woods, and S. J. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *5th International Working Conference on Reverse Engineering (WCRE 1998)*, pages 154–163. IEEE, 1998. 38, 40

[119] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008. 25

[120] S. Kent. Model Driven Engineering. In *International Conference on Integrated Formal Methods (iFM 2002)*, pages 286–298. Springer, 2002. 23

[121] H. Kern, A. Hummel, and S. Kühne. Towards a Comparative Analysis of Metametamodels. In *Proceedings of the compilation of the co-located workshops on*

DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11, pages 7–12. ACM, 2011. 25

[122] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. How do Professionals Perceive Legacy Systems and Software Modernization? In *ACM/IEEE 36th International Conference on Software Engineering (ICSE 2014)*, pages 36–47. ACM, 2014. 5, 13

[123] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA Explained - the Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003. 28

[124] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf. A Study on the Current State of the Art in Tool-supported UML-based Static Reverse Engineering. In *9th International Working Conference on Reverse Engineering (WCRE 2002)*, pages 22–32. IEEE, 2002. 70

[125] D. S. Kolovos. Establishing Correspondences Between Models with the Epsilon Comparison Language. In *European Conference on Model Driven Architecture-Foundations and Applications (ECMFA 2009)*, pages 146–157. Springer, 2009. 114

[126] D. S. Kolovos, R. F. Paige, and F. A. Polack. Merging Models with the Epsilon Merging Language (EML). In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, pages 215–229. Springer, 2006. 90

[127] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, et al. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering (BigMDE'13), co-located with STAF conferences*, page 2. ACM, 2013. 122

[128] D. S. Kolovos, L. M. Rose, N. D. Matragkas, R. F. Paige, F. A. Polack, and K. J. Fernandes. Constructing and Navigating Non-invasive Model Decorations. In *3rd International Conference on Theory and Practice of Model Transformations (ICMT 2010)*, pages 138–152. Springer, 2010. 90

[129] E. Korshunova, M. Petkovic, M. Van Den Brand, and M. R. Mousavi. CPP2XMI: Reverse engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code. In *13th International Working Conference on Reverse Engineering (WCRE 2006)*, pages 297–298. IEEE, 2006. 70

[130] J. Koskinen, J. J. Ahonen, H. Sivula, T. Tilus, H. Lintinen, and I. Kankaanpaa. Software Modernization Decision Criteria: An Empirical Study. In *9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 324–331. IEEE, 2005. 5, 13

[131] T. B. La Fosse, M. Tisi, and J.-M. Mottu. Injecting Execution Traces into a Model-Driven Framework for Program Analysis. In *Federation of International Conferences on Software Technologies: Applications and Foundations (STAF 2017)*, pages 3–13. Springer, 2017. 137

[132] P. Langer, K. Wieland, M. Wimmer, and J. Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):1–29, 2012. 89

[133] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. CodeCrawler: an Information Visualization Tool for Program Comprehension. In *ACM/IEEE 27th International Conference on Software Engineering (ICSE 2005)*, pages 672–673. ACM, 2005. 38, 40

[134] E. Leblebici, A. Anjorin, and A. Schürr. Developing eMoflon with eMoflon. In *7th International Conference on the Theory and Practice of Model Transformations (ICMT 2014)*, pages 138–145. Springer, 2014. 93

[135] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. 5, 13

[136] P. F. Linington. RM-ODP: The Architecture. In *Open Distributed Processing*, pages 15–33. Springer, 1995. 81

[137] T. Mayerhofer, P. Langer, and G. Kappel. A Runtime Model for fUML. In *7th Workshop on Modelsrun.time, co-located with the ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012)*, pages 53–58. ACM, 2012. 69, 73

[138] E. Mayol and E. Teniente. A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In *Advances in Conceptual Modeling (ER'99): Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 62–73. Springer, 1999. 95

[139] E. Mayol and E. Teniente. A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In *18th International Conference on Conceptual Modeling (ER'99)*, pages 62–73. Springer, 1999. 139

[140] J. McAffer, J.-M. Lemieux, and C. Aniszczyk. *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2010. 31

[141] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Supporting Executable Mappings in Model Management. In *ACM SIGMOD International Conference on Management of Data*, pages 167–178. ACM, 2005. 95

[142] T. Mens and P. Van Gorp. A taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. 25

[143] A. Menychtas, K. Konstanteli, J. Alonso, L. Orue-Echevarria, J. Gorronogoitia, G. Kousiouris, C. Santzaridou, H. Bruneliere, B. Pellens, P. Stuer, et al. Software Modernization and Cloudification Using the ARTIST Migration Methodology and Framework. *Scalable Computing: Practice and Experience*, 15(2):131–152, 2014. 71, 79, 82, 130

[144] Mia-Software, Sodifrance Group. Mia-Quality, 2018. URL: `http://www.mia-software.com/produits/mia-quality/`. 62

[145] P. Mohagheghi and V. Dehlen. Where is the Proof? A Review of Experiences from Applying MDE in Industry. In *European Conference on Model Driven Architecture-Foundations and Applications (ECMFA 2008)*, pages 432–443. Springer, 2008. 27

[146] P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling modeling modeling. *Software & Systems Modeling*, 11(3):347–359, 2012. 24

[147] NaoMod Team. frex source code repository, 2018. URL: `https://github.com/atlanmod/fREX`. 74

[148] M. Nassar. VUML: a Viewpoint Oriented UML Extension. In *ACM/IEEE 18th International Conference on Automated Software Engineering (ASE 2003)*, pages 373–376. IEEE, 2003. 94

[149] M. L. Nelson. A Survey of Reverse Engineering and Program Comprehension. *arXiv preprint cs/0503068*, 2005. 35

[150] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *ACM/IEEE 22th International Conference on Software Engineering (ICSE 2000)*, pages 742–745. ACM, 2000. 70

[151] NoMagic/3DS. Magicdraw, 2018. URL: https://www.nomagic.com/products/magicdraw. 38, 41

[152] F. Noyrit, S. Gérard, and B. Selic. FacadeMetamodel: Masking UML. In *ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2012)*, pages 20–35. Springer, 2012. 91

[153] Object Management Group (OMG). Abstract Syntax Tree Metamodel (ASTM), 2018. URL: https://www.omg.org/spec/ASTM. 50

[154] Object Management Group (OMG). Architecture Driven Modernization (ADM), 2018. URL: http://adm.omg.org. 30, 37, 40, 50

[155] Object Management Group (OMG). Business Process Model and Notation (BPMN), 2018. URL: http://www.bpmn.org. 30

[156] Object Management Group (OMG). Knowledge Discovery Metamodel (KDM), 2018. URL: https://www.omg.org/spec/KDM. 50

[157] Object Management Group (OMG). Meta Object Facility (MOF), 2018. URL: http://www.omg.org/mof. 29

[158] Object Management Group (OMG). Model Driven Architecture (MDA), 2018. URL: http://www.omg.org/mda. 28, 94

[159] Object Management Group (OMG). MOF Model To Text Transformation Language (MOFM2T), 2018. URL: urlhttp://www.omg.org/spec/MOFM2T. 30

[160] Object Management Group (OMG). Object Constraint Language (OCL), 2018. URL: https://www.omg.org/spec/OCL. 29, 57, 118

[161] Object Management Group (OMG). Query/View/Transformation (QVT), 2018. URL: http://www.omg.org/spec/QVT. 30, 48, 92

[162] Object Management Group (OMG). Requirements Interchange Format (ReqIF), 2018. URL: https://www.omg.org/spec/ReqIF. 120

[163] Object Management Group (OMG). Semantics of a Foundational Subset for Executable UML Models (fUML), 2018. URL: https://www.omg.org/spec/FUML. 70, 71, 76

[164] Object Management Group (OMG). Structured Metrics Metamodel (SMM), 2018. URL: https://www.omg.org/spec/SMM. 50

[165] Object Management Group (OMG). Systems Modeling Language (SysML), 2018. URL: http://www.omgsysml.org. 30

[166] Object Management Group (OMG). Unified Architecture Framework (UAF), 2018. URL: https://www.omg.org/spec/UAF/. 30

[167] Object Management Group (OMG). Unified Modeling Language (UML), 2018. URL: www.uml.org. 7, 15, 30, 71, 120

[168] Object Management Group (OMG). XML Metadata Interchange (XMI), 2018. URL: http://www.omg.org/spec/XMI. 29, 66

[169] T. Olsson and J. Grundy. Supporting Traceability and Inconsistency Management between Software Artefacts. In *International Conference on Software Engineering and Applications (SEA 2002)*. IASTED, 2002. 38, 40

[170] Oracle. Java technology/language, 2018. URL: https://www.java.com. 25

[171] OSGI Alliance. Open Services Gateway initiative (OSGI), 2018. URL: https://www.osgi.org. 31

[172] M. J. Pacione, M. Roper, and M. Wood. A Novel Software Visualisation Model to Support Software Comprehension. In *11th International Working Conference on Reverse Engineering (WCRE 2004)*, pages 70–79. IEEE, 2004. 38, 40

[173] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46(9):1155–1179, 2016. 38, 39

[174] R. Pérez-Castillo, I. G.-R. De Guzman, and M. Piattini. Knowledge Discovery Metamodel-ISO/IEC 19506: A Standard to Modernize Legacy Systems. *Computer Standards & Interfaces*, 33(6):519–532, 2011. 70

[175] W. J. Premerlani and M. R. Blaha. An Approach for Reverse Engineering of Relational Databases. In *International Working Conference on Reverse Engineering (WCRE 1993)*, pages 151–160. IEEE, 1993. 36

[176] A. Prout, J. M. Atlee, N. A. Day, and P. Shaker. Semantically Configurable Code Generation. In *ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 20080)*, pages 705–720. Springer, 2008. 138

[177] Ó. S. Ramón, J. S. Cuadrado, and J. G. Molina. Model-driven Reverse Engineering of Legacy Graphical User Interfaces. *Automated Software Engineering*, 21(2):147–186, 2014. 38, 39, 137

[178] M. G. Rekoff. On Reverse Engineering. *IEEE Transaction on Systems, Man and Cybernetics*, 15(2):13–17, 1985. 35

[179] E. Roberts. An Overview of MiniJava. *ACM SIGCSE Bulletin*, 33(1):1–5, 2001. 74

[180] B. Roy and T. N. Graham. An Iterative Framework for Software Architecture Recovery: An Experience Report. In *European Conference on Software Architecture (ECSA 2008)*, pages 210–224. Springer, 2008. 38, 39

[181] S. Rugaber and K. Stirewalt. Model Driven Reverse Engineering. *IEEE Software*, 21(4):45–53, 2004. 6, 15, 37

[182] D. C. Schmidt. Model-driven Engineering. *IEEE Computer*, 39(2):25, 2006. 5, 13, 23

[183] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994)*, pages 151–163. Springer, 1994. 93

[184] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003. 27

[185] O. Semeráth, C. Debreceni, Á. Horváth, and D. Varró. Incremental Backward Change Propagation of View Models by Logic Solvers. In *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*, pages 306–316. ACM, 2016. 94

[186] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-driven Software Development. *IEEE Software*, 20(5):42–45, 2003. 25

[187] K. Smolander, K. Lyytinen, V.-P. Tahvanainen, and P. Marttiin. MetaEdit - A Flexible Graphical Environment for Methodology Modelling. In *International Conference on Advanced Information Systems Engineering (CAiSE 1991)*, pages 168–193. Springer, 1991. 37

[188] H. M. Sneed. Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture. In *International Conference on Software Maintenance (ICSM 1992)*, pages 105–116. IEEE, 1992. 38, 39

[189] H. M. Sneed. Migrating from COBOL to Java. In *International Conference on Software Maintenance (ICSM 2010)*, pages 1–7. IEEE, 2010. 38, 39

[190] Sodifrance Group. Mia-Software, 2018. URL: http://www.mia-software.com. 50, 79, 134

[191] Softeam. Modelio, 2018. URL: https://www.modelio.org. 38, 41

[192] SonarQube. Sonarqube tool, 2018. URL: https://www.sonarqube.org. 63

[193] Sparx Systems. Enterprise architect, 2018. URL: http://sparxsystems.com/products/ea/. 38, 41

[194] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008. 32

[195] W. Sun, S. Li, D. Zhang, and Y. Yan. A Model-driven Reverse Engineering Approach for Semantic Web Services Composition. In *WRI World Congress on Software Engineering (WCSE 2009)*, volume 3, pages 101–105. IEEE, 2009. 38, 39

[196] The Open Group. The TOGAF standard, 2018. URL: http://www.opengroup.org/subjectareas/enterprise/togaf. 131

[197] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *European Conference on Model Driven Architecture-Foundations and Applications (ECMFA 2009)*, pages 18–33. Springer, 2009. 26

[198] M. Tisi, S. Martinez, and H. Choura. Parallel Execution of ATL Transformation Rules. In *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, pages 656–672. Springer, 2013. 136

[199] J.-P. Tolvanen and M. Rossi. MetaEdit+: Defining and Using Domain-specific Modeling Languages and Code Generators. In *18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, pages 92–93. ACM, 2003. 37

[200] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An Integrated Development Environment for Live Model Queries. *Science of Computer Programming*, 98:80–99, 2015. 94

[201] W. M. Ulrich and P. Newcomb. *Information Systems Transformation: Architecture-driven Modernization Case Studies*. Morgan Kaufmann, 2010. 7, 15

[202] A. Van Deursen, E. Visser, and J. Warmer. Model-driven Software Evolution: A Research Agenda. In *1st international workshop on Model-driven Software Evolution (MoDSE)*, pages 41–49. University of Nantes, 2007. 37

[203] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework. *Software & Systems Modeling*, 15(3):609–629, 2016. 139

[204] D. Varró and A. Pataricza. VPM: A Visual, Precise and Multilevel Metamodeling Framework for Describing Mathematical Domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics). *Software & Systems Modeling*, 2(3):187–210, 2003. 24

[205] A. Vignaga, F. Jouault, M. C. Bastarrica, and H. Bruneliere. Typing Artifacts in Megamodeling. *Software & Systems Modeling*, 12(1):105–119, 2013. 18

[206] Visual Paradigm International. Visual paradigm, 2018. URL: https://www.visual-paradigm.com. 38, 41

[207] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2013. 23

[208] A. Von Mayrhauser and A. M. Vans. Program Comprehension during Software Maintenance and Evolution. *IEEE Computer*, 28(8):44–55, 1995. 5, 13, 39

[209] R. Wettel, M. Lanza, and R. Robbes. Software Systems as Cities: A Controlled Experiment. In *ACM/IEEE 33rd International Conference on Software Engineering (ICSE 2011)*, pages 551–560. ACM, 2011. 38, 40

[210] J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014. 6, 14, 27, 122

[211] J. Wiegand et al. Eclipse: A Platform for Integrating Development Tools. *IBM Systems Journal*, 43(2):371–383, 2004. 31

[212] E. D. Willink. Deterministic Lazy Mutable OCL Collections. In *Federation of International Conferences on Software Technologies: Applications and Foundations (STAF 2017)*, pages 340–355. Springer, 2017. 139

[213] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2016. 141

[214] A. T. Wood-Harper, L. Antill, and D. E. Avison. *Information Systems Definition: The Multiview Approach*. Blackwell Scientific Publications, Ltd., 1985. 83

[215] World Wide Web Consortium (W3C). Extensible Markup Language (XML), 2018. URL: https://www.w3.org/XML/. 24

[216] World Wide Web Consortium (W3C). Mathematical Markup Language (MathML), 2018. URL: https://www.w3.org/Math/. 25

[217] Z. Yang and M. Jiang. Using Eclipse as a Tool-integration Platform for Software Development. *IEEE Software*, 24(2), 2007. 56

[218] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Reverse Engineering Goal Models from Legacy Code. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 363–372. IEEE, 2005. 138

[219] J. A. Zachman. A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3):276–292, 1987. 81

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre : Approches Génériques Basées sur les Modèles pour la Rétro-Ingénierie et Compréhension du Logiciel

Mots clés : Ingénierie Dirigée par les Modèles, Rétro-Ingénierie, Compréhension, Fédération, Vues

Résumé : De nos jours, les entreprises font souvent face à des problèmes de gestion, maintenance, évolution ou remplacement de leurs systèmes logiciel existants. La Rétro-Ingénierie est la phase requise d'obtention de diverses représentations de ces systèmes pour une meilleure compréhension de leurs buts / états.
L'Ingénierie Dirigée par les Modèles (IDM) est un paradigme du Génie Logiciel reposant sur la création, manipulation et utilisation intensive de modèles dans les tâches de conception, développement, déploiement, intégration, maintenance et évolution. La Rétro-Ingénierie Dirigée par les Modèles (RIDM) a été proposée afin d'améliorer les approches de Rétro-Ingénierie traditionnelles. Elle vise à obtenir des modèles à partir d'un système existant, puis à les fédérer via des vues cohérentes pour une meilleure compréhension.
Cependant, les solutions existantes sont limitées car étant souvent des intégrations spécifiques d'outils. Elles peuvent aussi être (très) hétérogènes, entravant ainsi leurs déploiements. Il manque donc de solutions pour que la RIDM puisse être combinée avec des capacités de vue / fédération de modèles.
Dans cette thèse, nous proposons deux approches complémentaires, génériques et extensibles basées sur les modèles ainsi que leurs implémentations en open source basées sur Eclipse-EMF : (i) Pour faciliter l'élaboration de solutions de RIDM dans des contextes variés, en obtenant différents types de modèles à partir de systèmes existants (e.g. leurs codes source, données). (ii) Pour spécifier, construire et manipuler des vues fédérant différents modèles (e.g. résultant de la RIDM) selon des objectifs de compréhension (e.g. pour diverses parties prenantes).

Title: Generic Model-based Approaches for Software Reverse Engineering and Comprehension

Keywords: Model Driven Engineering, Reverse Engineering, Comprehension, Federation, Views

Abstract: Nowadays, companies face more and more the problem of managing, maintaining, evolving or replacing their existing software systems. Reverse Engineering is the required phase of obtaining various representations of these systems to provide a better comprehension of their purposes / states.
Model Driven Engineering (MDE) is a Software Engineering paradigm relying on intensive model creation, manipulation and use within design, development, deployment, integration, maintenance and evolution tasks. Model Driven Reverse Engineering (MDRE) has been proposed to enhance traditional Reverse Engineering approaches via the application of MDE. It aims at obtaining models from an existing system according to various aspects, and then possibly federating them via coherent views for further comprehension.
However, existing solutions are limited as they quite often rely on case-specific integrations of different tools. Moreover, they can sometimes be (very) heterogeneous which may hinder their practical deployments. Generic and extensible solutions are still missing for MDRE to be combined with model view / federation capabilities.
In this thesis, we propose to rely on two complementary, generic and extensible model-based approaches and their Eclipse/EMF-based implementations in open source: (i) To facilitate the elaboration of MDRE solutions in many different contexts, by obtaining different kinds of models from existing systems (e.g. their source code, data). (ii) To specify, build and manipulate views federating different models (e.g. resulting from MDRE) according to comprehension objectives (e.g. for different stakeholders).