# Towards Scalable Model Views on Heterogeneous Model Resources

Hugo Bruneliere, Florent Marchand de Kerchove, Gwendal Daniel, Jordi Cabot

# Towards Scalable Model Views on Heterogeneous Model Resources

Hugo Bruneliere
IMT Atlantique - LS2N (CNRS) & ARMINES
Nantes, France
hugo.bruneliere@imt-atlantique.fr

Florent Marchand de Kerchove
IMT Atlantique - LS2N (CNRS) & ARMINES
Nantes, France
florent.marchand-de-kerchove@imt-atlantique.fr

Gwendal Daniel
IN3, Universitat Oberta de Catalunya (UOC)
Barcelona, Spain
gdaniel@uoc.edu

Jordi Cabot
ICREA & Universitat Oberta de Catalunya (UOC)
Barcelona, Spain
jordi.cabot@icrea.cat

## ABSTRACT

When engineering complex systems, models are used to represent various systems aspects. These models are often heterogeneous in terms of modeling language, provenance, number or scale. They can be notably managed by different persistence frameworks adapted to their nature. As a result, the information relevant to engineers is usually split into several interrelated models. To be useful in practice, these models need to be integrated together to provide global views over the system under study. Model view approaches have been proposed to tackle such an issue. They provide an unification mechanism to combine and query heterogeneous models in a transparent way. These views usually target specific engineering tasks such as system design, monitoring, evolution, etc. In our present context, the MegaM@Rt2 industrially-supported European initiative defines a set of large-scale use cases where model views can be beneficial for tracing runtime and design time data. However, existing model view solutions 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 sources. This paper presents the current status of our work towards a general solution to efficiently support scalable model views on heterogeneous model resources. It describes our integration approach between model view and model persistence frameworks. This notably implies the refinement of the view framework for the construction of large views from multiple model storage solutions. This also requires to study how parts of queries can be computed on the contributing models rather than on the view. Our solution has been benchmarked on a practical large-scale use case from the MegaM@Rt2 project, implementing a runtime – design time feedback loop. The corresponding EMF-based tooling support and modeling resources are fully available online.

*MODELS '18, October 14–19, 2018, Copenhagen, Denmark*

## CCS CONCEPTS

• **Software and its engineering** → Model-driven software engineering; Abstraction, modeling and modularity; • **Information systems** → Database design and models;

## KEYWORDS

Modeling, Views, Scalability, Persistence, Database, Design Time, Runtime

**ACM Reference Format:**
Hugo Bruneliere, Florent Marchand de Kerchove, Gwendal Daniel, and Jordi Cabot. 2018. Towards Scalable Model Views on Heterogeneous Model Resources. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239408

## 1 INTRODUCTION

Different, and quite often, many kinds of models are used when engineering complex systems. This is notably the case for systems of systems or Cyber-Physical Systems (CPSs) [13], 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 rely on various modeling languages, come from different sources or be very large. Moreover, these models usually contain complementary information that is not uniformly spread among them. Thus, engineers need to combine these 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.

The MegaM@Rt2 collaborative 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 ↔ design time feedback loop that could be deployed and used in different industrial domains. Such a feedback loop can bring interesting benefits in the above-mentioned engineering activities, for instance. To realize this in practice, model views are used to transparently relate together all the required (design and runtime) models.

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

Several model view approaches have already been proposed in order to provide support for such views [4]. Based on MDE, they allow specifying, creating and handling views on possibly heterogeneous models. Once built, the views can be used to uniformly manipulate and query the data coming from the different contributing models. However, most of the view approaches have only been deployed on models of small to medium size (e.g. manually created design models) and not on large to very large models (e.g. automatically-generated runtime models). Scalability is one of the main issues hampering the adoption of MDE in the industry [18], and is a major challenge to be addressed in MegaM@Rt2 as well. Moreover, the support for strongly interconnected models coming from different data sources (e.g. databases) appears to be the focus of only a few of the proposed approaches. Those are problems that are also particularly relevant in the context of MegaM@Rt2.

In this paper, we present the current status of our work towards efficiently supporting scalable model views over heterogeneous model resources. This includes the following contributions:

(1) A conceptual integration approach for the complementary use of model view and model persistence frameworks.
(2) A scalable realization of this approach combining the EMF Views [6] model view solution with the NeoEMF [10] and CDO [14] model persistence frameworks.
(3) An evaluation of our approach and current implementation on a large-scale and realistic scenario from the MegaM@Rt2 project.

The rest of this paper is organized as follows. Section 2 motivates our work by introducing the MegaM@Rt2 context, use case and related objectives. Section 3 presents our conceptual approach to integrate model view techniques and model persistence frameworks as a solution. Section 4 details how we implemented this conceptual approach on top of well-known MDE solutions based on the Eclipse Modeling Framework (EMF). Section 5 provide an evaluation of our approach and implementation via different scalability benchmarks performed on a MegaM@Rt2 use case. Finally, Section 6 reviews the related work while Section 7 concludes with our main lessons learned and planned future work.

## 2 BACKGROUND & MOTIVATION

The MegaM@Rt2 collaborative project[2] is a large and industrially-supported European initiative that officially started on April 2017. It relies on a wide consortium composed of 27 partners from 6 different national clusters (Sweden, France, Spain, Italy, Finland & Czech Republic) including large companies such as Atos, Thales, Nokia, Volvo and Bombardier. MegaM@Rt2 aims at incorporating methods and tools in order to develop a continuous system engineering and validation approach that can be practically deployed in various industrial domains [1]. To this intent, the project also comes with 9 case studies covering a variety of potential application areas: aeronautics, railway, warehouse, telecommunication, etc. One of its main expected contributions is notably to propose a runtime ↔ design time feedback loop that is (re)usable in these different contexts. In order to realize this, scalable model-based methods and tools are being considered to improve the productivity, quality and predictability of such large complex industrial systems. MegaM@Rt2 deliverables reporting the project's progresses have already been produced during the first year of the project and new ones are currently being developed [25].

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

### 2.1 MegaM@Rt2 Use Case

From the current MegaM@Rt2 results (notably the description of the industrial requirements and case studies), we have been able to extract a general scenario that concretely illustrates the need for scalable model views and their persistence. Thus, let us consider the realization of the previously mentioned runtime ↔ design time feedback loop via a view gathering 4 different models covering both runtime and design time aspects of a given system in the project. As shown on Figure 1, this view relies on a runtime log model (that conforms to a simple trace metamodel), a Java code model (that conforms to the Java metamodel from MoDisco [5]), a component model (that conforms to OMG UML [27]) and a requirement model (that conforms to OMG ReqIF [26]).

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

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

A concrete example of the view from Figure 1 is given in Figure 2. 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 (here a *trace.Log* element), one can move back to the originating Java code instructions (here *java.ClassDeclaration* elements). One can then follow links to the components (here *uml.Component* elements) the code implements, and up to the actual requirements these components fulfill (here *reqif10.SpecObject* elements).

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

To summarize, the main benefit of using a view is to collect in a transparent way information that is spread among different models. Without such a view, the engineer has to query the different models one by one and then aggregate the obtained results by herself. This includes recreating the mappings between related elements from different models in the view. Instead, using a view,

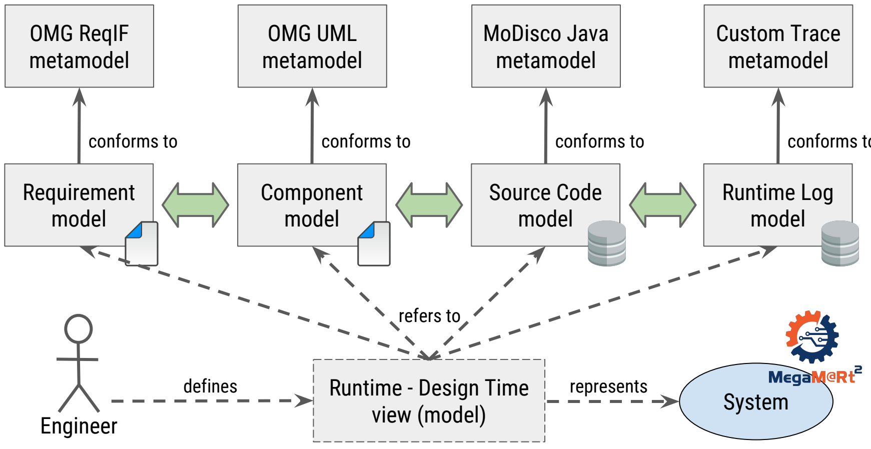
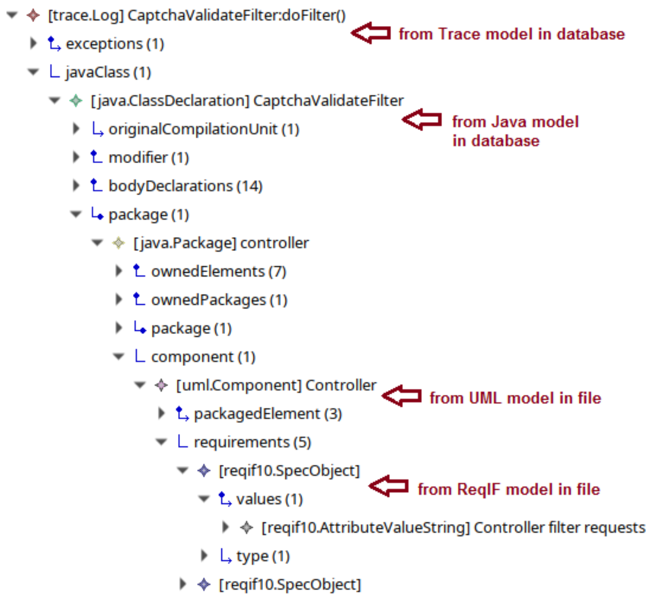


**Figure 1: Running use case from the MegaM@Rt2 industrial project.**



```
▾ [trace.Log] CaptchaValidateFilter:doFilter()
  ▸ exceptions (1)
  ▾ javaClass (1)
    ▾ [java.ClassDeclaration] CaptchaValidateFilter
      ▸ originalCompilationUnit (1)
      ▸ modifier (1)
      ▸ bodyDeclarations (14)
      ▾ package (1)
        ▾ [java.Package] controller
          ▸ ownedElements (7)
          ▸ ownedPackages (1)
          ▸ package (1)
          ▾ component (1)
            ▾ [uml.Component] Controller
              ▸ packagedElement (3)
              ▾ requirements (5)
                ▾ [reqif10.SpecObject]
                  ▾ values (1)
                    ▸ [reqif10.AttributeValueString] Controller filter requests
                  ▸ type (1)
                ▸ [reqif10.SpecObject]
```

**Figure 2: Concrete example of a view in MegaM@Rt2 (based on the use case from Figure 1).**

queries traversing several contributing models (such as the queries mentioned previously) can be expressed and computed naturally as if dealing with a single model.

### 2.2 Objectives

In the MegaM@Rt2 context, it is not sufficient to be able to build model views: the view mechanism must scale up when aggregating very large models provided by the industrial partners. More generally, the need for scalable modeling solutions has been observed in several industrial contexts [18, 33], and is also recognized as a long-term challenge from a research perspective [23]. However, existing model view solutions do not handle large and very large models well, if at all (cf. Section 6).

In the present paper, we target the building, handling and querying of scalable model views over heterogeneous model resources. We consider here heterogeneity in terms of both the used modeling languages (i.e. metamodels) and the underlying persistence solutions. We can notably leverage database model persistence frameworks that are particularly adapted to address scalability-related issues. Thus, we propose the following roadmap for our approach:

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

In Section 3, we describe our conceptual approach to achieve these goals. This approach has been implemented relying on the EMF ecosystem (cf. Section 4) and evaluated in practice on our MegaM@Rt2 use case (cf. Section 5). At the time of writing, we have successfully tackled (1), (2) and (3), and partially addressed (4).

## 3 INTEGRATION APPROACH

In what follows, we describe our general approach to support scalable model views on heterogeneous model sources. This section introduces, for each goal listed in Section 2.2, a conceptual solution to the related issue(s). The overall objective is to be able to build views that do scale up in practice: views that are built on top of several models where some are too large to be loaded, handled and stored only in memory (e.g. using the base EMF features, cf. Section 4). This is made possible notably by relying on models that can be persisted and manipulated, when necessary, using appropriate database backends. The general approach we consider is depicted in Figure 3.

A *Modeling Framework* is usually composed of two main parts: a *Core* component providing the inner behavior (i.e. model manipulation facilities) and a *Generic API* as the interface provided externally for (re)use by *Model-based Tools*. The *Modeling Framework* also often provides a base *File Persistence Framework* relying on the local

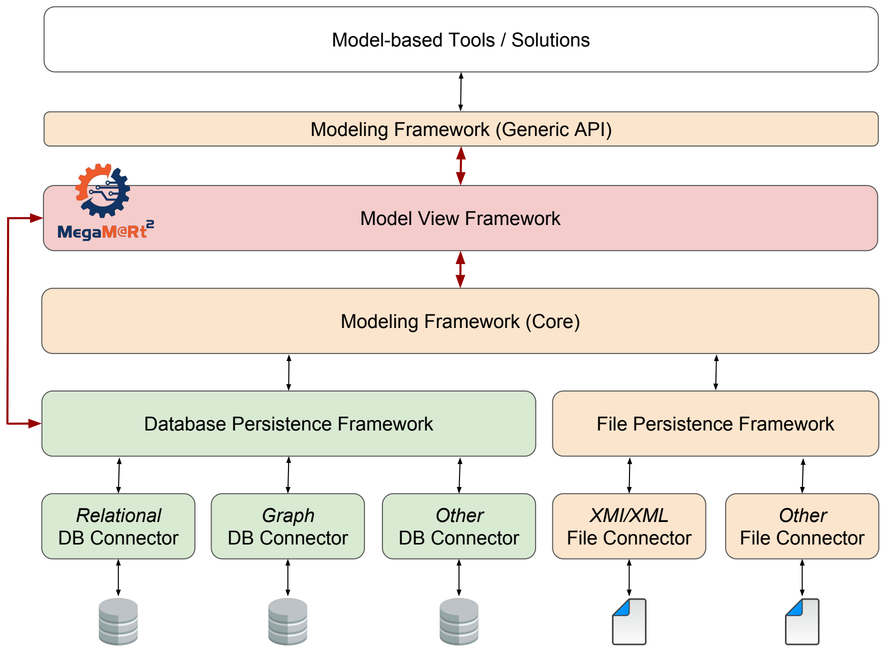

Model-based Tools / Solutions
Modeling Framework (Generic API)
MegaM@Rt2
Model View Framework
Modeling Framework (Core)
Database Persistence Framework
File Persistence Framework
Relational DB Connector
Graph DB Connector
Other DB Connector
XMI/XML File Connector
Other File Connector

**Figure 3: A conceptual approach for integrating model view and model persistence capabilities.**

file system, coming with some file import/export capabilities in different serialization formats. This default mechanism is used to store the design models introduced in our running example (cf. Section 2.1).

*Database Persistence Frameworks* have been proposed to connect the modeling framework to databases of various kinds (relational-based, graph-based, etc.). These solutions are typically used to store large models (e.g. the runtime models of our running example, cf. Section 2.1) with a reduced memory footprint.

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

## 3.1 Building Views on Heterogeneous Model Sources

This goal is of primary importance in MegaM@Rt2, and is a prerequisite to the three subsequent ones. As said earlier, most (if not all) modeling frameworks provide a default file serialization support, usually relying on XML-based format(s). However, they are very often not supporting other data sources. This is notably the case when needed to load/store models from/into different kinds of databases (e.g. relational, graph, etc.). Such databases can be existing ones, e.g. handled by external applications, or can be created just for the sake of modeling.

Model view approaches generally rely on the model persistence support provided by their underlying modeling framework. Thus, they usually lack of support for scalable model persistence solutions, e.g. relying on databases. As a consequence, it is required to perform the integration of the model view framework with such database persistence framework(s). This way, depending on the nature of the contributing models, different persistence backends can be selected and combined in the context of a same view. This is the case in our running scenario from Section 2.1, for example.

Such an integration can be performed in different ways. In some cases, it can be realized indirectly. The considered modeling framework can be first refined to be able to use the database persistence framework. Then, the view framework can simply rely on the general interface of the modeling framework in order to access transparently the underlying database resources. In some other situations, a direct connection can be required between the model view framework and the persistence framework. This notably allows implementing particular optimizations that could not be realized if passing by the modeling framework, cf. Section 4.1 for technical examples of these.

## 3.2 Persisting the View Information in a Scalable Way

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

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

Related scalability problems can occur when this view-specific data is too large to be handled correctly by the default support from the used modeling framework. Indeed, depending on the nature of the view, this extra data can be even larger than some of the contributing models themselves (e.g. as in our running scenario from Section 2.1). In these cases, it is required to be able to persist a view-specific model (storing this data) by using a more scalable database persistence framework. Adopting such a strategy can reduce significantly the memory footprint of given views, thus allowing to manipulate views that could not be loaded otherwise. Section 4.2 describes how this can be realized in practice.

## 3.3 Optimizing the View Loading and Element Access

With very large views, some operations can rapidly become heavy in terms of execution time and memory consumption. This can even go to a point where the view is not really usable anymore. For example, this is the case when the response time is too long (e.g. when the user navigates the view) or when the view simply ends by crashing. The situation is notably critical during the process of initializing/loading the view, as it can require a significant number of model element accesses.

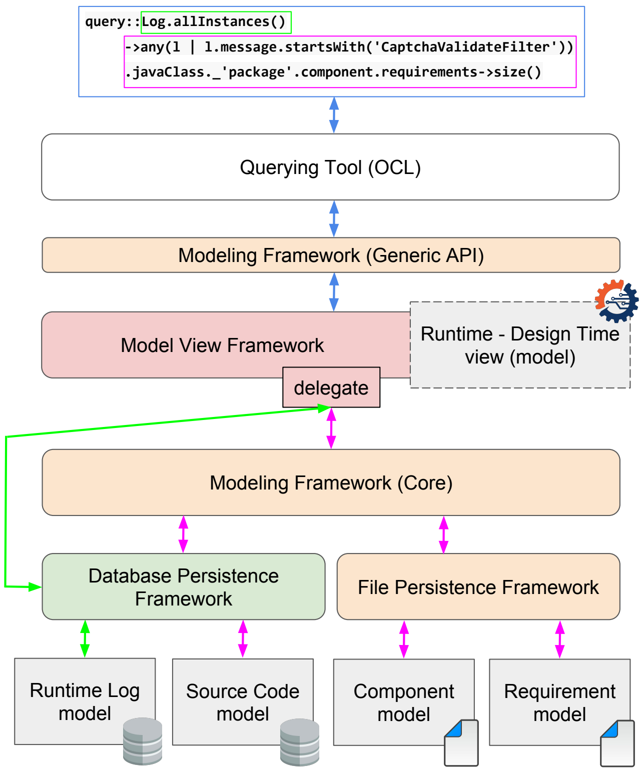
Relevant performance gains can be obtained by applying various lazy loading techniques at different levels. In the general case, any hit to an actual model element has to be delayed as much as possible and must only occurs when strictly needed. Such optimizations also concern accesses to both the various contributing models (cf. Section 3.1) and the view-specific elements (cf. Section 3.2). Ideally, all these accesses must be delayed without impacting the overall correctness of the view.

Moreover, depending on the used persistence framework(s), the model view framework can be refined differently. For given model element accesses, the view framework can directly benefit from specific capabilities provided by a database type (e.g. graph). For instance, the view framework can leverage the database API to turn full traversals of models into more selective requests, as traversals are time- and memory-intensive for large models. Section 4.3 gives more technical insights on how such optimizations can look like.

## 3.4 Optimizing the View Querying

Once a view has been correctly created and loaded (cf. Section 3.3), it can be navigated and queried as any regular model according to the needs of the engineering activity it supports. As said earlier, the view framework usually relies on a generic interface provided by a modeling framework and shared between different tools from a same ecosystem. This way, it also natively supports the execution of queries defined in languages supported by this modeling framework.

However, when implementing this in practice, performance issues can arise. For instance, some models can be serialized in standard XML-based files while others can be stored in databases (cf. Section 3.1). In this situation, using the default querying support might not take advantage of backend-specific optimizations. Thus, more elaborated schemes have to be considered.



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

Querying Tool (OCL)
Modeling Framework (Generic API)
Model View Framework
Runtime - Design Time view (model)
delegate
Modeling Framework (Core)
Database Persistence Framework
File Persistence Framework
Runtime Log model
Source Code model
Component model
Requirement model

**Figure 4: Optimizing model view querying by delegating to model persistence backends.**

The optimization of model querying techniques has already been studied (cf. Section 6), but not really in the context of model views. Base operations (e.g. `allInstances` in OCL) can be costly to execute with the default behavior of the modeling framework. For better efficiency, such operations could be delegated to the various persistence frameworks used in a view. This is illustrated in Figure 4 where an OCL query navigates the view from Section 2.1 and returns the number of design requirements that are impacted by a specified runtime event, captured as a log from an execution trace. This query can be optimized by delegating the `allInstances` call to the database persistence framework that contains the related elements, thus bypassing the default (less efficient) implementation of the modeling framework.

More generally, large performance gains are possible by splitting a query (on a given view) into a request plan that is better suited to the underlying persistence frameworks. This requires the model view framework to be able to split any query, delegate its execution and collect its results by leveraging the specificities of the different persistence backends. We have made first concrete steps towards this, as described in Section 4.4.

## 4 IMPLEMENTATION

We have concretely implemented the conceptual approach described in Section 3 by relying on EMF [28] as our core Modeling Framework. Based on our own knowledge and expertise, we made the choice of using EMF Views [6] as our Model View Framework. Concerning the Persistence Frameworks, we used both NeoEMF [10] and CDO [14] as supporting graph and relational database backends (respectively). In what follows, we detail how all these technical solutions have been combined together in practice to address the four different key points of our conceptual approach. Our current implementation is freely available from a branch of the EMF Views source code repository[3].

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

*4.1.1 EMF Views for Model Views.* EMF Views[4] is a model view mechanism that reuses the well-known concept of view in databases and transpose it to the modeling world. It embeds a lightweight model virtualization framework that can be used on top of any EMF-based model. This way, it allows to create (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. The approach has the following characteristics:

- Lightweight: elements in a view are only proxies to actual elements from the contributing models (which are never copied). This allows for a low overhead when creating and navigating views.
- Filtering: existing elements, attributes or references from the contributing models can be hidden in a view.
- Virtualization: views can contain new elements, attributes or references that exist only at the view level and are not part of the contributing models.
- Non-intrusiveness: all the additional view-specific information (pointers to contributing models, filters, virtual elements, references between different models) is described in a separate *weaving model* [5] and does not imply any change on the contributing models.

EMF Views is released as a set of open source Eclipse plugins. Its native integration with EMF-based tools as well as its characteristics constitute an interesting starting point for our implementation. Note that alternative view mechanisms are also considered in the related work (cf. Section 6).

*4.1.2 CDO and NeoEMF for Model Persistence.* The Connected Data Objects model repository (CDO) [14] 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. CDO's default implementation uses a relational database connector to serialize models into SQL compatible databases. However, the modular architecture of the frameworks can be extended to support different data storage solutions (even if, in practice, only relational connectors are used and regularly maintained).

NeoEMF[6] is a complementary model persistence framework that relies on the scalable nature of NoSQL databases to store and manipulate large models. NeoEMF supports three model-to-database mappings, i.e. graph, key-value and column stores. Each one of them is adapted to a specific modeling scenario, such as atomic element accesses (key-value) or complex navigations (graph). As other persistence solutions, NeoEMF provides a *lazy-loading* mechanism that allows to obtain significant gains in terms of performances.

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

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

### 4.2 Persisting the View Information with NeoEMF

As explained before, EMF Views uses a *weaving model* that represent the view-specific information. This model can potentially contain entries for many elements coming from the different contributing models. Thus, it can get as large or even larger (depending on the view) than the contributing models themselves. In order to improve the scalability of our approach on large-scale views (such as in our running example from MegaM@Rt2, cf. Section 2), we chose to persist this model using NeoEMF instead of using the default XMI serialization.

Since the weaving model is also defined as a standard EMF model, its migration to NeoEMF has been done quite transparently by changing the model serialization behavior (and initializing the corresponding database backend). Persisting the weaving model in NeoEMF allows us to handle views that cannot fit in memory otherwise.

### 4.3 Optimizing the View Loading and Element Access in EMF Views

When dealing with large database resources, many operations of the EMF API that had little to no overhead with small in-memory resources now potentially bear high costs in execution time and memory consumption. So we had to pay extra attention to minimize the impact of such operations. For instance, checking whether a

[3]https://github.com/atlanmod/emfviews/tree/integrate-neoemf
[4]https://github.com/atlanmod/emfviews/
[5]We do not elaborate in this paper how such weaving model can be obtained in general (the initial EMF Views paper [6] covers this point), but we describe one construction scheme in Section 5.2.
[6]https://github.com/atlanmod/neoemf/

reference has any contents can be done by calling the *EList.isEmpty* operation. A naive implementation of this operation compares the size of the collection against zero, where getting the size is an $O(n)$ operation. On small in-memory resources, $n$ is small and a call to the *isEmpty* operation triggers no issue. On large database resources, $n$ is large and the overhead of hitting the database can become a bottleneck. A better implementation of *isEmpty* rather checks if at least one element exists, and thus exits early when this is not the case. Similarly, getting the $n$th 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 less hits to the database and the operation has a minimal cost.

We significantly improved EMF Views for large model resources by following these ideas. As introduced in Section 3.3, our key tenet was: delay actual hits to the resource as much as possible.

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

A third point of optimization was to tweak the way the data is stored into the graph database handled by NeoEMF. The runtime log model of our example (cf. Section 2.1) 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 performances of NeoEMF. To solve this issue, we developed a new mapping from model to graph for NeoEMF, using in-database linked lists. This mapping, dedicated to large collections, allowed us to speed up the creation of the runtime log model and the access to its elements by a factor of 30.

## 4.4 Optimizing the View Querying in OCL

Since views are regular EMF models, querying tools like OCL or transformation tools like ATL can be applied transparently on views (regardless of the underlying persistence framework used by the contributing models). However, relying only on the EMF API leaves a lot of performance on the table. Base operations, like `allInstances` in OCL, can be quite costly to execute naively using the EMF API [32]. On the contrary, persistence backends may provide more efficient ways to execute such operations. For example, NeoEMF resources expose a `getAllInstances` method which can compute the set of instances of a given classifier. This method is around 40 times faster than using the EMF API directly.

We extended the standard OCL interpreter in order to specialize some operations according to the data store they target. In the following paragraph we detail our implementation of the `allInstances` operation, but other native operation implementations can be easily defined to enhance query computation performances. However, our implementation still needs a generic operation delegation mechanism that, along with the support for other query languages, is currently left for future work (cf. Section 7).

The OCL API provides a way to customize the behavior of the `allInstances` operation through a *Model Manager* (or *Extents Map* in the legacy implementation). We define a custom extents map that allows to specialize the `allInstances` call according to the concrete data stores used in a given view. When instances of a classifier are looked up, the extents map redirects the call to the view that fetches instances—using native database calls—from each contributing model and them combines these instances as the result. Compared to the standard OCL implementation that iterates the entire model to match elements of a given type, this approach benefits from the low-level optimizations of the databases (such as built-in indexes and caches). We show in our evaluation that such optimization can dramatically improve query computation performances (cf. Section 5).

## 5 EVALUATION

To evaluate our integration approach and its current implementation, we applied them on the use case from the MegaM@Rt2 project (cf. Section 2.1). In this section, we focus on measuring the *time overhead* of our current implementation, because it directly impacts the interactive user experience (as opposed to batch processing). However, as dealing with on-disk resources is inevitably (one to two orders of magnitude) slower than dealing with in-memory resources, matching the speed of in-memory resources is not a realistic goal. Thus, we rather insist on the asymptotic behavior of our approach and on gains made by our optimizations.

For reproducibility, the complete source code of the performed benchmarks (including the models and views) as well as more detailed results are available online[7][8]. All the benchmarks have been realized on a laptop with an i7-7600 (2.80GHz) processor, 32GB of RAM, and M.2 PCIe SSD, using OpenJDK 64-bit 1.8.0.

### 5.1 Overview

We built two versions of the same view answering to the MegaM@Rt2 use case.

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

The second version demonstrates our capability to build views over heterogeneous model resources. It uses a mix of file-based and database resources as contributing models. More precisely, the Runtime Log model and the weaving model are persisted in a Neo4j graph database handled by NeoEMF, using our mapping developed for optimizing flat models (cf. Section 4.3). The Java Source Code model is persisted in a relational database handled by CDO. Only

[7]https://github.com/atlanmod/scalable-views-heterogeneous-models
[8]http://remodd.org/content/towards-scalable-views-heterogeneous-model-resources

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

**Table 1: Time (in minutes) to create the view-specific models (weaving models).**

the two remaining models, namely the UML Component model and the ReqIF Requirement model, are serialized as XMI files handled by the standard EMF implementation.

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

## 5.2 Benchmark 1: Creating the View-specific Information

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

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

As a consequence, creating the weaving model implies checking different matches between two elements coming from two contributing models. For large models, such as the Runtime Log from the MegaM@Rt2 scenario, these matches can be very numerous. In these cases, the whole matching process can take a significant amount of time. Table 1 compares the time it takes to create the weaving model using the two versions of our view, while Table 2 compares the sizes of the persisted weaving models on disk. The *Size* column in both tables refers to the number of log elements in the Runtime Log model.

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

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

**Table 2: Size (in megabytes) of the view-specific models (weaving models) on disk.**

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

**Table 3: Time (in seconds) to load the view.**

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

**Table 4: Time (in seconds) to iterate over the full content of the view.**

## 5.3 Benchmark 2: Loading the View

In the second benchmark, we evaluated both the loading of a view and the iteration over all its contents. Again, we performed this on the two versions of the view (full XMI vs. databases + XMI). This benchmark measures the overhead of accessing the content of the different models contributing to the view. Table 3 compares the time it takes to load the two versions of the view, while Table 4 compares the time required to iterate over the full content of the view.

A first point is that loading the heterogeneous view takes a relatively low and constant time (between 1 and 4 seconds), regardless of the size of the Runtime Log model (i.e. the largest one) contributing to the view. For the largest model size, it takes four times longer to load the first view compared to the similar heterogeneous view which uses our approach. This difference can be explained by the lazy loading of our approach, where most of the actual loading takes place when navigating model elements.

When iterating over the full content of the view, the overhead remains relatively small, but slightly increases as the Runtime Log model gets larger. For the largest size, the heterogeneous view is

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

**Table 5: Time (in seconds) to run the OCL query (1).**

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

**Table 6: Time (in seconds) to run the OCL query (2).**

42 times slower to navigate, which is around the expected speed difference between RAM and disk. This large increase in time can be partly explained by the data model of the underlying database, for which exhaustive iteration is a very costly operation due to numerous loads/unloads between database and memory. While a full iteration scenario may not be very common in practice, it is a useful reminder that the choice of data representation can have a strong impact on performance.

## 5.4 Benchmark 3: Querying the View

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

```
(1) Log.allInstances()->size()
(2) Log.allInstances()
    ->any(l| l.message.startsWith('CaptchaValidateFilter'))
    .javaClass._'package'.component.requirements
    ->size()
(3) SpecObject.allInstances()
    ->any(r| r.values->selectByType(AttributeValueString)
          ->exists(v| v.theValue.startsWith('Controller')))
     .components->collect(c| c.javaPackages)
    ->collect(p| p.ownedElements)
    ->selectByType(ClassDeclaration)
    ->collect(c| c.traces)
    ->size()
```

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

Table 5 compares the time it takes to execute query (1) on the two versions of our view. Tables 6 and 7 do the same for queries (2) and (3), respectively. In these three tables, the two additional *(Opt.)* columns refer to optimized views that use the custom extents map we described in Section 4.4.

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

**Table 7: Time (in seconds) to run the OCL query (3).**

One observation is that, on the heterogeneous view, the queries can be 11 to 30 times slower than on the XMI view, which is still lower than the expected speed difference between RAM and disk. The optimized versions are a 2- to 6-fold improvement, which brings the overhead down to 3 to 30 times slower, as the optimizations also benefit the XMI view. The effect of the specialization of the *allInstances* operation on the Runtime Log model stored in database is the most evident on the last line of Table 5. For the other two queries, the improvement is lower, but still significant. Some further gains may lie in fully specializing the queries into backend-specific request plans, as proposed in Section 3.4.

## 6 RELATED WORK

Several solutions have already been proposed in order to support the definition, creation and handling of views over models. We have already studied and tried to classify them according to their main characteristics [4]. From what we have been able to observe so far, there are currently not many proofs that these solutions do scale up in the context of very large models. This can be considered as an important issue in this area, as scalability is key to related challenges such as view update or incremental maintenance (such as seen in our detailed study).

Moreover, few model view approaches focus on integrating strongly interconnected models coming from different data sources. For instance, ModelJoin [7] proposes a DSL for querying different models and building a view as a result. However, it does not currently provide particular support for various model persistence backends (other than the EMF default ones). On the contrary, OpenFlexo [16] is natively intended to support several data sources. But it privileges graphical (manual) modeling to specify views and does not come with an extended querying support at view-level. These are important aspects that were lacking in our current MegaM@Rt2 context.

Another relevant approach is Epsilon [21], a modeling platform defining a family of languages to query, transform and compare models. Among these languages, Epsilon provides EML [22], that allows to merge models, and Epsilon Decoration [24], that allows to decorate an existing model with additional features. These can be seen as kind of equivalent to a (partial) support for model views. Compared to our solution based on EMF Views [6], Epsilon does not currently provide optimized strategies to heterogeneous backend queries, nor allows to decorate multiple models/metamodels. Moreover, it only relies on in-memory structures to store both the merging and decoration constructs, which can be a limitation in terms of scalability.

There has also been several initiatives more related to the model querying domain. For example the VIATRA [31] project, relying on IncQuery [30] for efficient incremental querying, is a reactive model transformation platform. The platform also embeds some capabilities for manipulating multiple models and defining sorts of views over them [12]. 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 solution is rather designed to access models from different sources (notably from databases), and to benefit from their internal structure in order to efficiently perform single query computations. However, incremental querying could be integrated to our solution by experimenting on the use of IncQuery over our views (in addition to OCL as described in this paper).

A few approaches have been proposed to improve the computation of model-level queries over several scalable persistence solutions. The Mogwaï tool [11] is a translation approach that maps OCL constructs to Gremlin [29], a graph database language: generated queries are sent directly to the database for computation, thus bypassing the limitation of current modeling frameworks. A similar approach is used in the Hawk query framework [2], that dynamically translates Epsilon queries into graph database native calls. However, these solutions focus on specific data sources and are not designed to handle heterogeneous backends (such as in a view). Nevertheless, they could be integrated to our solution in order to speed-up parts of queries related to the backend they target.

Finally, in the data warehouse community, views on heterogeneous storage solutions have already been studied for relational databases [17]. They regained interest quite recently thanks to polystore data warehouses [8]. On the query side, CloudMdsQL [20] is a SQL-like language for querying multiple data stores within a single query. To do so, it extends the standard SQL syntax with additional constructs allowing to embed native datastore queries. A similar approach can be found in generic query frameworks such as Apache Drill[9]. Such solutions could also be integrated to our approach to improve the overall computation of queries and views on models stored in heterogeneous data backends.

## 7 CONCLUSION: LESSONS LEARNED & FUTURE WORK

In this paper, we presented our approach to support the efficient creation and handling of model views over heterogeneous large-scale models. We started by detailing our conceptual solution to integrate model views and model persistence frameworks, emphasizing on important goals to be addressed. Then we described our current EMF-based implementation of this conceptual approach, that is being developed in the context of the MegaM@Rt2 industrial project. We also summarized our evaluation of this approach and implementation onto a view scenario taken from MegaM@Rt2. In the current state, our work has shown promising results when creating and loading views on very large models. Our evaluation has also shown that there is still room for improvement when querying large-scale views using our approach, even though we already started to address some frequently encountered cases. We believe that, by continuing to leverage model persistence-specific optimizations, we can achieve scalability at a low overhead with our solution.

[9]https://drill.apache.org/

*Lessons Learned.* From our experience in this paper, we can attest to the benefit 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, but these methods are not available to EMF Views when only relying on the standard EMF API. Secondly, using such 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. Ultimately, to minimize the overhead of views on large models, we have to consider more elaborated strategies. We can notably identify when it is relevant to delegate a given operation or not, and to which persistence backend. Thanks to our approach, we have been able to embed such practical knowledge into our integrated model view solution. This way, users do not have to pay the full cost of the generic API while still benefiting from its genericity and interoperability capabilities.

*Future Work.* In the next steps of our work, we will push further our experiments by also testing model transformation tools on our views, such as ATL [19] or VIATRA [31] ones. To this intent, we can use the view information to delegate parts of the transformation computations directly to the underlying database backends. We plan to do this by integrating scalable query/transformation approaches, such as Mogwaï [9] for instance. Another possible work is to extend our integration approach to additional model handling and querying environments, such as Epsilon [21] or KMF [15] for instance. Moreover, we could also study how incremental querying techniques [3] could be integrated in our approach (in addition to the reuse of some already realized optimizations, at OCL-level for instance [34]). Finally, we are going to continue the developments around our model view solution and apply them in the context of other use cases from the MegaM@Rt2 project. For example, 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.

## ACKNOWLEDGMENTS

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.

## REFERENCES

[1] Wasif Afzal, Hugo Bruneliere, Davide Di Ruscio, Andrey Sadovykh, Silvia Mazzini, Eric Cariou, Dragos Truscan, Jordi Cabot, Abel Gomez, Jesus Gorronogoitia, Luigi Pomante, and Pavel Smrz. 2018. 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 (2018), 86 – 95.

[2] Konstantinos Barmpis and Dimitris Kolovos. 2013. Hawk: Towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*. ACM, 6.

[3] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. 2010. Incremental Evaluation of Model Queries over EMF models. In *International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*. Springer, 76–90.

[4] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2017. A Feature-based Survey of Model View Approaches. *Software & Systems Modeling* (2017), 1–22.

[5] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. MoDisco: A Model Driven Reverse Engineering Framework. *Information and Software Technology* 56, 8 (2014), 1012–1032.

[6] Hugo Bruneliere, Jokin Garcia Perez, Manuel Wimmer, and Jordi Cabot. 2015. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In *International Conference on Conceptual Modeling (ER 2015)*. Springer, 317–325.

[7] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. 2016. View-based Model-driven Software Development with ModelJoin. *Software & Systems Modeling* 15, 2 (2016), 473–496.

[8] Max Chevalier, Mohammed El Malki, Arlind Kopliku, Olivier Teste, and Ronan Tournier. 2015. How can we implement a Multidimensional Data Warehouse using NoSQL?. In *International Conference on Enterprise Information Systems*. Springer, 108–130.

[9] Gwendal Daniel, Frédéric Jouault, Gerson Sunyé, and Jordi Cabot. 2017. Gremlin-ATL: a scalable model transformation framework. In *Proceedings of the 32nd ASE Conference*. IEEE, 462–472.

[10] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. 2017. NeoEMF: A Multi-database Model Persistence Framework for Very Large Models. *Science of Computer Programming* 149 (2017), 9–14.

[11] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. 2016. Mogwaï: a Framework to Handle Complex Queries on Large Models. In *Proceedings of the 10th RCIS Conference*. IEEE, 225–237.

[12] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. 2014. Query-driven incremental synchronization of view models. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, 31.

[13] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. 2012. Modeling Cyber–Physical Systems. *Proc. IEEE* 100, 1 (2012), 13–28.

[14] Eclipse Foundation. 2018. Connected Data Objects (CDO). (2018). https://www.eclipse.org/cdo/ URL: https://www.eclipse.org/cdo/.

[15] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. 2012. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *Proceedings of the 15th MoDELS Conference*. Springer, 87–101.

[16] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, and Christophe Guychard. 2016. Addressing Modularity for Heterogeneous Multi-model Systems using Model Federation. In *Companion Proceedings of the 15th International Conference on Modularity*. ACM, 206–211.

[17] Himanshu Gupta. 1997. Selection of views to materialize in a data warehouse. In *International Conference on Database Theory*. Springer, 98–112.

[18] John Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors that Lead to Success or Failure. *Science of Computer Programming* 89 (2014), 144–161.

[19] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A Model Transformation Tool. *Science of computer programming* 72, 1-2 (2008), 31–39.

[20] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jimenez-Peris, Raquel Pau, and José Pereira. 2016. CloudMdsQL: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases* 34, 4 (2016), 463–503.

[21] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2006. Eclipse development tools for epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, Vol. 20062. 200.

[22] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2006. Merging models with the epsilon merging language (EML). In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 215–229.

[23] Dimitrios S Kolovos, Louis M Rose, Nicholas Matragkas, Richard F Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, et al. 2013. 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*. ACM, 2.

[24] Dimitrios S Kolovos, Louis M Rose, Nikolaos Drivalos Matragkas, Richard F Paige, Fiona AC Polack, and Kiran J Fernandes. 2010. Constructing and navigating non-invasive model decorations. In *International Conference on Theory and Practice of Model Transformations*. Springer, 138–152.

[25] MegaM@Rt2 Project (ECSEL - H2020). 2018. Project deliverables. (2018). https://megamart2-ecsel.eu/deliverables/

[26] Object Management Group (OMG). 2018. Requirements Interchange Format (ReqIF). (2018). https://www.omg.org/spec/ReqIF

[27] Object Management Group (OMG). 2018. Unified Modeling Language (UML). (2018). http://www.uml.org

[28] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework*. Pearson Education.

[29] Tinkerpop. 2018. The Gremlin Language. (2018). www.gremlin.tinkerpop.com URL: www.gremlin.tinkerpop.com.

[30] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An Integrated Development Environment for Live Model Queries. *Science of Computer Programming* 98 (2015), 80–99.

[31] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework. *Software & Systems Modeling* 15, 3 (2016), 609–629.

[32] Ran Wei and Dimitrios S Kolovos. 2015. An Efficient Computation Strategy for allInstances(). *Proceedings of the 3rd BigMDE Workshop* (2015), 32–41.

[33] Jon Whittle, John Hutchinson, and Mark Rouncefield. 2014. The State of Practice in Model-Driven Engineering. *IEEE Software* 31, 3 (2014), 79–85.

[34] Edward D Willink. 2017. Deterministic Lazy Mutable OCL Collections. In *Federation of International Conferences on Software Technologies: Applications and Foundations (STAF 2017)*. Springer, 340–355.