Scalable Model Views over Heterogeneous Modeling Technologies and Resources

Hugo Bruneliere, Florent Marchand de Kerchove, Gwendal Daniel, Sina Madani, Dimitris Kolovos, Jordi Cabot

Software and Systems Modeling (SoSyM) manuscript No.
(will be inserted by the editor)

Scalable Model Views over Heterogeneous Modeling Technologies and Resources

Hugo Bruneliere · Florent Marchand de Kerchove · Gwendal Daniel · Sina Madani · Dimitris Kolovos · Jordi Cabot

Received: date / Accepted: date

Abstract When engineering complex systems, models are typically used to represent various systems aspects. These models are often heterogeneous in terms of modeling languages, provenance, number or scale. As a result, the information actually relevant to engineers is usually split into different kinds of interrelated models. To be useful in practice, these models need to be properly integrated to provide global views over the system. This has to be made possible even when those models are serialized or stored in different formats adapted to their respective nature and scalability needs. Model view approaches have been proposed to tackle this issue. They provide unification mechanisms to combine and query various different models in a transparent way. These views usually target specific engineering tasks such as system design, monitoring, evolution, etc. In an industrial context, there can be many large-scale use cases where model views can be beneficial, in order to trace runtime and design-time data for example. However, existing model view solutions are generally designed to work on top of one single modeling technology (even though

This work has received funding from the ECSEL Joint Undertaking under grant agreement No. 737494 (MegaM@Rt2 project). This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation program and from Sweden, France, Spain, Italy, Finland & Czech Republic.

Hugo Bruneliere, Florent Marchand de Kerchove
IMT Atlantique, LS2N (CNRS) & ARMINES
Nantes, France
E-mail: hugo.bruneliere@imt-atlantique.fr, florent.marchand-de-kerchove@imt-atlantique.fr

Gwendal Daniel, Jordi Cabot
ICREA & Universitat Oberta de Catalunya (UOC)
Barcelona, Spain
E-mail: gdaniel@uoc.edu, jordi.cabot@icrea.cat

Sina Madani, Dimitris Kolovos
University of York
York, United Kingdom
E-mail: sm1748@york.ac.uk, dimitris.kolovos@york.ac.uk

model import/export capabilities are sometimes provided). Moreover, they mostly rely on in-memory constructs and low-level modeling APIs that have not been designed to scale in the context of large models stored in different kinds of data sources. This paper presents a general solution to efficiently support scalable model views over heterogeneous modeling resources possibly handled via different modeling technologies. To this intent, it describes our integration approach between a model view framework and various modeling technologies providing access to multiple types of modeling resources (e.g. in XML/XMI, CSV, databases). It also presents how queries on such model views can be executed efficiently by benefiting from the optimization of the different model technologies and underlying persistence backends. Our solution has been evaluated on a practical large-scale use case provided by the industry-driven European MegaM@Rt2 project, that aims at implementing a runtime $\leftrightarrow$ design time feedback loop. The corresponding EMF-based tooling support, modeling artifacts and reproducible benchmarks are all available online.

Keywords Modeling · Views · Heterogeneity · Scalability · Persistence · Database · Design Time · Runtime

1 Introduction

Different types of models are required when engineering complex systems. This is notably the case for systems of systems or Cyber-Physical Systems (CPSs) [20], where models have to represent both software and hardware aspects. Depending on the concerned application domains, the nature and number of the involved models can vary significantly: they can be captured using various modeling languages, come from different sources or have a considerably large size. Moreover, such models usually contain complementary information that is not distributed uniformly across them. Thus, engineers need to combine different models in order to have a better vision and understanding of the whole system. The combined models are intended to support particular engineering activities such as system design, development, monitoring or adaptation/evolution.

Several model view approaches have already been proposed in order to provide such support [7]. Relying on model-based principles and techniques, they allow specifying, creating and handling views on models that possibly conform to different metamodels. Once built, the model views can be used to uniformly manipulate and query the aggregated data coming from the various contributing models. However, most of the model view approaches have only been deployed on models of small to medium size (e.g. manually created design models) and not on large to huge models (e.g. automatically generated runtime models). Indeed, scalability is known to be one of the main issues hampering the full adoption of MDE in industry [26]. In parallel, the support for strongly interconnected models whose content is coming from different sources appears to be the focus of only a few of the proposed approaches. Those two aspects

are particularly relevant in real industrial contexts, such as in MegaM@Rt2 for example.

The collaborative MegaM@Rt2 project[1] is a recent and large European initiative supported by both industry and academic partners. As part of its continuous system engineering approach [1], the project notably aims at providing a runtime $\leftrightarrow$ design time feedback loop that could be deployed and used in different industrial domains. Such a feedback loop can bring interesting benefits when used in the context of the above-mentioned engineering activities, for instance. To realize this feedback loop in practice, model views can be used to transparently combine all the required (design and runtime) models.

In this paper, we present the current state of our work on supporting the creation of scalable model views over heterogeneous modeling technologies providing access to multiple types of modeling resources (i.e. models whose data is coming from different sources). This paper is an extended version of a previous publication at the 21^{st} ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2018) [11]. The present paper focuses not only on scalability (as in the past work published at MODELS 2018), but it also introduces the important and complementary dimension of heterogeneity. This allowed us to combine various modeling technologies (as well as related modeling resources) inside a same model view at reasonable / acceptable performance levels.

As a consequence, this paper naturally covers the main contributions from the first MODELS 2018 paper:

1. A conceptual integration approach for the joint use of a model view framework with various model persistence capabilities, thus providing access to different kinds of modeling resources.
2. A scalable instantiation of this conceptual approach combining the EMF Views [12] model view solution with NeoEMF [17] and CDO [21] modeling resources handled via the EMF [44] modeling technology.
3. An evaluation of our conceptual approach and its practical instantiation on a large-scale and realistic use case from the MegaM@Rt2 project.

In addition, for this extended version, we significantly reworked and augmented the previous content in order to present the following new contributions:

1. A generalization of the initial conceptual integration approach that now supports various modeling technologies, each of them providing access to different modeling resources and their coexistence in a same model view.
2. A generic and more efficient model view query execution dispatching strategy that delegates partial query computations to the underlying backends of the respective modeling resources whenever appropriate.
3. An instantiation of this upgraded approach and definition of optimization strategies that refine EMF Views and NeoEMF in order to integrate the use of Epsilon [41] as another modeling technology complementary to EMF, thus providing access to alternative types of modeling resources.

[1] `http://megamart2-ecsel.eu/`

4. An evaluation of this upgraded instantiation on a modified version of our MegaM@Rt2 use case that now uses additional and different types of modeling resources.

We kept up to date the description of these different items (i.e. both the 3 initial contributions from our MODELS 2018 paper and the 4 new contributions in this SoSyM paper) in order to reflect our latest advancements and achievements. Finally, we extended the paper with more related work, lessons learned from our research effort and some further conclusions from performed experiments.

The rest of this paper is organized as follows. Section 2 introduces the required background in terms of model view and model persistence solutions, in order to be able to comprehend the contributions presented afterwards. Section 3 motivates our work by describing its main objectives and illustrating them with a practical use case from the industrial MegaM@Rt2 project. As a solution, Section 4 presents our conceptual approach to integrate a model view framework with multiple modeling technologies and model persistence solutions. Then, Section 5 details how we implemented this conceptual approach in practice by using modeling solutions based on the Eclipse Modeling Framework (EMF) and Eclipse Epsilon. Section 6 provides a detailed evaluation of our approach and implementation via a set of benchmarks performed in the context of our MegaM@Rt2 use case. Section 7 reviews related work. Finally, Section 8 summarizes main general lessons we have learned from the realized work and Section 9 concludes by proposing directions for future work.

2 Background

In this section, we introduce the main concepts related to model view and model persistence solutions. Indeed, these are the kinds of solutions which are central to the contributions proposed in this paper. The terminology described in what follows is then used and referred to in the remainder of the paper.

2.1 Model Views

As explained in detail in an existing survey of model view approaches [7], the terms *view* and *viewpoint* have been used in several different ways depending on the contexts and concerned model view solutions. However, the overall terminology appears to start converging since the specification of the ISO standard 42010 [27].

A *view* is usually considered as a special kind of model. It contains information that is related to and coming from other models, which can also be themselves views (i.e. views can be defined over views). The relation between the view and these other models can be specified by various means such as (model) transformations, rules, queries, etc. A view is always a view on something: 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. Thus, as any model, a view conforms to a metamodel which describes a particular *viewpoint*. This viewpoint can be defined a-priori, or can be sometimes deduced from the specification of the view itself.

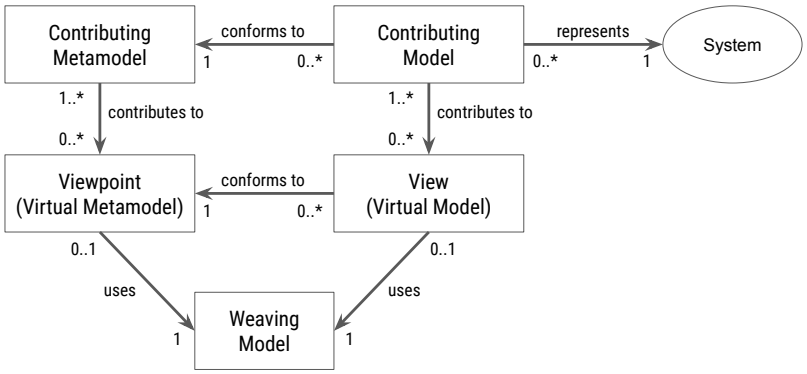

Fig. 1 A terminology for model views.

As shown in Figure 1, we consider the following general definitions:

- A **system** is a unit consisting of multiple largely interdependent entities which are designed and implemented by engineers. A system can encompass software, hardware, business components, as well as any 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. It consists of concepts coming from one or more base metamodels, possibly complemented with some new interconnections between them and newly added features. In our present context, it is realized as a virtual metamodel that describes the types of elements that can appear in corresponding views.
- A **view** is a representation of a specific system from the perspective of a given viewpoint. It is an instance of a particular viewpoint and consists of a set of elements coming from one or more base models. It is possibly complemented with some new interconnections between them as well as additional data, manually entered and/or computed automatically (e.g. thanks to model transformations). In our present context, it is realized as a virtual model that describes the actual content of the view.
- A **base or contributing metamodel** is a metamodel that contributes to a given viewpoint. Depending on the approach, a viewpoint specification can possibly have one or several different base metamodels.
- A **base or contributing model** is a model that contributes to a given view. Depending on the approach and on the corresponding defined view-

point, a view can possibly gather elements coming from one or more base models.
- A **virtual metamodel** is a metamodel whose (virtual) meta-elements are just proxies to actual meta-elements contained in other metamodels. Thus, it does not duplicate any content from the base/contributing meta-models and just describes the additional metadata (e.g. via a related weaving model).
- A **virtual model** is a model whose (virtual) elements are just proxies to actual elements contained in other models. Thus, it does not duplicate any content from the base/contributing models and just describes the additional data (e.g. via a related weaving model). Moreover, it conforms to a virtual metamodel.
- A **weaving model** is a model that describes links between elements coming from other different models. It conforms to a weaving metamodel that specifies the types of links that can be represented at weaving model-level.

2.2 Model Persistence

Model persistence is one of the cornerstones in MDE processes: input models are loaded in memory from an existing source, navigated/updated (e.g. using model queries and transformations) and stored into particular formats. Such formats are meant to be supported by client applications, or shared between modelers. In the following subsections, we introduce the two main approaches proposed to store and to access models.

2.2.1 File-based Serialization

Since the publication of the XMI standard [39], file-based XML serialization has been the most popular format for storing and sharing models and metamodels. Indeed, modeling frameworks were originally designed to handle human-produced models whose size does not cause significant performance concerns. However, MDE practices in the industry [49] as well as the development of generative frameworks such as MoDisco [8] have created the need for handling large and complex (potentially generated) models.

This has emphasized the limitations of XMI, as XML-based serialization presents two drawbacks: (i) it sacrifices compactness in favor of human-readability and (ii) XML files need to be completely parsed and loaded in memory to obtain navigable models. The former reduces the efficiency of I/O accesses while the latter increases the memory required to load and query models, and also limits the use of proxies and partial loading [18]. Moreover, XMI persistence layers lack advanced features related to transactions or collaboration: large monolithic model files are challenging to integrate into existing versioning systems [4].

JSON [22], a lightweight text-based data-interchange format, has been proposed as an open and standard alternative to XMI / XML. For example,

this format is notably being used in the context of several modern NoSQL databases and related model persistence solutions (cf. Subsection 2.2.2).

2.2.2 Database Serialization

To tackle the issues of file-based serialization, several persistence solutions based on databases have been proposed [21,40]. They typically provide an intermediate mechanism to serialize in-memory models into an in-database representation by describing a model mapping, allowing to save and to access model elements.

Historically, RDBMS have been the preferred solution to store large models [21,45]. Some approaches derive a relational schema from an existing metamodel, e.g. creating tables to store the instances of each class of a metamodel and columns for every class attribute. This schema is then used to store model elements, to access attributes or to navigate associations using low-level query languages such as SQL. Existing frameworks implement the *de-facto* standard EMF API, and can be transparently integrated (once configured) into existing modeling applications to enhance their scalability.

While these solutions have proven their efficiency with respect to XMI-based implementations, the highly interconnected nature of models often requires multiple table join operations to compute complex model queries. This usually limits performance in terms of both execution time and memory consumption [5]. Moreover, the strict schema system used in RDBMS makes them hard to align with metamodel updates defining new types, associations, etc.

As a result, a new generation of model persistence solutions [17,40] have been proposed to benefit from the advancements of the NoSQL movement (that relies on the JSON format to store data). They come with task-specific databases overcoming RDBMS limitations in specific data processing contexts (e.g. when querying highly interconnected data). The proposed approaches are based on the schema-less nature of NoSQL data-stores to efficiently handle metamodel modifications. They also rely on the database query performance to compute complex model navigation more efficiently. As previous solutions, NoSQL-based persistence frameworks rely on an internal mapping to represent models using the database primitives (e.g. vertices and edges for graph databases), and provide a *lazy-loading* mechanism reducing memory consumption (e.g. by loading only accessed objects).

3 Motivation

In this section, we present the main objectives of our work aiming at providing a scalable model view support over possibly heterogeneous modeling technologies and resources (cf. Subsection 3.1). We also describe our motivating industrial use case coming from the European collaborative MegaM@Rt2 project (cf. Subsection 3.2).

3.1 Main Objectives

The need for combining different kinds of models (and corresponding resources) in a scalable way has been observed in several industrial contexts [49, 26]. Moreover, it is also recognized as a long-term challenge from a research perspective [31]. However, already existing model view solutions do not handle large and very large heterogeneous models in a very satisfying way, if at all (cf. Section 7). This is even more noticeable in the case of model views over (large) contributing models coming from various and varied data sources, and possibly handled via different modeling technologies. Thus, scalability and heterogeneity are two important and complementary challenges to be addressed in such a model view context.

Moreover, from a practical perspective, one of the the main benefits of using model views in an engineering context is to collect in a transparent way information that is spread across different models. Without such model views, engineers have to navigate and query the different models one by one, and then have to aggregate the obtained results. Notably, this includes recreating in the model views (manually or programmatically) the mappings between related elements originating from different contributing models. Instead, using model views, navigation and queries traversing several contributing models can be expressed and performed transparently as if dealing with a single model.

In this paper, we propose an integration approach for building scalable model views over heterogeneous modeling resources, possibly relying on different modeling technologies. Depending on their experience or contexts, engineers can use different modeling technologies and resources. Moreover, we do not want to impose any particular technology or modify any existing resource for our solution to be used. Thus, we aim at generically supporting models expressed using (i) various languages (i.e. metamodels) and (ii) various modeling technologies allowing to persist and query them. To this intent, we can notably benefit from different model persistence solutions that are particularly adapted to address scalability-related issues. Because we cannot reasonably cover all existing modeling technologies and resources, we demonstrate the applicability and relevance of our approach on some commonly used ones.

The four main general objectives we want to achieve with our integration approach are:

1. Refine the model view framework in order to support the integration of multiple (types of) modeling resources via different modeling technologies.
2. Persist the corresponding model view-specific information in a scalable way.
3. Load views and access corresponding model view elements with an acceptably low overhead.
4. Query model views efficiently, by leveraging both modeling technology and model persistence optimization.

In Section 4, we describe a generic conceptual approach to achieve these four objectives. In Section 5, we then explain how this approach has been implemented to integrate two existing modeling technologies: EMF and Epsilon. In

Section 6, we evaluate this approach and implementation in practice on the MegaM@Rt2 use case introduced in the next subsection.

At the time of writing, we can reasonably claim that we have successfully tackled items (1), (2) and (3). We have also been able to obtain significant results in terms of performance concerning item (4).

3.2 MegaM@Rt2 Use Case

MegaM@Rt2[2] is a large, industry-driven, European collaborative project with a consortium of 26 partners from 6 different national clusters. It aims at developing a scalable continuous system engineering and validation approach that can be practically deployed in various industrial domains [1]. The project tackles 9 case studies from a variety of potential application areas: aeronautics, railway, warehouse, telecommunication, etc. Among the proposed model-based MegaM@Rt2 methods and tools, a main contribution is notably a runtime ↔ design time feedback loop (re)usable in these different contexts [9].

From our study of the industrial MegaM@Rt2 requirements and case studies, we materialized such a feedback loop as a view gathering 4 different models covering both runtime and design time aspects of a given system. To realize this in practice, there is the need for a solution supporting the creation and handling of scalable and heterogeneous model views.

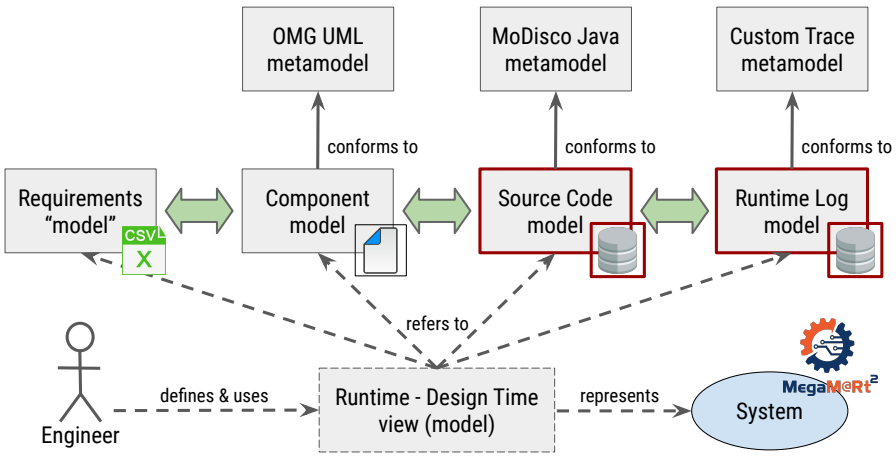

Fig. 2 Running use case from the industrial MegaM@Rt2 project.

As shown in Figure 2, the considered view combines a Runtime Log model (that conforms to a simple trace metamodel), a Source Code model (that conforms to the Java metamodel from MoDisco [8]), a Component model (that

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

conforms to OMG UML [38]) and a Requirements model (specified in a CSV file, that can be opened in a spreadsheet such as Excel).

Note that the Requirements model that initially conformed to OMG ReqIF [37], in the running example from our MODELS 2018 paper [11], has now been replaced by a CSV file that contains the same set of requirements[3]. This helps to illustrate the capability of our approach to integrate elements stored in non-modeling technologies (but widely used in practice) as part of our model views. However, for the sake of simplicity, we will still name this CSV file "Requirements model" in the remainder of the paper.

On the one hand, the Runtime Log model and (to a lesser extent) the Java Source Code model can be considered as *runtime models*. Depending on the size of the system under study, they can be extremely large (e.g. up to millions of model elements). This is especially the case for the Runtime Log model which represents actual system execution traces possibly covering a significant period of time. Thus, a typical solution to store and access them in a scalable way is to rely on database model persistence capabilities. The used technical solution then depends on the nature of the models, the types of expected accesses or manipulations (e.g. types of queries) or the way they have been (semi-)automatically obtained.

On the other hand, the Component model and Requirements model can be considered as *design models*. They are generally of a reasonable size compared to the runtime models, and are usually specified manually. Hence, they can be handled by modeling technologies relying on in-memory constructs and/or various kinds of file formats. For example, requirements can be specified by the system users or decision-makers using a familiar management tool or a spreadsheet that can produce a CSV export (e.g. Excel). A Component model is usually specified by architects or engineers via a dedicated CASE tool (e.g. producing a XMI/XML file).

A concrete example of the view from Figure 2 is given in Figure 3. By using 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 (in this case, a *trace.Log* element), one can move back to the originating Java code instructions (in this case, a *java.ClassDeclaration* element). One can then follow the links to the related components the code actually implements (in this case, a *uml.Component* element), finally up to the actual requirements these components fulfil (in this case, *csv.Row* elements).

Such a view combining different models can also be queried as any regular model, in order to compute/extract relevant data from 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*

[3] The Excel/CSV file stores the same requirement information than the ReqIF model from our initial MODELS 2018 paper but in a different format. For each row/requirement in the CSV file, the various columns contain the values of a requirement's properties (an identifier and a textual description for each requirement in our current example).

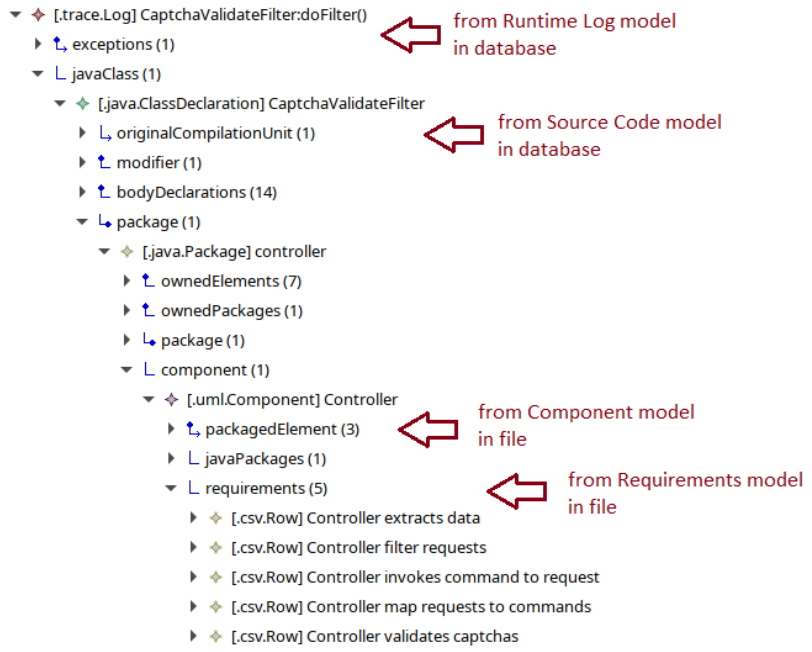

Fig. 3 Concrete example of a view in MegaM@Rt2 (based on the use case from Figure 2).

time to runtime traceability). In any case, our approach is flexible and can be adapted to the particular characteristics of a given application scenario (e.g. in terms of models and queries), according to the needs of industrial partners.

4 A Conceptual Approach for Integration

Subsection 4.1 describes our general integration approach to support scalable model views over heterogeneous modeling technologies and resources. In this context, Subsections 4.2 to 4.5 propose conceptual solutions to each one of our four identified objectives (cf. Subsection 3.1).

4.1 Overview of the Approach

Figure 4 graphically represents our integration approach. Subsection 4.1.1 provides more insights on this approach by explaining its main underlying concepts. Then, Subsection 4.1.2 describes its associated process and related steps.

4.1.1 Concepts

Any given **Modeling Technology** is usually composed of two main parts: a **Core** component providing the inner behavior (i.e. the base model manipu-

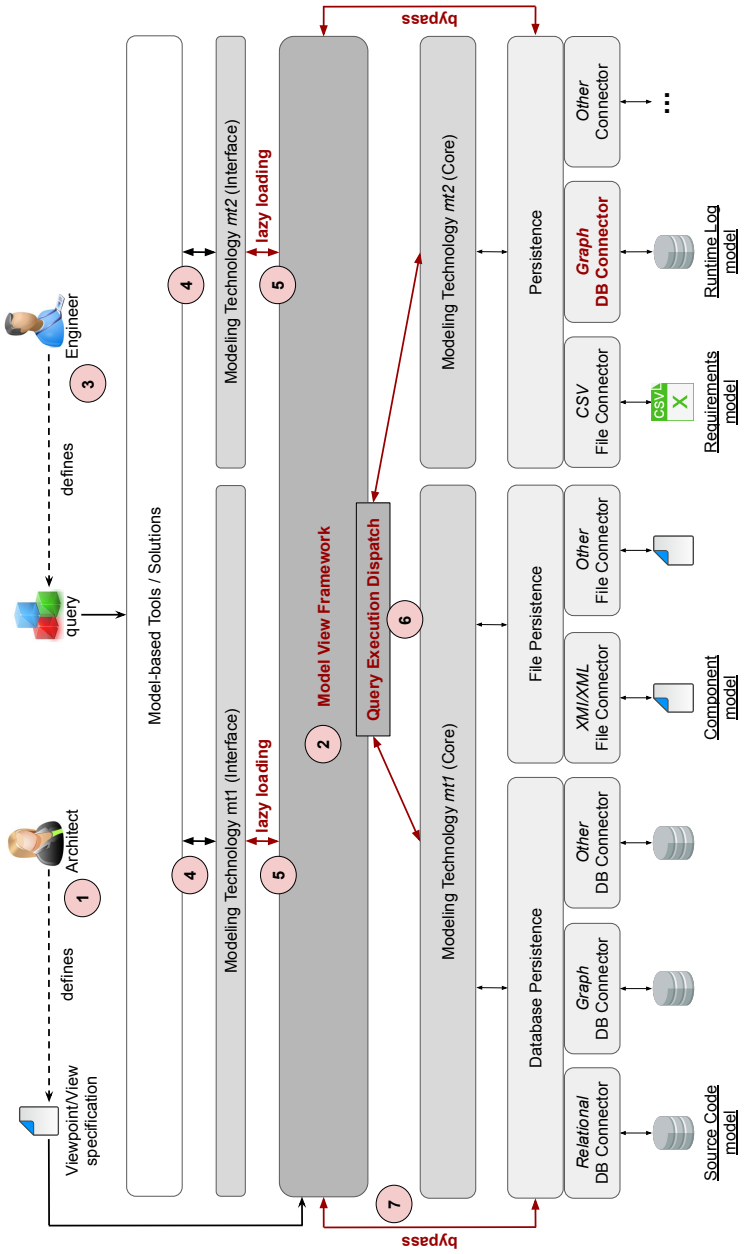

Fig. 4 A conceptual approach for integrating model view and model persistence capabilities via different modeling technologies.

lation facilities) and an **Interface** providing the API externally available for (re)use by different **Model-based Tools / Solutions**.

A **Modeling Technology** also generally comes with embedded support for one or several types of modeling resources. To this end, corresponding **Persistence** layers can be either explicitly defined and/or deeply integrated within the modeling technology.

It is common to have base **File Persistence** capabilities relying on the local file system, providing some file import/export capabilities using different serialization formats. This default mechanism can be used to store small to medium-sized models, such as the Requirements and Component design models from our use case (cf. Subsection 3.2).

In addition, **Database Persistence** capabilities can be proposed to connect a modeling technology to databases of various kinds (relational, graph-based, object-oriented etc.). These solutions are typically used to store large models, such as the Java Source Code and Runtime Log models from our use case, while still having an acceptable memory footprint.

Interestingly, the use of different **Modeling Technologies** can be combined within the same **Model-based Tool / Solution** in order to benefit from the various persistence capabilities and related optimization they may provide (e.g. more advanced lazy loading or querying features). This is the global integration principle we applied in the work presented in this paper.

In the general case, the **Model View Framework** must be integrated with the supporting **Modeling Technologies** (for instance by defining an appropriate connector) and comply with their respective **Interfaces**. This allows client applications of the same **Modeling Technologies** to query the corresponding views transparently as regular models. Moreover, for model views to scale with large models, the **Model View Framework** has to leverage the characteristics of the supported **Persistence** capabilities. This requires setting up a **Query Execution Dispatch** mechanism to benefit from the optimization provided by the different **Modeling Technologies**.

4.1.2 Process

The approach we propose also comes with a supporting process. In what follows we describe the different steps, still considering our use case from Subsection 3.2 (note that the step numbers correspond to those in Figure 4):

1. **Defining the viewpoint/view specification.** System architects specify how the view should be assembled from the various contributing models, e.g. by using dedicated languages such as VPDL [12] or MEL [10]. To this intent, they generally write a *viewpoint specification* describing how the different contributing metamodels are related to each other and which concepts these metamodels are supposed to provide to express corresponding views. In our use case, the OMG UML, MoDisco Java and Custom Trace metamodels are concerned[4]. Then, from this viewpoint specification, they

[4] Note that the CSV file has no explicit semantics but a column layout that provides an implicit (pseudo-)semantic model. In such a case, in addition to our knowledge of this format, we can rely on the model representation provided by the *Modeling Technology* handling this particular resource.

write a *view specification* describing a particular combination of models (that conform to the contributing metamodels) to build an actual view.

2. **Initializing the model view.** The viewpoint/view specification is provided as input to the *Model View Framework* in order to initialize the actual view. In our use case, a Requirements model (from an Excel worksheet serialized as a CSV file), a UML Component model (serialized in XMI), a Java Source Code model (stored in a relational database) and a Runtime Log model (stored in a graph database) are concerned. This results in a runtime to design time view, materialized as a single virtual model, that is now accessible as a regular model by *Model-based Tools / Solutions*. More details on how such a model view (and corresponding viewpoint) is produced by the *Model View Framework* are provided in the following subsections.
3. **Defining a query on the model view.** Once the view has been initialized, engineers can try to collect relevant information from it via different queries. For example, in our use case, they may want to trace a log entry (originally from the Runtime Log model) back to the corresponding system requirement (in the Excel worksheet serialized as a CSV file), passing by corresponding source code elements (in the Java model) and design components (in the UML model). Such a query can be expressed with any model manipulation or query language that is supported by the *Model-based Tools / Solutions* using the view.
4. **Running the query on the model view.** When engineers actually execute the query on the view, the concerned *Model-based Tools / Solutions* automatically transfer this query to the *Model View Framework* via the standard interfaces of the *Modeling Technology* these tools / solutions rely on. From the engineer's perspective, this is realized transparently as they interface only with their *Model-based Tools / Solutions*. It is then at this step that the *Model View Framework* actually starts loading the previously initialized view.
5. **Lazy loading view/model elements on-demand.** In a scalable approach, the model view should not be completely loaded at once. Thus, the proposed approach heavily relies on lazy loading techniques. This way, model view content from the contributing models are not loaded in memory until the concerned *Modeling Technology* actually requests them explicitly. These loading requests come from *Model-based Tools / Solutions* in charge of executing the query. For example, in our use case, Log entries from the Runtime Log model will only be loaded into the view when accessed in the query.
6. **Dispatching query execution.** Complementary to the lazy loading of view/model elements, a query execution dispatching mechanism is also used in order to optimize the query execution. This mechanism analyzes the query and identifies which contributing models are relevant to different query subsets. Thus, the *Model View Framework* automatically delegates its execution to the appropriate *Modeling Technology* in order to run a

given query subset. Section 5 provides technical insights on how this has been realized in practice.

7. **Bypassing the modeling technology (when more efficient).** In parallel to lazy loading and query execution dispatch, a bypassing mechanism is also used in order to improve the overall scalability of the solution. The *Model View Framework* can be informed that specific operations of a query (e.g. *allInstances()* in OCL) have already been optimized in the context of particular persistence solutions (e.g. using graph databases). In such cases, the *Model View Framework* is able to automatically interact with the persistence solution without having to access the *Modeling Technology*.

In what follows, we put the focus on how the proposed conceptual approach and associated process intended to tackle our four main objectives. Subsection 4.2 concerns both steps (1) and (2) of our process, while Subsections 4.3 and 4.4 relate to step (2) in particular. Subsection 4.5 addresses the more elaborate process from steps (3) to (7).

4.2 Building Model Views on Heterogeneous Modeling Sources

This activity is a prerequisite to the three subsequent ones. It basically covers the steps (1) and (2) of our overall process (cf. Subsection 4.1.2).

As introduced earlier, models can be possibly handled by different modeling technologies in the context of a global solution. Most (if not all) of these modeling technologies provide a default file persistence support, usually relying on XML-based format(s). However, they quite often support only a fixed number of types of data sources (or they require to implement specific extensions in order to support others). This can be a serious limitation when needing to load/store large-scale models from/into different kinds of databases (e.g. relational, graph-based etc.) or to rely on user-friendly non-model-based formats (e.g. spreadsheet). In some cases, such data sources already have a purpose of their own, e.g. they are used by external applications to accomplish certain engineering activities. In other cases, they can be created solely for the purpose of being used by a given model-based solution. Anyway, a model view framework should be able to rely on these (different kinds of) data sources by combining the support already provided by different modeling technologies.

In practice, model view approaches generally reuse the model persistence support provided by the main modeling technology they rely on. They are normally built upon one single modeling technology and do not provide out-of-the-box support for other ones. Moreover, they often lack native support for scalable model persistence solutions, i.e. optimized solutions for different kinds of modeling sources (e.g. databases). As a consequence, the model view framework needs to be properly integrated with such modeling technologies and the related persistence support. This way, depending on the nature of the contributing models, different persistence data backends (possibly associated with different modeling technologies) can be selected and combined within the context of the same view. This is notably the case in our use case, for example.

Such an integration can be performed in different ways. In some cases, it can be realized indirectly. The concerned modeling technology (there can be several connected to the model view framework) may be first refined to use the requested persistence solution. Then, the model view framework can rely on the general interface of this modeling technology to transparently access the underlying data resources. In some situations, a direct connection between the model view framework and the persistence solution may be desirable (via their respective APIs). For instance, this can allow for specific optimizations which could not be realized if (parts of) queries are systematically performed through the associated modeling technology. Subsection 5.1 provides practical examples of these in our technical context.

4.3 Persisting the Model View Information in a Scalable Way

This activity mostly relates to step (2) of our overall process (cf. Subsection 4.1.2).

Depending on the model view specification and the language used to define it, additional data can also be required to fully compute the model view. For instance, this is the case when a given model view provides new relationships between elements coming from different contributing models (as in our use case), or when it adds new properties to elements from one of the contributing models.

When initializing such a model view, this view-specific information has to be obtained in some way. One possibility is to compute it dynamically at runtime when loading the view. For example, this can be based on the data already available from the contributing models and complemented with the results of some predefined queries executed on top of these models. Another option is to collect it from an existing data source (in any format) or from an additional model created to this particular intent (i.e. not one of the contributing models). Such a model can collect manual inputs from the view users, e.g. retrieved via dedicated user interfaces. It can also be the result of executing an external application or model transformation, for example. In any case, the view mechanism has to be able to gather the appropriate information and reuse it in order to build the model view.

Related scalability problems can appear when the data specific to the model view are too large to be handled correctly by the persistence support provided by the used modeling technology. Indeed, depending on the nature of the model view, this additional data can be even larger than some of the contributing models themselves (e.g. as in our use case). In these cases, it is required to also persist the model storing these view-specific data by using a scalable persistence solution (e.g. a database). By relying on mechanisms embedded in such a solution (e.g. lazy loading), adopting this strategy can significantly reduce the memory footprint of given model views. Thus, this enables the manipulation of model views that could not even be loaded otherwise. Subsection 5.2 describes how we have realized this in practice in our technical context.

4.4 Optimizing the Model View Loading and Model Element Access

Similarly to the activity in Subsection 4.3), this activity mostly relates to step (2) of our overall process (cf. Subsection 4.1.2).

In the context of very large views, some operations can rapidly become expensive in terms of execution time and memory consumption. In extreme situations, 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 failing to load due to resource depletion while in use (e.g. by an actual user or by an application/tool querying it). The situation is notably critical during the process of initializing and/or loading the model view. Indeed, this particular phase usually requires a significant number of model element accesses, possibly to different model persistence solutions via several modeling technologies.

Substantial performance gains can be obtained by applying various lazy loading techniques at different levels in the resulting solution. In the general case, any hit to an actual model element has to be delayed as much as possible and must only occur when strictly needed. In terms of performance, the most efficient pieces of code are the ones that are never executed. Such optimization also concerns accesses to the various contributing models themselves (cf. Subsection 4.2) as well as to the view-specific elements (cf. Subsection 4.3). In all cases, this should not impact the overall usability and correctness of the handled model views.

Moreover, depending on the modeling technologies(s) used and related persistence solution(s), the model view framework can be refined differently. For given model element accesses, the model view framework can directly benefit from specific capabilities provided by certain kinds of data sources (e.g. graph databases). For instance, the view framework can leverage on a database-specific API to turn full traversals of models into more selective (and so more efficient) requests. This can lead to significant performance gains, as traversals are time and memory-intensive in the case of large models. Subsection 5.3 gives technical insights on optimization of this kind in our technical context.

4.5 Optimizing the Model View Querying

This complex and final activity covers steps (3) to (7) in our overall process (cf. Subsection 4.1.2).

Once a model view has been correctly created and loaded (cf. the three previous subsections), it can be navigated and queried as any regular model according to the needs of the engineering activity it supports. As presented earlier, the model view framework usually relies on interfaces provided by one or several modeling technologies and shared in common between different tools from the same ecosystem. This way, it also supports the execution of queries defined in languages supported by these modeling technologies.

However, when implementing such a generic support in practice, performance issues can easily arise. For instance, some models can be serialized in XML-based files while others can be stored in databases via different modeling technologies and underlying persistence solutions (cf. Subsection 4.2). In this situation, the default querying support offered by a given modeling technology might not take advantage of backend-specific optimizations. Thus, more elaborate schemes should be considered in order to exploit their full potential.

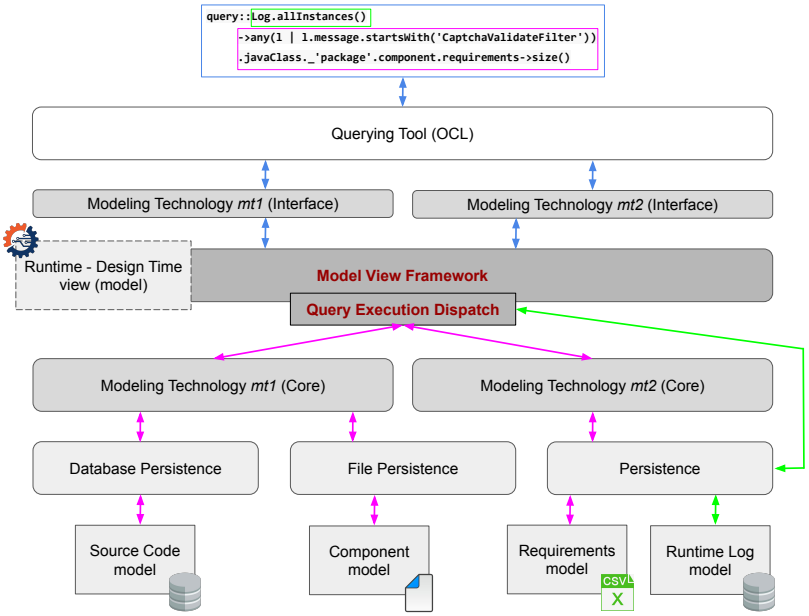

Fig. 5 Optimizing model view querying by delegating to modeling technologies and underlying persistence backends.

The optimization of model querying techniques has already been studied quite well (cf. Section 7). However, it has not been really explored so far in the context of model views combining models handled with different modeling technologies and possibly using different persistence backends. In our context, some base operations can be costly to execute on the model view when reusing only the default behavior of the underlying modeling technologies. For better efficiency, such operations could be delegated appropriately to the various modeling technologies and related persistence solutions. For example, this is illustrated in Figure 5 where an OCL query navigates the view from our use case and returns the number of design requirements that are impacted by a specified runtime log. This query can be optimized by delegating the `allInstances` call directly to the database persistence solution handling the model that contains the related `Log` elements. This way, the default (less efficient) implementation of `allInstances` is bypassed.

In addition, large performance gains are also possible by splitting a query (on a given view) into sub-queries better suited to the corresponding modeling technologies and their underlying persistence solutions. This requires the model view framework to provide a query execution dispatch mechanism: once a query is split, sub-query execution is then delegated appropriately and corresponding results are collected by leveraging the specific features of the different persistence backends. Complementary techniques, such as the parallel execution of different sub-queries or the replacement of sub-optimal operations, can be considered in some cases. Subsection 5.4 describes concrete steps we have made towards this in our technical context.

5 A Concrete Instantiation of our Conceptual Approach

In order to be able to perform actual experiments with our proposed conceptual approach, we first had to realize this approach in practice. The current concrete instantiation of our approach is depicted in Figure 6.

We propose an implementation of our conceptual approach that relies on the widely-used Eclipse Modeling Framework (EMF) [44] as our main underlying modeling technology. We also decided to use EMF because it already comes with a rich ecosystem of related model-based tools, e.g. for querying models or for persisting them using different kinds of data sources.

As introduced earlier, we have considered the integration of an alternative modeling technology to be combined with the use of standard EMF. We quite naturally opted for Epsilon [41], as it is also Eclipse-based (which facilitates the integration from a technical perspective) and well-known in the community. Interestingly in our context, Epsilon comes with its own family of modeling management languages, such as EOL for querying models. It also comes with model connectivity capabilities for complementary kinds of data sources. This way, we have shown that our conceptual approach can support the integration of different modeling technologies (i.e. EMF and Epsilon in the current instantiation). We are now able to build views (and query them) over more heterogeneous types of data sources.

Based on our own knowledge and expertise, we made the choice of using EMF Views [12] as our model view framework. This decision is also reinforced by the fact that EMF Views is Eclipse-based and relies on EMF. This way, we were able to extend it to be compatible with Epsilon. For similar reasons, we used NeoEMF [17] and CDO [21] as model persistence solutions supporting graph and relational database backends (respectively).

As we want to show the capability to build scalable views on heterogeneous modeling technologies, we now propose a different set-up than in our initial MODELS 2018 paper [11]. Thus, we voluntarily distributed the four contributing models of our view between the two different modeling technologies and different kinds of persistence solutions. Instead of using NeoEMF with EMF such as we already did in the past, this time we opted for using it via Epsilon. This way, we had the opportunity to develop an EMC connector for NeoEMF

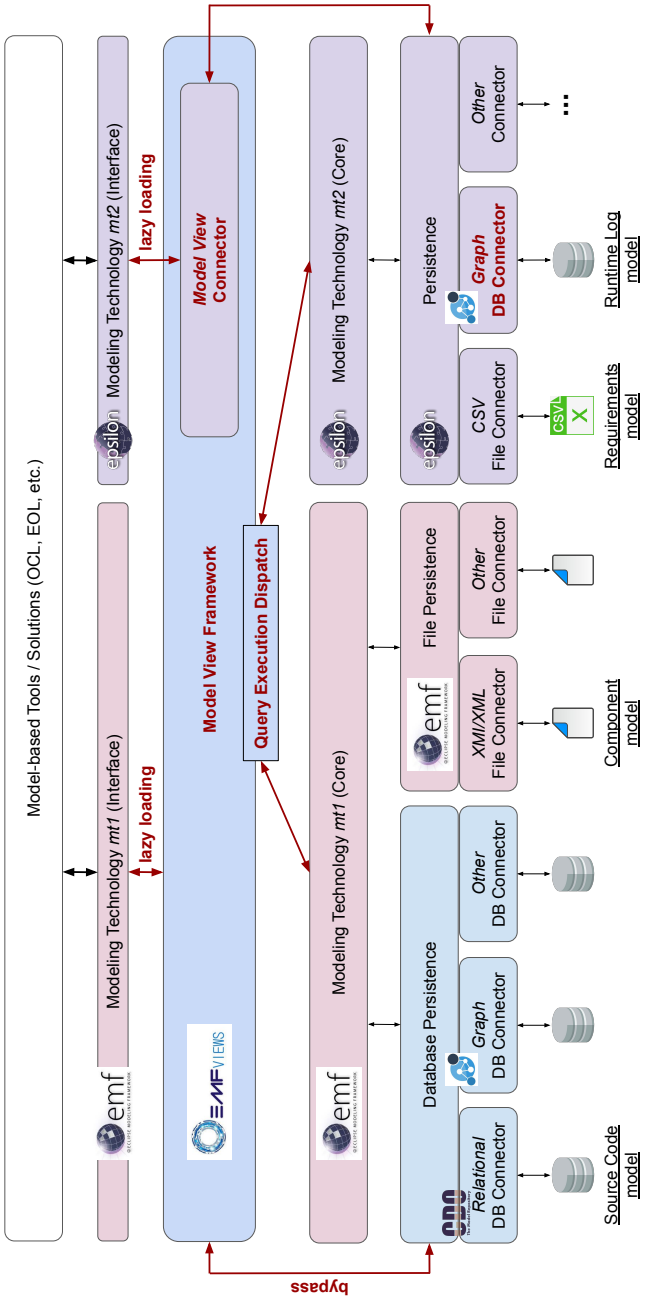

Fig. 6 A concrete instantiation of our conceptual approach (cf. Figure 4), based on Eclipse technical solutions.

(as not existing before) and corresponding optimizations. For distribution pur-

pose, we then opted for using CDO via EMF (instead of relying on the already available EMC connector for CDO).

In the next four subsections, we detail how all these technical solutions have been combined in practice in order to address our four different big objectives (cf. Subsection 3.1) with our conceptual integration approach (cf. Section 4). The complete source code of this current instantiation of our approach is freely available[5] and covers all the presented implementation details.

5.1 Building Model Views with EMF Views on Heterogeneous Modeling Sources via EMF and Epsilon

In this subsection, we describe how we integrated the EMF Views model view framework with the CDO and NeoEMF model persistence solutions via two different modeling technologies: EMF and Epsilon (respectively).

5.1.1 EMF Views for Model Views

EMF Views[6] [12] is a model view solution that proposes to reuse the well-known concept of *views* in databases and to transpose it to the modeling world. To this intent, it embeds a lightweight model virtualization framework that can be used on top of any EMF-based model. This enables the creation of (currently read-only) views that aggregate elements coming from different models. EMF Views itself is fully compliant with the EMF API and the views it produces act as standard models: they can be navigated, queried and taken as inputs of model transformations (for example). The associated approach has the following characteristics:

- Lightweight: Thanks to the model virtualization mechanism, elements in a view are only proxies to actual elements from the contributing models (which are never copied). This allows for a lower overhead in terms of memory when creating and navigating model views.
- Filtering: When expressed in the viewpoint specification, existing elements, attributes or references from the contributing models can be hidden in corresponding model views.
- Virtualization: While avoiding content duplication, views can possibly contain newly created elements, attributes or references that exist only at the view level and are not part of the contributing models.
- Non-intrusiveness: All additional view-specific information (pointers to contributing models, filters, virtual elements) are described in a separate *weaving model* [7]. This does not imply any change on the contributing models.

[5] `https://github.com/atlanmod/scalable-views-heterogeneous-models`

[6] `https://www.atlanmod.org/emfviews`

[7] In past work [12], we provide insights on this weaving model and notably on how it can be created (by using the ViewPoint Description Language) and then internally used within the model view solution (via the Epsilon Comparison Language engine).

EMF Views is available as a set of open-source Eclipse plugins. Its native integration with EMF-based tools and its characteristics provide an interesting starting point for our implementation. Note that alternative model view mechanisms are also discussed in the related work (cf. Section 7).

5.1.2 CDO and NeoEMF for Model Persistence

The Connected Data Objects model repository (CDO) [21] is a model persistence framework designed to handle large EMF models by relying on a client-server repository structure. CDO is based on a lazy-loading mechanism and supports transactions, access control policies as well as concurrent model editing. The default implementation of CDO uses a relational database connector to serialize models into SQL-compatible databases. However, the modular architecture of the framework can be extended to support different data storage solutions (even if, in practice, mostly relational connectors are frequently used and maintained).

NeoEMF[8] is a complementary model persistence framework that relies on the scalable nature of NoSQL databases to store and manipulate large models. NeoEMF supports three different model-to-database mappings, namely graph, key-value and column stores. Each one of them is meant to be particularly adapted to a specific modeling scenario, such as atomic element accesses (key-value) or complex navigation paths (graph). As other persistence solutions, NeoEMF provides a *lazy-loading* mechanism which delivers significant performance gains in different contexts.

Since CDO and NeoEMF are two state-of-the-art frameworks in the area of scalable model persistence, we chose to rely on them in the current implementation of our approach (cf. Section 7 for possible alternative solutions).

5.1.3 Epsilon as a Complementary Modeling Technology for Model Persistence and Querying

Epsilon[9] [41] is a family of extensible model management tools and languages for performing various modeling tasks, including notably model querying. All these languages are implemented independently from any modeling technology, so that they can be used with any type of model in theory (which is of particular interest in the context of the present work). This is achieved thanks to the Epsilon Model Connectivity (EMC) layer providing an abstraction for model access and CRUD (Create/Read/Update/Delete) operations through a number of common interfaces. The EMC interfaces are abstract/generic enough to support a broad range of technologies. Implementations of EMC are referred to as model *drivers*. Epsilon already has drivers for commonly used modeling formats such as EMF, XML, spreadsheet or CSV, as well as for other (proprietary) technologies such as Simulink. Interestingly, there are also drivers for

[8] `https://neoemf.atlanmod.org`

[9] `https://www.eclipse.org/epsilon/doc/`

database technologies. In addition, we also developed a specific EMC driver for NeoEMF in the context of the present work.

The Epsilon languages are built on top of a common interpreted OCL-inspired language called the Epsilon Object Language (EOL). Like OCL, EOL is suitable for declarative model querying via an OCL-compatible syntax and standard library of first-order operations (such as *select*). Moreover, EOL comes with imperative programming constructs such as loops and mutable variables. Finally, EOL also supports the parallel execution of some operations as well as laziness through data streams [35].

5.1.4 Integration

Since EMF Views, NeoEMF, CDO and Epsilon are all Eclipse-based (and are packaged as sets of Eclipse plugins), they can be easily integrated together into a single Eclipse workbench. Moreover, EMF Views, NeoEMF and CDO are part of the EMF ecosystem, making their integration even more straightforward since they implement the same EMF model handling interface. It is also interesting to note that Epsilon provides an EMC driver for EMF, thus allowing it to interoperate with EMF-based solutions (such as EMF Views).

In all cases, it is mostly a matter of telling EMF Views how to properly retrieve and load the right model resources. However, CDO and NeoEMF model 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 and Epsilon (e.g. via an EMC driver for NeoEMF, cf. Subsection 5.1.3).

Once correctly loaded, all the model resources are navigated through the standard modeling interface of EMF and/or Epsilon (depending on the case). When required, the model view framework transparently delegates (in a scalable manner) the operations to the appropriate modeling technologies and underlying persistence solutions.

5.2 Persisting the Model View Information with NeoEMF or Creating it Dynamically

As explained before, EMF Views uses a *weaving model* that represents the view-specific information. This model can potentially contain entries for many elements coming from the different contributing models. In practice, it can get as large or even larger (depending on the view) than the contributing models themselves. In order to deploy our approach on large-scale views (such as in our MegaM@Rt2 use case), we need to be able to persist or create such weaving model in a scalable way. However, it is important to note that the focus of our experiments in this paper is on model view loading and querying only (cf. Section 6). To this end, we consider that the weaving model is already existing and correct. Thus, how the weaving model has been produced in the first place, how inconsistencies in the inter-model links are handled and how the

view can be kept synchronized based on these links are challenges voluntarily not tackled here (even though relevant as well from a more specific model view perspective).

In our MODELS 2018 paper [11], we already showed that it is possible to handle the model view information in a scalable way by relying on NeoEMF (instead of using the default XMI serialization provided by EMF). Since the corresponding weaving model is also defined as a standard EMF model, its migration to NeoEMF was done quite transparently by changing the model serialization behavior (and initializing the corresponding database backend). Persisting the weaving model in NeoEMF allowed us to handle views that cannot fit in memory otherwise.

In this extended paper, we worked on adding an extra heterogeneity dimension to our approach (while still maintaining an interesting level of scalability) with the integration of Epsilon as an alternative modeling technology complementary to EMF. To this intent, we opted this time for dynamically creating the required model view information at runtime when the view is actually loaded. Thus, in the experiments presented later in Section 6, the weaving model is initialized programmatically via the implementation of an optimized algorithm relying on hash tables.

5.3 Optimizing Model View Loading and Model Element Access in EMF Views

When dealing with large database resources (such as in our use case), many operations of the EMF interface that had little to no overhead with small in-memory resources now potentially bear high costs in execution time and memory consumption. Thus, we had 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 native 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 the nth element of a multi-valued reference by using the *EList.get* operation can be costly if 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 fewer hits to the database and minimize the cost of the operation. We significantly improved the current EMF Views implementation to support large model resources by following these ideas.

A second point of optimization was to tweak the way the data is stored into the graph database handled by NeoEMF. The runtime log model of our use case is a large model but a flat one. It contains a top-level element holding

a large collection of execution logs (some of which have also children). Our experiments have shown that this flat structure was reducing the performance 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 such 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.

5.4 Optimizing the Model View Querying in OCL and EOL

Since views are regular EMF models, querying tools like OCL/EOL or transformation tools like ATL can be applied transparently on views (regardless of the persistence solutions used by the contributing models). However, relying only on the EMF interface can limit performance. Base operations, like `allInstances` in OCL, can be quite costly to execute natively using the EMF API [48]. Persistence backends may provide more efficient ways to execute them. For example, NeoEMF resources expose a `getAllInstances` method that is about 40 times faster than using the EMF interface directly.

We extended the standard OCL interpreter in order to specialize some operations according to the data store they target. We detail here our implementation of the `allInstances` operation, but other native operation implementations can be easily defined to further enhance performance.

The OCL interface allows customizing the behavior of the `allInstances` operation through an *Extents Map* (*Model Manager* in newer implementations). Thus, we defined 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 class are looked up, the extents map redirects the call to the view that fetches instances—using native database calls—from each contributing model and then combines these instances as the result. We show later in our evaluation that such dispatching mechanism can dramatically improve query computation performance (cf. Section 6).

As a concrete example, when executed the OCL query `Log.allInstances()` goes through the extents map to find all instances of the `Log` class. With the default extents map, OCL iterates over the entire model view (having several contributing models), tests each encountered element and keeps only the actual instances of the `Log` class. This default iteration process is completely generic but slow. With our dispatching mechanism, the custom extents map looks up which contributing models contain instances of the `Log` class. In our MegaM@Rt2 use case, the Runtime Log model is handled via Epsilon and persisted as a NeoEMF resource. Thus, the extents map delegates the instances lookup directly to the NeoEMF backend (using the `getAllInstances` method). This backend then uses built-in indexes and caches to return the result about 40 times faster.

Within Epsilon, the *IModel* interface provides several methods that can be overridden to optimize how EOL and other Epsilon languages interact with the resource. We optimized the `allInstances` (also simply called `all` in EOL)

operation by overriding the `IModel.getAllOfKind` method in the same way as described before. In addition, by returning a bespoke collection object in `getAllOfKind` which implements *IAbstractOperationContributor*, we can intercept other kinds of operations that will act on the results of the `all` operation. Thus, operations such as `select` or `collect` can then be overridden with our own implementation optimized for NeoEMF.

Moreover, capturing these operations allows us converting entire EOL expressions on NeoEMF resources (e.g. the Runtime Log model in our MegaM@Rt2 use case) into native graph traversals expressed in the Gremlin query language [2]. These traversals are then computed over the Gremlin virtual machine, directly benefiting from advanced optimizations such as database-level lazy-loading and caching. The concrete translation from EOL operations to Gremlin traversals is adapted from our previous work providing a set of mappings from OCL operations to Gremlin traversal parts (also called *steps*) [18]. Note that EOL-specific operation mappings are left for future work. Generally, such query rewriting approach has been inspired by a similar scheme to translate OCL expressions into SQL queries [33]. This allows us executing queries entirely in the database rather than in memory, thus improving performance regarding execution time and memory consumption.

6 Evaluation

In order to evaluate our integration approach and its current implementation, we applied them on our motivating use case from the MegaM@Rt2 project (cf. Subsection 3.2).

In this evaluation, we focus on measuring the *time overhead* of our current implementation because it directly impacts the interactive user experience (as opposed to batch processing), for instance when navigating and/or querying the view. The overall objective of the performed experiments is to show in practice that one can fully benefit from our approach (i.e. the support for heterogeneous modeling technologies and resources) while paying a reasonably low cost in terms of scalability (considering either small/medium or much larger models). As dealing with on-disk resources is inevitably (one to two orders of magnitude) slower than dealing with fully in-memory resources, matching the speed of such in-memory resources is not a realistic goal. Thus, we rather insist on the asymptotic behavior of our approach and on the significant performance gains we already obtained by our different optimization actions.

For reproducibility, the complete source code of the performed benchmarks (including the contributing models and viewpoint/view specifications) and more detailed results are available online[10]. This repository also contains links to the versions of the various tools we used for these benchmarks.

All the benchmarks have been realized on a HEDT system with the following specifications: AMD Ryzen Threadripper 1950X 16-core CPU @ 3.5

[10] `https://github.com/atlanmod/scalable-views-heterogeneous-models`

GHz ("Creator Mode"), 32(4x8) GB DDR4-2933 MHz RAM, Samsung 960 EVO 250 GB M.2 NVMe SSD, Fedora 30 OS (Linux kernel 5.1.12, mitigations=off), OpenJDK 11.0.3 Server VM with the following parameters "-Xmx28g -XX:MaxGCPauseMillis=550". The latter parameter was used to improve throughput with the default G1 garbage collector.

In terms of experimental process, we ran each benchmark 15 times and discarded the 5 first runs (used as a warmup for the JVM). Thus, in the following tables, we report on the arithmetic mean for the 10 latter runs only. Still, we might have had some pre-fetch optimizations taking place in such setup. To overcome a possible "static" setup bias, we could consider running more dynamic and randomized scenarios in additional future experiments. Nevertheless, we do not report on any significant deviation with the current setup as we have observed that it was consistently under 5%.

6.1 Overview

In order to perform our experiments, we built two versions of the same view implementing our MegaM@Rt2 use case:

1. As a point of comparison, we created a first version of the view that is fully file-based: all four contributing models are serialized using standard EMF-XMI. Thus, once loaded, the view resides fully in memory (simulating a "default" behavior).
2. The second version aimed at demonstrating our capability to build views over heterogeneous modeling technologies and resources. It uses a mix of file-based and database resources as contributing models. More precisely, the Runtime Log model is persisted into a Neo4j graph database handled by NeoEMF, using our optimized NeoEMF connector for Epsilon (cf. Subsection 5.3). The Java Source Code model is persisted into a relational database handled by CDO. The UML Component model is serialized as an XMI file handled by the standard EMF implementation. The Requirements model is serialized as a plain CSV file handled by the Epsilon CSV connector.

In both cases, the performed experiments also aimed at evaluating the scalability of the overall approach and current instantiation. To this intent, for each version of the view, 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).

6.2 Benchmark 1: Loading the Model View

In the first benchmark, we evaluated both the loading of a view and the iteration over all its contents. We performed this experiment on the two versions of

Table 1 Average time (in milliseconds) to load the two versions of the view, corresponding calculated overhead factor between these two versions.

Size	XMI (in ms)	Hetero. (in ms)	Overhead factor
10^1	161	6942	43.12
10^2	160	4975	31.09
10^3	191	5030	26.34
10^4	523	6290	12.03
10^5	3832	17885	4.67
10^6	36154	140492	3.89

the view (full XMI vs. databases + file-based). This benchmark measures the overhead of 1) matching model elements to create the view-specific information and 2) accessing the content of the different models contributing to the view.

Indeed, in our use case from MegaM@Rt2, we first have to compute the view-specific information needed to combine the four models contributing to the view:

- We connect a given execution log (from the Runtime Log model) to the Java class declaration (from the Source Code model) that emitted it.
- We relate the Java package this class declaration is part of (from the Source Code model) to the UML component (from the Component model) that represents it at design level.
- We link this UML component (from the Component model) to the corresponding CSV row, i.e. the requirement (from the Requirements model) the UML component aims to support.

As a consequence, creating the view-specific information (materialized as a weaving model) implies checking for matches between two elements coming from two contributing models. For large models, such as the Runtime Log model from our MegaM@Rt2 use case, these matches can rapidly become very numerous.

Table 1 compares the time it takes to load the two versions of the view, while Table 2 compares the time required to iterate over the full content of the two versions of the view. In both tables, the third column provides the overhead factor calculated by dividing the value in the case of the heterogeneous view by the value in the case of the XMI only view.

A first immediate observation of Table 1 shows that, for smaller model sizes (i.e. from 10^1 to 10^4), loading the heterogeneous view takes between 4975 and 6942 milliseconds. The loading time in the case of the full XMI view is significantly faster (quasi-equal to or less than 500 milliseconds). This can be explained by the additional operations needed to setup and open the related database resources, as well as by the inherent higher latency of doing disk I/O. However with larger models, the ratio between the two versions diminishes progressively and significantly. This can be explained by the fact that the time spent to compute the view-specific information becomes more dominant. In the general case, it is possible to mitigate the actual consequences of such

Table 2 Average time (in milliseconds) to iterate over the full content of the two versions of the view, corresponding calculated overhead factor between these two versions.

Size	XMI (in ms)	Hetero. (in ms)	Overhead factor
10^1	595	632	1.06
10^2	592	641	1.08
10^3	599	656	1.10
10^4	650	832	1.28
10^5	1125	2439	2.17
10^6	5989	18537	3.10

results. We can argue that, in many real-life scenarios, users can navigate and run multiple queries over a same model view without having to fully (re)load it each time. Moreover, it is also possible to speed up the loading process by computing the view-specific information in advance, at the cost of storing this information. For instance, this is what we already did in the context of our initial MODELS 2018 experiments [11].

When iterating over the full content of the view (cf. Table 2), we can observe that the overhead remains relatively limited. We can note a tendency to slightly increase as the Runtime Log model gets larger. For example, for the largest model size, the heterogeneous view is around 3 times slower to navigate than the full XMI one. We can notably observe that this ratio is still below the expected speed difference between RAM and disk. This 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. Again, such results can be mitigated by the fact that full iteration scenarios may not be very common in practice. However, it is worth mentioning that the choice of the data representation (when using such database resources) can have a strong impact on performance.

6.3 Benchmark 2: Querying the Model View

In the second benchmark, we measured the time it took to successfully run three different queries on top of our model view. From an user perspective, these queries correspond to three possible types of usage of such a view in the context of our use case from MegaM@Rt2. From an evaluation perspective, they correspond to various types of access and navigation concerning the different models contributing to the view.

A first base query allows performing an example of computation focusing only on runtime data: It simply counts all the instances of `Log` elements in the view, and thus only accesses the Runtime Log model via the view. A second and third more elaborated queries allow considering the full runtime $\leftrightarrow$ design time feedback loop (in both directions): They traverse the complete view, i.e. they access elements from all four contributing models, and navigate both inside and between these models, by using the view-specific information (cf. Subsection 6.2).

Table 3 Average time (in milliseconds) to run OCL query (1).

Size	XMI	XMI (Opt.)	Hetero.	Hetero. (Opt.)
10^1	618	10	658	40
10^2	614	10	661	41
10^3	623	10	694	44
10^4	675	12	859	57
10^5	1235	63	2729	227
10^6	6643	753	19875	2188

Table 4 Average time (in milliseconds) to run OCL query (2).

Size	XMI	XMI (Opt.)	Hetero.	Hetero. (Opt.)
10^1	609	11	649	41
10^2	610	11	652	41
10^3	616	11	667	41
10^4	668	10	853	42
10^5	1167	35	2587	42
10^6	6277	272	19789	61

Table 5 Average time (in milliseconds) to run OCL query (3).

Size	XMI	XMI (Opt.)	Hetero.	Hetero. (Opt.)
10^1	632	10	667	41
10^2	626	10	673	41
10^3	619	11	693	43
10^4	672	14	875	86
10^5	1212	66	2635	268
10^6	6744	709	20653	2651

The considered OCL queries are the following (semantically equivalent queries expressed in EOL have also been considered in this benchmark):

1. `Log.allInstances()->size()`
2.
```
csv::Row.allInstances()
->any(r| r.desc.startsWith('Controller')))
.components->collect(c| c.javaPackages)
->collect(p| p.ownedElements)
->selectByType(ClassDeclaration)
->collect(c| c.traces)
->size()
```
3.
```
Log.allInstances()
->any(l| l.message.startsWith('CaptchaValidateFilter'))
.javaClass._'package'.component.requirements
->size()
```

Table 3 compares the time it takes to execute query (1) on the two versions of our model view. Tables 4 and 5 do the same for queries (2) and (3), respectively. In these three tables, the two additional *(Opt.)* columns refer to optimized model views that use the custom extents map we described in Subsection 5.4.

Table 6 Average time (in milliseconds) to run EOL query (1).

Size	XMI	Hetero.	Hetero. (Opt.)
10^1	1	3	3
10^2	1	3	2
10^3	1	13	3
10^4	6	131	17
10^5	70	1330	176
10^6	700	12986	1851

Table 7 Average time (in milliseconds) to run EOL query (2).

Size	XMI	Hetero.	Hetero. (Opt.)
10^1	3	2	2
10^2	2	2	2
10^3	1	2	3
10^4	2	2	2
10^5	2	2	3
10^6	15	14	14

Table 8 Average time (in milliseconds) to run EOL query (3).

Size	XMI	Hetero.	Hetero. (Opt.)
10^1	1	2	4
10^2	1	3	3
10^3	2	14	4
10^4	6	139	18
10^5	70	1371	177
10^6	702	13244	1822

A general observation is that querying the non-optimized heterogeneous view is, overall, just slightly slower than querying the full XMI view (apart from the case of the largest model in which the difference is more important). This shows that the general overhead of using our approach is relatively low. In addition, the optimized versions are providing a 10- to 300-time improvement which is significant. Notably, they bring down the time to run all queries under 1 second (except from the case of the largest model) . This can be considered as a satisfactory response time from a usability perspective (either by a human or a depending tool). As a particular example, 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. Finally, we can note that the optimization we have implemented also benefits the full XMI view, although less importantly.

As mentioned earlier, we also wrote three semantically equivalent EOL queries. Again, we compared their execution on the two versions of our model view. The corresponding results are shown in tables 6, 7 and 8 (respectively). It is important to note that, since the EOL connector's optimization does not apply to XMI resources (i.e. this optimization targets only NeoEMF resources in our present case), the XMI (Opt.) column does not appear in these three latest tables.

Concerning the execution of the first EOL query, we can remark that the results obtained with the EOL optimization are somehow equivalent to the results obtained with the optimized OCL version. This shows that, thanks to our extended approach, we are now able to switch from one query language to the other without significantly impacting performance (at least in the case of these types of queries).

Concerning the execution of the second EOL query, we can observe that the proposed optimization does not have any noticeable effect (as results were already in the 10 milliseconds range without it). This shows that the use of our approach, with the integration of Epsilon as an alternative modeling technology, is quite transparent in this case. Indeed, it does not come with a distinct overhead in terms of performance. However, more experiments (with more varied queries, cf. next paragraph for instance) would be required in order to be able to fully generalize this.

Concerning the execution of the third EOL query, we can note that the improvement brought by the optimization is on the same order of magnitude (10×) as it was for the same query in OCL. This shows that our optimizations are consistent across query execution engines.

Finally, Figure 7 aggregates the results obtained with both OCL and EOL in order to represent them in a graphical way. We can observe that EOL queries are generally more efficient than OCL ones in terms of execution time. Interestingly, the optimizations proposed in this paper allow to obtain significant performance gains in both cases.

Overall, the performed evaluation shows that we successfully integrated the use of Epsilon as a complementary modeling technology (and provider of alternative modeling resources) in our approach. We obtain relevant results in terms of scalability of the global integration solution, from model view loading to model view querying. Nevertheless, there is still room for further improvements on different related aspects (cf. Section 9).

7 Related Work

In this section, we compare our approach with existing work from the state-of-the-art in the related areas of model views and model queries.

7.1 Model Views

Several existing solutions have been proposed in order to support the definition, creation and handling of views over models. In another research effort [7], we have studied these solutions, described them and tried to classify them according to their main characteristics. Voluntarily, we do not go into such detail again in the present paper: we point the reader to this large study for more detailed information. In what follows, and directly based on the results of this study, we rather provide a summary of the situation concerning the two

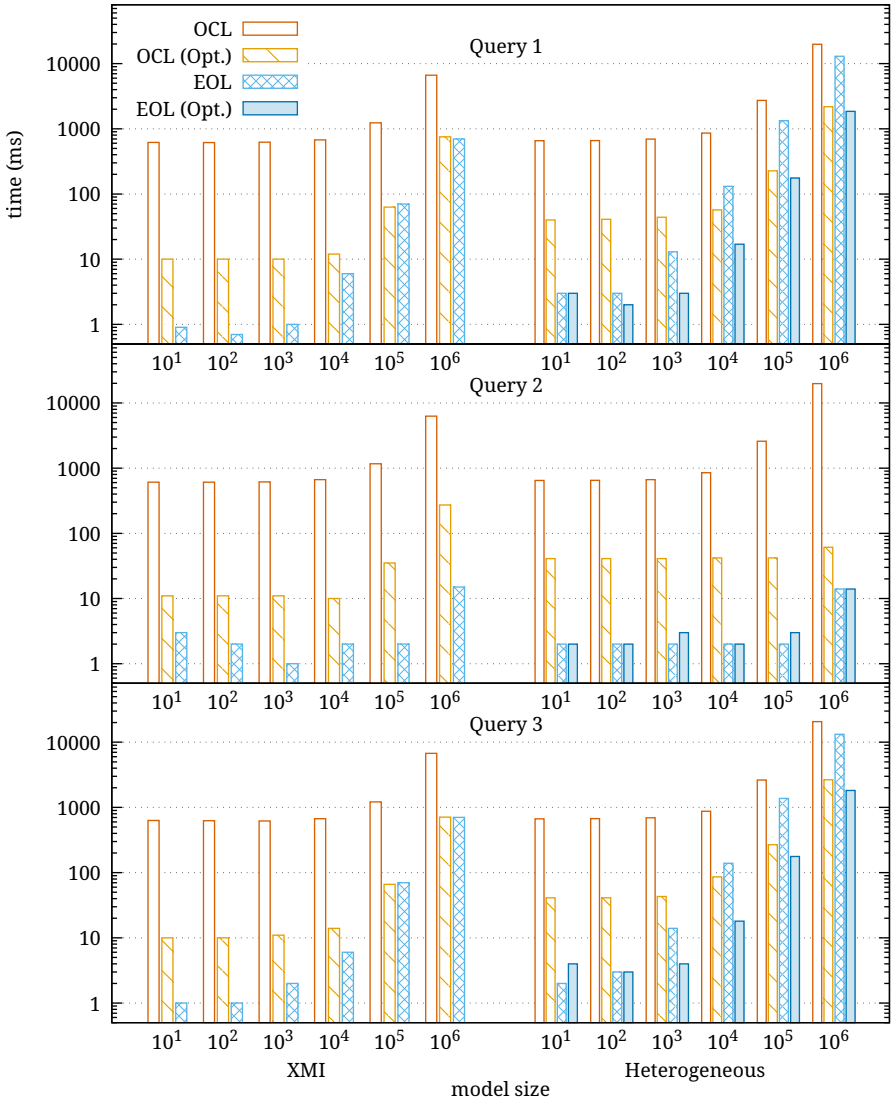

Fig. 7 Aggregated results from Tables 3 to 8. Both axes are in logarithmic scale.

main properties we consider in this paper: scalability (cf. Subsection 7.1.1) and heterogeneity (cf. Subsection 7.1.2). We have notably observed that, up to our current knowledge, there does not seem to be any solution specifically addressing scalability issues in a strongly heterogeneous context (such as the one we consider in this paper).

7.1.1 Scalability

In the general case, the observations resulting from our study indicate that there is little evidence regarding the scalability of the available solutions in the context of model views combining very large models. This current limitation can be considered as an important issue in this area, as scalability is key to the deployment and actual use of model view solutions in realistic industrial contexts. As we have already seen in past work [11], building scalable model views is a challenging problem. With the extended work presented in this paper, and notably with the performed experiments, we intend to take a step in this direction.

Moreover, scalability is also a fundamental aspect as far as related view-specific challenges are concerned. For instance, as we have seen in our detailed study, this is the case for the view update problem (as already a long-term concern in the database community [3]) or for the issue of the incremental maintenance of model views [36]. These particular model view challenges, even though important to be addressed, are not the core topic of the work presented in this paper. However, this shows that there is still room for interesting and relevant future work by extending further our current approach in order to support such capabilities (cf. Section 9).

7.1.2 Heterogeneity

In terms of heterogeneity, relatively few existing model view approaches provide support for integrating interconnected models whose content is coming from very different data sources. There are indeed approaches allowing to support views combining several models possibly having different metamodels (cf. the features *Type Structure - Metamodel Arity and Model Arity* from our feature model for model view approaches [7]). We already identified them in our past study and they are briefly introduced again in the following paragraphs. Nevertheless, as it can be seen hereafter, they are generally rather limited in terms of the underlying modeling technologies and resources that can be used to create and handle corresponding model views.

For example, ModelJoin [13] proposes a DSL for querying altogether different models and building a view as a result. However, at the time of writing, it does not fully support a variety of modeling technologies and related model persistence backends other than EMF-based ones. The same is also true for the Sirius framework [43], as well as for the Kitalpha framework [34] that relies on Sirius, whose final objective is to facilitate the creation of modeling workbenches proposing different kinds of views. They are both directly built upon Eclipse/EMF and do not natively aim at integrating other modeling technologies and related resources (even if customer-specific extensions could be developed in a case-by-case manner). In the same vein, the VIATRA Viewers solution[11] that emerged from the EMF IncQuery initiative [46] focuses more

[11] `https://www.eclipse.org/viatra/documentation/addons.html`

on scalable querying rather than on supporting such heterogeneity (cf. also Subsection 7.2).

Alternatively, OpenFlexo [24] has been designed as a way to support several data sources (whether they are model-based ones or not). However, it privileges graphical (manual) modeling to build views and does not come with an extended (and optimized) querying support at view-level. This latest point is of a particular importance in our current industrial MegaM@Rt2 context (for instance). Another relevant approach is the Epsilon family of languages [41]. Among the available languages, Epsilon provides EML [30] that allows merging models, and Epsilon Decoration [32] that allows decorating existing models with additional features. These languages can be considered as providing kind of a (partial) support for model views. Nevertheless, we have shown in this paper that Epsilon is rather a good complementary solution to our approach. Notably, we have shown in practice (with our experiments on the MegaM@Rt2 use case) that Epsilon can be effectively integrated as an alternative modeling technology and resource provider for the EMF Views model view solution [12].

7.2 Model Querying

In addition to solutions specifically dedicated to model views, there has also been several initiatives more related to model querying challenges in general. For example the VIATRA project [47], relying on IncQuery [46] for efficient incremental querying features, is a reactive model transformation platform. The platform also embeds some capabilities for manipulating multiple models altogether and defining sorts of views over them [19]. In terms of querying support, VIATRA and our approach work best in different scenarios. VIATRA is very efficient when the contributing models fit in memory and queries are executed multiple times over the same "view". Our integration solution is rather designed to be able to access models from different kinds of modeling resources (notably from databases), and to benefit from their internal structure in order to efficiently perform single query computations. However, incremental querying capabilities could be integrated to our solution by experimenting on the use of IncQuery over our views (in addition to OCL and EOL, as already described in this paper).

Quite recently, a few approaches have been proposed in order to improve the computation of model-level queries over several scalable persistence solutions. For example, the Mogwaï tool [18] is a translation approach that maps OCL constructs to Gremlin [2] which is a graph database language. In this approach, automatically generated queries are sent directly to the database for computation, thus bypassing some limitations of current modeling frameworks. A similar approach is used in the Hawk query framework [4] that dynamically translates Epsilon queries into calls to graph database native operations. However, such solutions focus on specific data sources and are not designed to handle heterogeneous backends, as in the kind of model views we intend to handle in the present paper. Nevertheless, they could be integrated to our so-

lution in order to speed up (parts of) queries related to the particular backend they target and optimize.

Finally, there is also some related work in the data warehouse community. In this research area, querying views on heterogeneous storage solutions has already been studied, for example in the context of relational databases [25]. This type of approaches regained interest quite recently thanks to the emergence of polystore data warehouses [14]. Simply said, the goal of a polystore database management system is to allow building databases on top of multiple and heterogeneous storage engines. This is kind of a similar objective to what we intend to achieve with the present paper in the model view area. Still in the same vein, CloudMdsQL [29] is an SQL-like language for querying multiple data stores by using a single query. To do so, it extends the standard SQL syntax with additional constructs allowing to directly embed various native datastore queries. A similar approach can also be found in generic query frameworks such as the Apache Drill open source framework[12].

In the general case, such querying solutions could also be reused and/or integrated to our approach in order to improve the overall computation of (parts of) queries on views combining models stored in even more heterogeneous data backends. Such a more advanced integration would raise two interesting challenges in this particular area: 1) the (automated) translation of the model view queries to the (generic) database query languages and 2) the (correct) integration of the database implicit schemes of the various models contributing to the view (as required to produce queries possibly taking full advantage of the target backend capabilities).

8 Lessons Learned

While working on integrating our model view solution with heterogeneous modeling technologies and their respective model backends, we have been able to acquire some experience that could be beneficial (to us of course, but also to the reader we believe) in different kinds of software modeling and/or engineering processes. We briefly share this experience hereafter:

- **Using a standard (modeling) API really makes tool integration easier, but it has to be done with caution.** From our experience in this paper, we can attest the benefits of relying on a standard API (such as the EMF one) when it comes to integrating together different tools. This largely facilitated the initial combination of the EMF Views, NeoEMF and CDO tools in order to obtain a solution working on basic cases. However, relying on such generic API can also hinder performance when tackling larger-scale scenarios. Firstly, the generic API may hide some of the features and capabilities of the underlying tools, and so prevent from using them. For instance, the NeoEMF API exposes additional efficient methods for navigating model resources. Unfortunately, these methods are

[12] `https://drill.apache.org/`

not available to EMF Views when only relying on the standard EMF API. Secondly, using such a generic API can also hide the actual impact of some actions. Various backends may have weak spots in their implementation of the generic API, and some usage patterns may be preferred to others for performance or even correctness reasons. For example, when creating a model and adding elements to it using EMF, the order of some operations may matter to the persistence backends while it may have no impact on standard in-memory resources. In this case, a naive use of the generic API may lead to surprising performance issues. In order to minimize the overhead of views on large models, we have to consider elaborate strategies as a more efficient solution. We can notably identify when it is relevant to delegate a given operation or not, and to which persistence backend in particular. Thanks to our approach, we have been able to embed such practical knowledge into our integrated solution. This way, users do not have to pay the full cost of using the generic API while still benefiting from its genericity and interoperability capabilities.

- **Using a more abstract (modeling) API does not prohibit optimization, but it still requires some effort.** In addition to the use of the EMF standard modeling API, we have made a complementary use of the EMC API from Epsilon. It is worth mentioning the relative ease with which this EMC API allows for specific optimization to be made and integrated into our overall solution. We believe this comes from the fact that this EMC API is more general/abstract than the EMF one. Indeed, contrary to EMF that is strongly object-oriented, EMC has been natively designed to support different kinds of modeling resources and structures. We consider this as a practical example (among others) where good practices in API design does not prohibit or complicate the implementation of performance optimization. Let us take a concrete example of one current shortcoming in EMC, and introduce the effort that would be required to optimize it. The *allInstances()* (which returns all model elements) and *getAllOfKind* (which returns all model elements of a specified kind) operations both return a Collection, and not a Stream or Iterable, with no alternative like *streamAllOfKind()* or *streamInstances()*. Although it is generally possible to create a lazy collection, the use of a Collection usually implies a finite and complete in-memory data structure which is non-lazy (and so costly). Moreover, the full result of such operations is rarely needed: they are mostly used as a starting point for queries and transformations where only a subset is required. Thus as a solution, providing a lazy implementation of these EMC operations (possibly with different return types) would be preferable, especially for large models or memory-constrained systems. If we generalize, a well-designed modeling interface should be expressive enough to support all necessary features/operations without relying too heavily on implementation details which could limit potential optimization.
- **Packaging the tooling/benchmarks facilitates portability and reproducibility, but it comes with non-negligible cost.** In this paper,

we have presented an overall approach and its concrete Eclipse-based instantiation that integrates together different model-based frameworks and tools. We have also described the benchmarks we have performed on our MegaM@Rt2 use case in order to evaluate our solution in practice. In order to benefit from interesting computing resources available at the University of York, we decided to run the performed benchmarks only in their premises. However, most of the technical developments were done remotely (i.e. in France and Spain). Thus, in our work we had to consider multiple development environments and one single benchmark environment. As a solution, we first had to realize a complete reusable packaging of both the tooling for the solution and the full benchmarks on which this solution had to be run (including all the needed modeling resources). This task required a significant development effort, which is not something you can immediately valorize from a research perspective. Nevertheless, we have been able to gain important benefits afterwards. Firstly, this has allowed us running intermediate benchmarks in parallel of bug fixing and of new developments on the current version of the solution. This way, our development process and our progresses towards the latest version of the solution presented in this paper have been made more agile. Secondly, this has also allowed a better testing of the scalability aspects of our solution by having access to more powerful resources. Thanks to this, we have been able to go a step further than with our initial experiments from the MODELS 2018 paper [11] (that this paper extends). Thirdly, this has allowed us evaluating the portability of our overall solution to another environment than the one of our local machines (on which the technical developments have been made). We believe that this generally contributes to facilitate the reproducibility of our results, now and in the future (within the context of other technical environments on which our solution could be deployed).

9 Conclusion and Future Work

In this paper, we presented our approach to support the efficient creation and handling of scalable model views over heterogeneous modeling technologies and resources. We detailed our conceptual approach (i.e. the concepts, process and main associated activities) to integrate a model view framework with different modeling technologies, and with their underlying model persistence capabilities. Notably, we emphasized on four important objectives to be addressed. We then described our concrete Eclipse-based instantiation of this conceptual approach, now able to use both the EMF and Epsilon modeling technologies at the same time (and the various model backends they come with). In order to demonstrate the applicability of our global solution, we evaluated it on a realistic model view use case coming from MegaM@Rt2 project. The presented work has shown that we are already able to build and query model views in a sufficiently scalable way from a usability perspective. More interestingly, they have also shown that such model views can be based on contributing models

which are handled by heterogeneous modeling technologies and their modeling resource providers (while still preserving satisfactory overall performance).

Nevertheless, there is still room for interesting improvements in different aspects of our global solution.

Firstly, to go further in terms of querying capabilities over model views, we could study how incremental querying techniques [6] could be integrated into our approach. Based on such techniques, we could expect to have interesting performance gains in some specific situations (e.g. in the case of the multiple execution of queries over a same model view). Moreover, this could be performed in a complementary way to the reuse of some other already proposed optimization, at OCL-level for instance [50]. This could allow pushing the scalability of our approach even further.

Secondly, in addition to pure model querying, we could extend our experiments by also testing model transformation tools over our model views, such as ATL [28] or VIATRA [47]. In such cases, a given model view could be taken as input to a model transformation in a transparent way, instead of taking as inputs the various models contributing to this view. For optimization purposes, we could use the view information (e.g. which contributing model is stored where and how) in order to delegate parts of the transformation computation directly to the underlying modeling technologies and their backends. One possibility to realize this could be to integrate the use of scalable query/transformation approaches, such as Mogwaï [16] for instance.

Thirdly, from a model view perspective, we could refine our approach in order to support more advanced features such as view update (i.e. synchronization from the view to the contributing models) or view maintenance (i.e. synchronization from the contributing models to the view). These are complex problems that have already been studied in the past in other domains (e.g. in databases [15]). The fact that we intend to rely on heterogeneous modeling technologies and resources (and not only on homogeneous ones) could also add an extra degree of complexity. In the modeling community, there are some recent research efforts [42,36] we could already try to capitalize on. Moreover, studying the capability to create and save complete snapshots of model views at given points in time could also be helpful to this respect.

Finally, from a more practical perspective, we are going to continue the developments around our model view solution in general. Notably, we plan to apply it in the context of other real-life use cases in order to evaluate further its adaptation and deployment capabilities. For example, as part of the MegaM@Rt2 project, we have recently used EMF Views in conjunction with the JTL traceability solution to provide a runtime-to-design feedback loop in the context of a safety critical system from CLEARSY [23]. In terms of additional future use cases, we already have plans to build views tracing the architectural models of an industrial system with runtime models representing the configuration and running of corresponding physical machines.

References

1. Afzal, W., Bruneliere, H., Di Ruscio, D., Sadovykh, A., Mazzini, S., Cariou, E., Truscan, D., Cabot, J., Gomez, A., Gorronogoitia, J., 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. Microprocessors and Microsystems **61**, 86 – 95 (2018)
2. Apache TinkerPop: The Gremlin Language (2020). URL `https://tinkerpop.apache.org/gremlin.html`
3. Bancilhon, F., Spyratos, N.: Update Semantics of Relational Views. ACM Transactions on Database Systems (TODS) **6**(4), 557–575 (1981)
4. Barmpis, K., Kolovos, D.: Hawk: Towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, p. 6. ACM (2013)
5. Barmpis, K., Kolovos, D.S.: Comparative Analysis of Data Persistence Technologies for Large-scale Models. In: Proceedings of the 1st XM Workshop, pp. 33–38. ACM (2012)
6. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF models. In: International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), pp. 76–90. Springer (2010)
7. Bruneliere, H., Burger, E., Cabot, J., Wimmer, M.: A Feature-based Survey of Model View Approaches. Software & Systems Modeling **18**(3), 1931–1952 (2019)
8. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: A Model Driven Reverse Engineering Framework. Information and Software Technology **56**(8), 1012–1032 (2014)
9. Bruneliere, H., Eramo, R., Gomez, A., Besnard, V., Bruel, J.M., Gogolla, M., Kästner, A., Rutle, A.: Model-Driven Engineering for Design-Runtime Interaction in Complex Systems: Scientific Challenges and Roadmap. In: MDE@DeRun 2018 workshop, co-located with the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences, *Software Technologies: Applications and Foundations (STAF 2018) Workshops*, vol. LNCS 11176. Toulouse, France (2018)
10. 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: European Conference on Modelling Foundations and Applications (ECMFA 2015), pp. 62–74. Springer (2015)
11. Bruneliere, H., Marchand de Kerchove, F., Daniel, G., Cabot, J.: Towards Scalable Model Views on Heterogeneous Model Resources. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 334–344. ACM, New York, NY, USA (2018)
12. Bruneliere, H., Perez, J.G., Wimmer, M., Cabot, J.: EMF Views: A View Mechanism for Integrating Heterogeneous Models. In: International Conference on Conceptual Modeling (ER 2015), pp. 317–325. Springer (2015)
13. Burger, E., Henss, J., Küster, M., Kruse, S., Happe, L.: View-based Model-driven Software Development with ModelJoin. Software & Systems Modeling **15**(2), 473–496 (2016)
14. Chevalier, M., El Malki, M., Kopliku, A., Teste, O., Tournier, R.: How can we implement a multidimensional data warehouse using nosql? In: International Conference on Enterprise Information Systems, pp. 108–130. Springer (2015)
15. Cho, J., Garcia-Molina, H.: Synchronizing a Database to Improve Freshness. In: ACM SIGMOD Record, pp. 117–128. ACM (2000)
16. Daniel, G., Jouault, F., Sunyé, G., Cabot, J.: Gremlin-ATL: a scalable model transformation framework. In: Proceedings of the 32nd ASE Conference, pp. 462–472. IEEE (2017)
17. Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., Cabot, J.: NeoEMF: A Multi-database Model Persistence Framework for Very Large Models. Science of Computer Programming **149**, 9–14 (2017)
18. Daniel, G., Sunyé, G., Cabot, J.: Mogwaï: a framework to handle complex queries on large models. In: Proceedings of the 10th RCIS Conference, pp. 225–237. IEEE (2016)
19. Debreceni, C., Horváth, Á., Hegedüs, Á., Ujhelyi, Z., Ráth, I., Varró, D.: Query-driven Incremental Synchronization of View Models. In: Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, p. 31. ACM (2014)

20. Derler, P., Lee, E.A., Vincentelli, A.S.: Modeling Cyber–Physical Systems. Proceedings of the IEEE **100**(1), 13–28 (2012)
21. Eclipse Foundation: Connected Data Objects (CDO) (2020). URL `https://www.eclipse.org/cdo/`
22. ECMA International: JSON (JavaScript Object Notation) (2020). URL `https://www.json.org`
23. Eramo, R., de Kerchove, F.M., Colange, M., Tucci, M., Ouy, J., Bruneliere, H., Di Ruscio, D.: Model-driven Design-Runtime Interaction in Safety Critical System Development: an Experience Report. Journal of Object Technology **18**(2), 1–22 (2019)
24. Golra, F.R., Beugnard, A., Dagnat, F., Guérin, S., Guychard, C.: Addressing Modularity for Heterogeneous Multi-model Systems using Model Federation. In: Companion Proceedings of the 15th International Conference on Modularity, pp. 206–211. ACM (2016)
25. Gupta, H.: Selection of views to materialize in a data warehouse. In: International Conference on Database Theory, pp. 98–112. Springer (1997)
26. Hutchinson, J., Whittle, J., Rouncefield, M.: 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)
27. ISO/IEC/IEEE: Standard 42010:2011, Systems and Software Engineering - Architecture Description (2020). URL `https://www.iso.org/standard/50508.html`
28. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of computer programming **72**(1-2), 31–39 (2008)
29. Kolev, B., Valduriez, P., Bondiombouy, C., Jimenez-Peris, R., Pau, R., Pereira, J.: Cloudmdsql: querying heterogeneous cloud data stores with a common language. Distributed and parallel databases **34**(4), 463–503 (2016)
30. Kolovos, D.S., Paige, R.F., Polack, F.A.: Merging models with the epsilon merging language (eml). In: International Conference on Model Driven Engineering Languages and Systems, pp. 215–229. Springer (2006)
31. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M., 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, p. 2. ACM (2013)
32. Kolovos, D.S., Rose, L.M., Matragkas, N.D., Paige, R.F., Polack, F.A., Fernandes, K.J.: Constructing and navigating non-invasive model decorations. In: International Conference on Theory and Practice of Model Transformations (ICMT 2010), pp. 138–152. Springer (2010)
33. Kolovos, D.S., Wei, R., Barmpis, K.: An Approach for Efficient Querying of Large Relational Datasets with OCL based Languages. In: Proceedings of the Workshop on Extreme Modeling co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2013), Miami, Florida, USA, September 29, 2013., pp. 46–54 (2013). URL `http://ceur-ws.org/Vol-1089/6.pdf`
34. Langlois, B., Exertier, D., Zendagui, B.: Development of Modelling Frameworks and Viewpoints with Kitalpha. In: Proceedings of the 14th Workshop on Domain-Specific Modeling (DSM), co-located with SPLASH 2014, pp. 19–22. ACM (2014)
35. Madani, S., Kolovos, D., Paige, R.F.: Towards Optimisation of Model Queries: A Parallel Execution Approach. Journal of Object Technology - The 15th European Conference on Modelling Foundations and Applications (ECMFA'19) **18**(2), 3:1–21 (2019)
36. Marussy, K., Semeráth, O., Varró, D.: Incremental View Model Synchronization Using Partial Models. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), pp. 323–333. ACM (2018)
37. Object Management Group (OMG): Requirements Interchange Format (ReqIF) (2020). URL `https://www.omg.org/spec/ReqIF`
38. Object Management Group (OMG): Unified Modeling Language (UML) (2020). URL `http://www.uml.org`
39. OMG: MOF 2 XMI Mapping Specification version 2.5.1 (2020). URL `http://www.omg.org/spec/XMI/2.5.1/`

40. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: A Repository for Scalable Model Management. Software & Systems Modeling **14**(1), 219–239 (2015)
41. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'09), pp. 162–171. IEEE Computer Society (2009)
42. Semeráth, O., Debreceni, C., Horváth, Á., Varró, D.: Incremental Backward Change Propagation of View Models by Logic Solvers. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), pp. 306–316. ACM (2016)
43. Eclipse Sirius project (2020). URL `https://eclipse.org/sirius/`
44. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
45. The Eclipse Foundation: Teneo (2020). URL `https://wiki.eclipse.org/Teneo`
46. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: An Integrated Development Environment for Live Model Queries. Science of Computer Programming **98**, 80–99 (2015)
47. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework. Software & Systems Modeling **15**(3), 609–629 (2016)
48. Wei, R., Kolovos, D.S.: An Efficient Computation Strategy for allInstances(). Proceedings of the 3rd BigMDE Workshop pp. 32–41 (2015)
49. Whittle, J., Hutchinson, J., Rouncefield, M.: The State of Practice in Model-Driven Engineering. IEEE Software **31**(3), 79–85 (2014)
50. Willink, E.D.: Deterministic Lazy Mutable OCL Collections. In: Federation of International Conferences on Software Technologies: Applications and Foundations (STAF 2017), pp. 340–355. Springer (2017)