Ref. Ares(2017)4761131 - 29/09/2017

WP4 - D4.1: Foundations for Model Management and Traceability

Due to project's M6
Final editing by Hugo Bruneliere (ARM) on 29/09/2017
With contributions from ARM, UOC, SOFT, UAQ, MDH, INT, VTT, SICS
Reviewed by NOK and MDH
Handle: *20.500.12004/1/P/MMART2/D4.1*

Executive summary

The MegaM@Rt2 approach and corresponding technical solutions heavily rely on the use of models. These models can have various and varied natures (e.g. design models defined manually by engineers, runtime models generated from system executions) as well as sizes (e.g. from relatively small design models to very large runtime ones). In addition, it is necessary to be able to properly relate and/or trace all these models together from their construction to their actual use in/by the modeled system. This is particularly true in MegaM@Rt2 that intends to provide support for the continuous development and validation of software systems via a feedback loop from runtime to design time. Thus, the main goal of WP4 is to elaborate on the required glue between the artifacts produced in WP2 (e.g. design models) and the ones produced in WP3 (e.g. runtime models). As a result, it is expected to provide a so-called global MegaM@Rt Model Management and Traceability framework to be a core part of the MegaM@Rt2 overall solution and to be notably deployed on the project's use cases (among possibly others). In this context, identifying, (re)using or eventually extending already existing model management and traceability principles and techniques appear to be fundamental.

As the initial step in WP4, the present deliverable provides an overall state-of-the-art in terms of existing model management and traceability solutions. It presents the main common principles and approaches related to model storage, querying, handling and linking with others models and modeling artifacts, notably via model views and/or so-called megamodels. It also insists on the available traceability and interoperability solutions. It describes both existing research approaches as well as some more business-oriented tools or environments which are relevant in this given context. Finally, it ends with a list of technical solutions provided by the project's partners and relevant in the context of the present document/topics. All along the deliverable, a particular importance is given to aspects related to the scalability of the available solutions.

The main purpose of this deliverable is to prepare the work for specifying the Model Management & Traceability framework to be developed and further used in MegaM@Rt2 (cf. the coming deliverable D4.2 which is due at M12). Its goal is also to help identifying some of the key problems to be addressed while implementing this framework in the future (cf. deliverables D4.3 and D4.4 due at M20 and M32 respectively). Among others, the following big challenges can be mentioned: scalable model storage and querying, well-synchronized and verified model views, performant and decentralized global model management, efficient integration of inter-model traceability and interoperability support.

Table of Contents

Acronyms

DSL	Domain-Specific Language
DSML	Domain-Specific Modeling Language
EA	Enterprise Architecture
EMF	Eclipse Modeling Framework
MDE	Model Driven Engineering
MDRE	Model Driven Reverse Engineering
OMG	Object Management Group
QVT	Query/View/Transformation (from the OMG)
RDBMS	Relational Database Management System
TOGAF	The Open Group Architecture Framework
UML	Unified Modeling Language (from the OMG)
XMI	XML Metadata Interchange (from the OMG)
XML	eXtensible Markup Language (from the W3C)

1. Introduction

An essential element of the MegaM@Rt2 project is to provide a scalable support for managing the traceability between all layers of the system design and execution, from high-level engineering practices to low-level logs. This is fundamental for enabling the continuous development and validation approach targeted by MegaM@Rt2. To this intent, WP4 has been conceived with the goal to elaborate on the required glue between the artifacts produced in WP2 (e.g. design models) and the ones produced in WP3 (e.g. runtime models). As a result, it is expected to provide a global MegaM@Rt Model Management and Traceability framework to be deployed and experimented on the project's use cases whenever relevant.

Before starting working on the actual design (and then implementation) of such a common global framework, we first need to study attentively the current state-of-the-art related to the corresponding scientific and technical aspects to be possibly considered, reused, extended and/or refined in the context of the project. Some of the considered research approaches and existing (commercial and/or open source) solutions for megamodelling / model management & traceability support are being developed by MegaM@Rt2 partners, while others are coming from external organizations. In addition to listing and commenting on all these already existing solutions, it is also important to immediately identify current main issues as well as still open challenges we could face during the project. This is the main objective of T4.1, and of the present deliverable D4.1 that reports on this particular effort during the first 6 months of the project.

As introduced right above, there are different aspects to be studied as far as model management and traceability is concerned. Firstly, models must be stored, retrieved and queried as efficiently as possible. The Modeling research community is rather active on this topic since some years already. The current status of its main realizations is presented in Section 2. Secondly, the significant number and variety of models to be handled in our context implies to be able to correctly combine or aggregate them. The objective is to provide support for building usable/suitable views over such heterogeneous models. Recent progresses towards these important capabilities are described in Section 3. Then, complementary to the two previous items, approaches for globally representing and handling the cartography of all the involved models, views, etc. are needed. While, both the Modeling research community and companies have been working on this topic in the past years, there are still open issues to be solved. The current situation is depicted in Section 4. Finally, as a key aspect of MegaM@Rt2, a specific focus should be put on principles and techniques directly addressing the traceability and interoperability between models (as well as between related metamodels / modeling languages). The relevant families of approaches are presented in Section 5.

This deliverable also provides a dedicated discussion section, as Section 6, intending to offer a global integrated vision over the different related work abovementioned. The document ends with a short conclusion (cf. Section 7) and an Appendix listing the tools/prototypes already developed by the MegaM@Rt partners and that might be enhanced and employed in the context of WP4.

2. Model Storage and Querying

File-based XML serialization has been the preferred format for storing and sharing models and metamodels, especially since the publication of standards like XMI[1]. This choice was driven by the fact that modeling frameworks were originally designed to handle human-produced models, whose size does not cause significant performance concerns. However, the adoption of MDE practices in the industry (Whittle et. al. 2014) as well as the development of generative framework such as Model Driven Reverse Engineering - MDRE approaches (Bruneliere et. al. 2014) has popularized the need to handle large and complex (potentially generated) models, emphasizing XMI's limitations.

Indeed, 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 a navigational model of their contents. The former reduces efficiency of I/O accesses, while the later increases the memory usage (i.e., the memory required to load and query models), and limits the use of proxies and partial loading to inter-document relationships. In addition, XMI persistence layers do not provide advanced features such as transactions or collaborative edition, and large monolithic model files are challenging to integrate in existing versioning systems (Barmpis et. al. 2013).

As a result, scalability of model persistence framework has been an active field of research in the last decade (Kolovos et. al. 2013), and several approaches have been proposed to reduce their memory usage and enable support for very large models. They can be classified into two categories based on their low-level model representations: (i) RDBMS-based solutions that store models in relational tables, and (ii) NoSQL solutions that uses semi-structured databases (such as graph databases or document-oriented stores). Existing approaches usually expose an interface that is semi-compliant with the de-facto standard modeling APIs, and provide a lazy-loading mechanism, which reduces the memory consumption by loading model elements from the data-store only when they are accessed.

In the traditional modeling environments, model queries and operations are typically computed by dedicated solutions that rely on the low-level model handling API. As an example, the Eclipse MDT OCL framework[2] is the standard solution to define and evaluate OCL expressions in the Eclipse Modeling Framework (EMF). The MDT OCL framework translates high-level OCL operations into low-level EMF API calls that are finally executed by the database. While these approaches are efficient for small models stored in memory, they have shown clear limitations when applied to larger models (Daniel et. al. 2016). Alternative query approaches have been proposed to tackle this issue by taking benefit of the database query facilities to improve query computation performances and memory footprint. For example, the Model Query Translator (De Carlos et. al. 2015) and Mogwaï frameworks (Daniel et. al. 2016) are two solutions that aim to generate database-specific queries from model operations expressed using high-level modeling languages such as OCL and EOL. Alternatively, some persistence solutions such as CDO also provide a raw access to their database to allow computation of native database queries.

[1] http://www.omg.org/spec/XMI/2.5.1/
[2] https://projects.eclipse.org/projects/modeling.mdt.ocl

2.1. Relational Persistence Layers

Historically, RDBMS have been the preferred solution to store large models (e.g. CDO, see below). Existing approaches derive a relational schema from an existing metamodel, for example by creating tables to store the instances of each metamodel's classes and columns for every class attribute (De Carlos et. al. 2015). This schema is then used to store model elements, access attributes, or navigate associations using low-level query languages such as SQL (De Carlos et. al. 2014). Existing frameworks implements the de-facto standard EMF API, and can be transparently integrated (once configured) into existing modeling applications to enhance their scalability.

The Connected Data Object model repository (CDO)[3] was the first attempt designed to handle large models by relying on a client-server repository structure. A CDO application can connect to a CDO server using a specialized interface, and a dedicated implementation of the EMF API is provided to manipulate the model. CDO is based on a lazy-loading mechanism and supports transactions, access control policies, and provides a collaborative modeling environment allowing concurrent editing of a model. CDO's default implementation uses a relational database connector to serialize models into SQL compatible databases, but the modular architecture of the frameworks can be extended to support different data storage solutions. However, in practice only relational connectors are used and regularly maintained.

Teneo[4] is another approach based on relational databases to store EMF models. It relies on a dedicated mapping that allows to store EMF models using the Hibernate ORM (Bauer et. al. 2005). Teneo uses metamodel information to derive a relational schema and an EMF-compatible API that allows to access the model at a high level of abstraction. The Hibernate implementation provides an additional API to express model queries using the HQL query language, improving performance by lowering the level of abstraction. Teneo is embedded in the default Hibernate connector provided by CDO.

While these solutions have proven their efficiency w.r.t XMI-based implementations, the highly interconnected nature of models often requires multiple table join operations to compute complex model queries, limiting the performance both in terms of execution time and memory consumption (Barmpis et. al. 2012). In addition, the strict schema used in RDBMS makes them hard to align with metamodel updates which can define new types, associations, etc. Finally, the extraction of the relational schema from an existing metamodel requires to integrate platform-specific initialization code, that can be a limiting factor for the adoption of the solution into existing applications.

2.2. NoSQL Persistence Layers

NoSQL-based solutions have been proposed to tackle the limitations of relational databases to handle large models. The proposed approaches are based on the schema-less nature of NoSQL data-stores to handle metamodel modifications efficiently, and rely on the database's query performance to compute complex model navigations efficiently.

Morsa (Pagan et. al. 2015) is the first approach designed to take benefit of the scalability features provided by NoSQL document databases to store and access large models in an efficient way. As CDO, Morsa relies on a lazy-loading mechanism to limit memory consumption, and supports

[3] http://www.eclipse.org/cdo/
[4] https://wiki.eclipse.org/Teneo

incremental updates. The framework is based on MongoDB, and uses the document hierarchy capabilities of the data-store to represent model elements and their associations. Morsa models can be created and accessed transparently using the standard EMF mechanisms. However, model queries have to be expressed using a dedicated query language (MorsaQL) to fully benefit from the underlying data-store performances.

Mongo EMF[5] is another alternative to store EMF models in a MongoDB database. Mongo EMF provides the same standard API than previous approaches, however, according to the documentation, the storage mechanism behaves slightly different than the standard persistence backend (for example, for persisting collections of objects or saving bi-directional cross-document containment references). For this reason, Mongo EMF cannot be plugged without performing any modification to replace another data-store in an existing system.

Hawk (Barmpis et. al. 2013) is a model indexer framework that stores models in graph data-stores and provides an efficient model query API. The framework allows model designers to define specific indexes that will be reused during the query computation to speed-up element and attribute access. While Hawk can be considered as a NoSQL persistence layer for large models, it is not designed to handle EMF-based query computation efficiently, and relies on the EOL (Kolovos et. al. 2006) to efficiently navigate and manipulate models.

EMF fragments (Scheidgen et. al. 2012) is another NoSQL-based persistence layer for EMF that aims to achieve fast storage of new data and fast navigation of persisted models. EMF fragments principles are simpler than in other similar approaches and reuse the existing proxy mechanism of EMF. In EMF fragments, models are automatically partitioned in several chunks (fragments). Unlike CDO and Morsa, the granularity of the lazy-loading mechanism is defined at the fragment level, that are entirely parsed and loaded when they are accessed. Another difference with other approaches is that additional information have to be specified in the metamodel to benefit from the partitioning capabilities of the framework. This approach makes EMF fragments both dependent on the quality of the provided partitions and the size of individual fragments.

NeoEMF (Daniel et. al. 2016) is a multi-database model persistence framework that aims to provide the appropriate model-to-database mapping according to a given modeling scenario. The framework provides three built-in NoSQL implementations that allows to store large models in: (i) key-value stores that are optimized for repeated atomic accesses typically generated by modeling APIs, (ii) graph databases that are designed to compute complex navigation queries efficiently, and (iii) wide column databases that provide advanced distribution and concurrency management. NeoEMF is fully compliant with the standard EMF API, and does not require platform-specific steps to setup and connect to the database.

These solutions typically improve the performance for storing and accessing large models when compared to relational database persistence layers (Barmpis et. al. 2013; Daniel et. al. 2016). However, they also often require a specific initial step to start the database server and open a connection before allowing model manipulations. In addition, the use of an additional query language is often required to fully benefit from the database capabilities (Pagan et. al. 2015). Alternative querying solutions have been proposed to tackle this issue by generating database queries from modeling operations expressed in high-level languages such as OCL and EOL (Daniel et. al. 2016, De

[5] https://github.com/BryanHunt/mongo-emf/

Carlos et. al. 2015), bypassing the modeling API and benefiting from the underlying database's features (such as query optimization and index accesses).

An example of a model to NoSQL mapping: A graph-based representation

As explained above, NoSQL-based solutions have been proposed to tackle the limitations of relational databases to handle large models. However, since these approaches are mainly schema-less, different ways of representing models' data have been proposed. Next, we describe one of the approaches that has been proposed using NoSQL technology. Specifically, we describe the graph-based representation proposed in NeoEMF to persist object models.

In order to better illustrate the data mapping, we first introduce an example of a simple metamodel and a corresponding instance, and secondly, we describe the mapping. Figure 1 shows an excerpt of a simple Java metamodel that represents Java programs at a low-level of abstraction. A *Package* is a named container that can recursively contain other *Packages* through its *subPackages* composition. A *Package* also contains several *Classes*, which define a *name* attribute and an *imports* association representing its imported *Classes*. Each *Class* contains a set of *Methods*. A *Method* is linked to a *Modifier* describing its *Visibility* (*public*, *private*, or *protected*), and a *returnType* that represents the *Type* that is returned by the method. Finally, a *Constructor* is a specialization of *Method*.

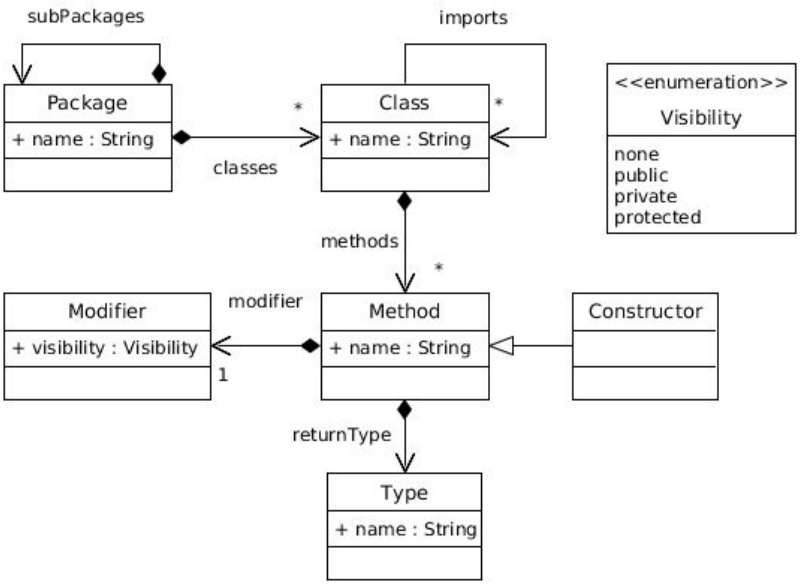

Figure 1: A Simple Java Metamodel

Figure 2 shows an instance of this metamodel that contains a single *Package* `p1` named *package1* composed of two *Classes* `c1` and `c2`, respectively named *class1* and *class2*. The *Class* `c2` *imports* `c1`, and contains a single *Method* `m1` named *method1*. This *Method* is linked to a *private Modifier* (`mod1`) and the *void ReturnType* (`t1`).

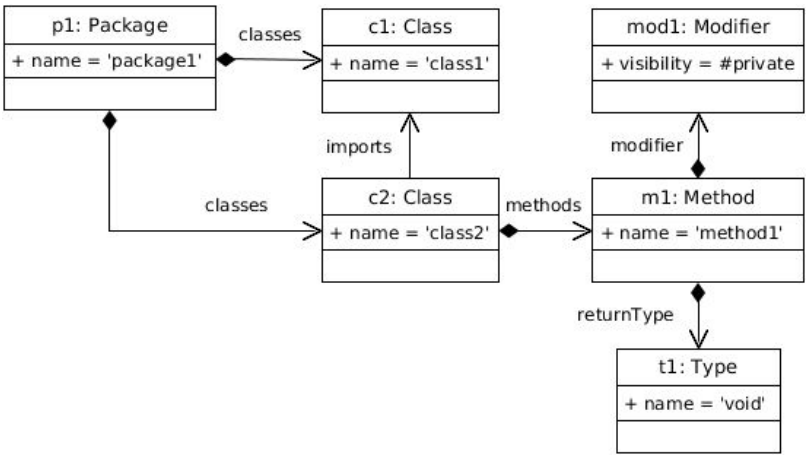

Figure 2: Example Instance of the Java Metamodel

The graph-based mapping in NeoEMF is implemented in NeoEMF/Graph – the connector designed to efficiently compute complex model navigations (Benelallam et. al. 2014). It relies on generic graph database structures (nodes, relationships, and properties) to represent models, where each element is represented as a database node, and connections between elements (associations, compositions, etc) are represented as relationships.

This generic graph-based representation allows benefiting from any database engine that is designed to efficiently compute complex relationship-based navigations. In NeoEMF, the graph connector is used as the basis of the Mogwai query framework (Daniel et. al. 2016) and the Gremlin-ATL transformation engine (Daniel et. al 2017) that intensively rely on complex database navigations to compute OCL queries and model-to-model transformations, respectively.

Figure 3 describes how the instance model previously presented is persisted in a graph database. Figures 1, 2, and 3 show that:

- Model elements are represented as nodes. Nodes `p1`, `c1`, `c2` are examples of this, and correspond to the elements `p1`, `c1`, and `c2` shown in Figure 2.
- Element attributes are represented as properties stored in the node corresponding to the containing element. Node properties are represented using key-value pairs {*property_name*, *property_value*}. For example, nodes `p1`, `c1`, and `c2` contain a *name* property that contains the value of the name attribute in Figure 2.
- Metamodel elements are also represented in the database as (grey) nodes, that are indexed to ease their access. Metamodel nodes contain two properties: the first one hold the name of the metamodel element, and the second one the metamodel unique identifier (*nsURI*). `Package` and `Class` are examples of metamodel nodes.
- Type conformance relationships are represented as `instanceof` edges between the node representing the metamodel element and the one representing the instance of this particular type.
- References are represented as edges between the nodes corresponding to the connected elements. These edges are labeled with the name of the association defined in the metamodel, and can contain a `position` property that defines the index of the relationship if

the base association is multi-valued. This is emphasized in Figure 2 by the two classes edges between `p1` and `c1`/`c2`. For example the edge `imports` that links `c1` and `c2` corresponds to the `imports` association in Figure 2. Note that compositions are implicitly bidirectional[6], and are translated into one edge representing the composition itself, and an opposite one labeled `eContainer` (green edges in Figure 3) that represent this implicit containment link.

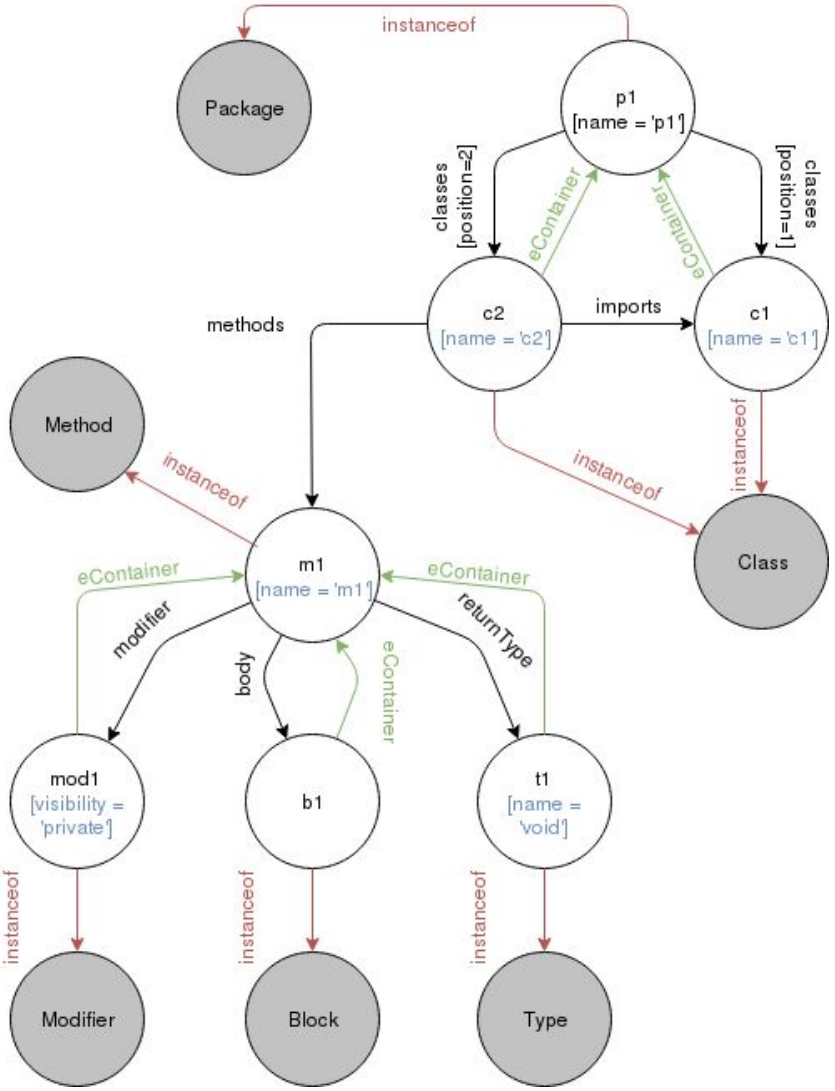

Figure 3: Example Instance of the Java Metamodel

The resulting database contains all the model's information as well as part of the metamodel structure to optimize type based operations. For example, the *allInstances* method can be easily computed by searching the indexed node representing the metaclass to compute the instances of, and by navigating all its outgoing `instanceof` edges. A graph representation intensively relies on the interconnected nature of models to efficiently compute navigation queries. This representation also uses node properties to store attribute information. In addition, the efficient navigation capabilities of the database engine are restricted by the high-level modeling framework APIs, that typically generate low-level fragmented queries that are hard to optimize and have a significant impact on the engine's performances.

[6] This is an implementation detail imposed by EMF and how EMF models are built.

3. Model Viewpoints and Views

This section provides an overall state-of-the-art of the approaches and techniques that can be used to efficiently create, edit and manipulate views (or similar concepts) over one or several existing models (that can be stored, retrieved and queried using the solutions previously studied and presented in Section 2). Note that we do not intend to tackle in this document specific graphical/visualization aspects (also sometimes related to "views"), but rather model views in the general sense (cf. next subsection and the analogy with views in databases). Related and complementary support for specifying the corresponding viewpoints over one or several available metamodels, possibly very heterogenous, is also described in what follows.

3.1. Background - Views in Databases

Views have a very long tradition in databases since the introduction of relational databases (WIEDERHOLD, 1986) and, consequently, have been also proposed for other types of database systems such as object-oriented (ABITEBOUL et al., 1991) ones or more recently graph ones (ANGLES et al., 2008).

The data in a database view is computed dynamically upon access, and can be cached in a database table as a so-called materialized view. Queries expressed in relational algebra (for a relational database of course) and corresponding view definitions specify two dimensions of the result sets to be computed:

1. The schema of the given result set, i.e. the specification of the needed columns (equivalent to the needed properties);
2. A selection of rows (i.e. tuples providing actual values) according to this schema.

The different main types of operators available in relational algebra have different effects on these dimensions. While projection affects the set of columns/properties, selection affects only the set of rows/tuples. Join operators affect both dimensions simultaneously.

(Relational) Database	Modeling
Schema	Metamodel
Table	Concept
Table column	Property (attribute of reference)
Table row	Model element
Table rows	Model

Table 1: Conceptual Analogy between Relational Database and Modeling - adapted from (BURGER et al., 2014).

In the Modeling community, views are usually defined in a similar way. In Table 1 the main constructs of metamodeling and relational structures have been identified and related to each other. Assuming that a metamodel corresponds to a database schema, a model corresponds to the content of database tables, i.e. rows/tuples. In this sense, each table in the database can be regarded as a concept from an equivalent metamodel. Each column from a given table can be thus considered as

describing the corresponding property (i.e. an attribute or reference) of such a concept. These analogy and similarities can be practically observed in several of the approaches mentioned later on in this deliverable. In somes cases, they are even the essence of the proposed solution.

3.2. Intra-Model Views within a Modeling Language

View / Viewpoint mechanisms can be considered in the context of given modeling languages, either standard / generic or domain-specific ones. Indeed, there are several approaches that have been designed for allowing users to define intra-model views. In this subsection, we are going to provide an overview of such approaches that focus on viewpoints / views for a given specific and/or single modeling language.

3.2.1. Multi-viewpoint Modeling

Multi-viewpoint modeling approaches are generally used to design different viewpoints for a given modeling language / base metamodel. They rely on considering a unified single model, already available as is or previously generated for this particular purpose (e.g. via model composition techniques, cf. Section 3.3.1), that provides all the required view data. As shown on the following figure, the application of such approaches results in the computation of different submodels (from the base model) that constitute the expected views (according to the specified viewpoints).

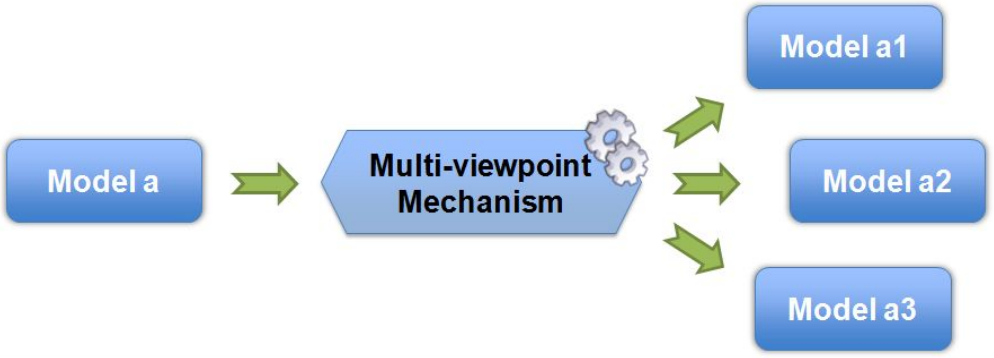

Figure 4: General Principle of (Model) Multi-viewpoints Mechanisms.

There are several existing approaches for multi-viewpoint modeling, cf. (GOLDSCHMIDT et al., 2012) for a survey. Orthographic Software Modelling (OSM) (ATKINSON et al., 2010), a projective multi-view methodology in which views are dynamically generated via model transformations from a single base model. However, due to this single base model, integrating models coming from several disciplines or layers does not seem to be their focus. A mechanism has been proposed to improve such views in OSM (BURGER, 2013), in which flexible views can be defined at development time using a textual domain-specific language (DSL). The currently offered DSL only supports read-only views. Another methodology has been developed to create an underlying metamodel (from base metamodels) by using structural mappings between metamodel elements specified using a correspondence DSL (KRAMER et al., 2013). This approach has a mechanism to support extending materialized viewpoints and synchronizing model views with the underlying model. In (CICCHETTI et al., 2011), the authors propose another multi-view modeling approach, where viewpoints are subsets of a base metamodel as in projective approaches. Higher order transformations are then used to create transformations generating views (submodels) from viewpoint definitions. In (ROMERO et al., 2009), the authors propose an approach for specifying and realizing correspondences between a set

of viewpoints. They demonstrate how correspondences can be modeled and propose some model transformations to derive extensional correspondences from intensional ones.

3.2.2. Specific Language Support

Some modeling languages have been natively designed to allow representing different views over the same modeled systems. This is especially the case of general-purpose modeling languages, which intend to cover a large set of potential contexts and application scenarios. Thus, from a same central model different views or "diagrams" can be considered over it (e.g. a test engineer and an architect would have a different view on the same model). Such languages usually follow a similar approach, even though they do not necessarily provide a proper implementation to actually realize it (cf. the standards mentioned hereafter for instance):

- They integrates directly within their core metamodel/specification the support for different views or "diagrams";
- They propose to customize the language by extending and/or refining its metamodel to address user-specific needs.

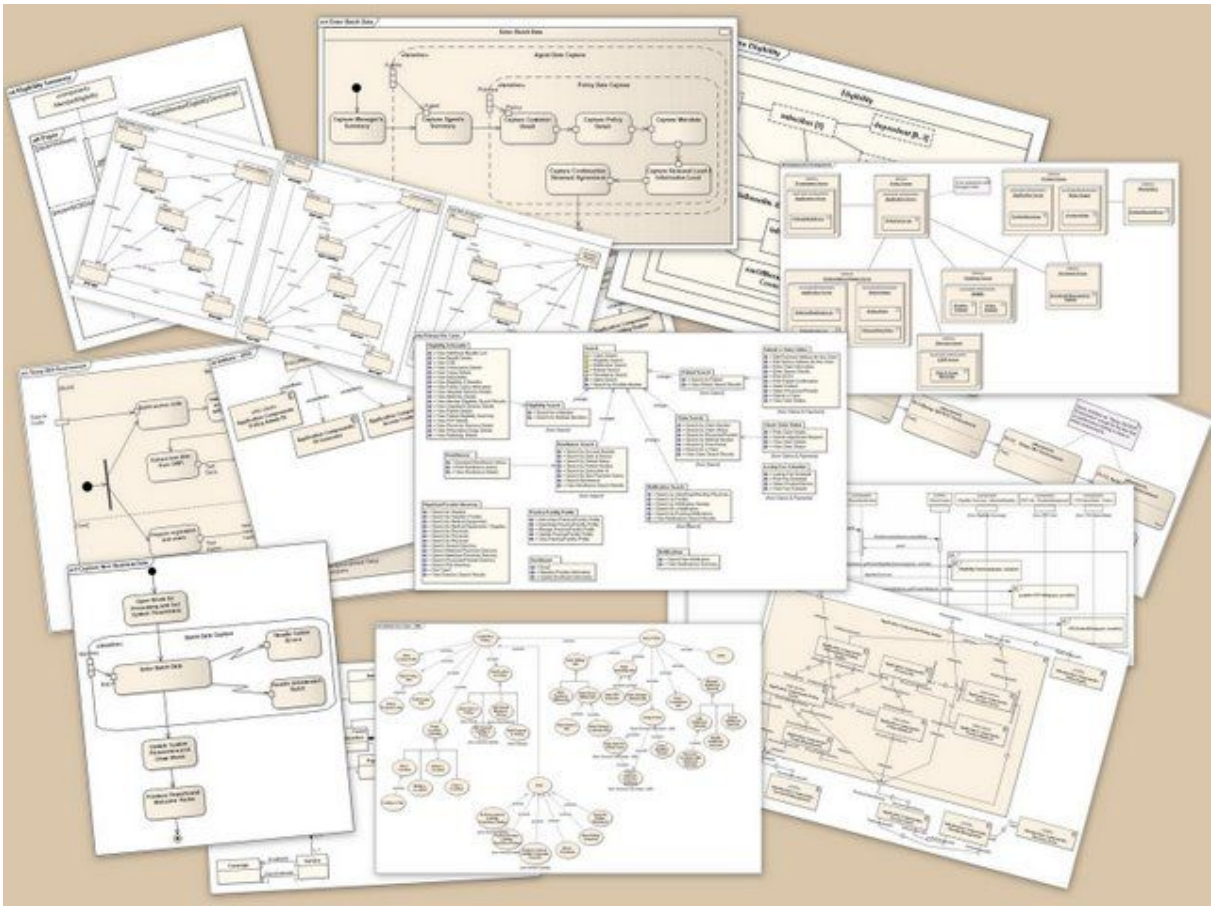

Figure 5: Different diagrams/views in OMG's UML.[7]

The most well-known and widespread example of such a language is probably the Unified Modeling Language (a.k.a. UML) from the Object Management Group (OMG's UML, 2017). As shown on Figure 5, this standard language comes with 14 different diagrams from two main kinds: the ones

[7] https://commons.wikimedia.org/wiki/File:UML_Diagrams.jpg

dealing with structural aspects of software systems (namely class, component, composite structure, deployment, object, package & profile diagrams) and the ones dealing with behavioral aspects of such systems (namely activity, communication, interaction overview, sequence, state, timing & use case diagrams). Each diagram can be seen as a distinct viewpoint from which a given software system can be observed. However, many UML model elements are actually common and shared across different diagrams. Kind of similarly to the multi-viewpoints modeling approach previously presented in this document, different sub-models / diagrams / views can thus be defined from a single large UML model containing all the required model elements.

Examples of language specific support for views also exist in other domains than pure Software Engineering. For instance, it is a common practice in the Enterprise Architecture domain where standards commonly provide different viewpoints from which to observe and represent an enterprise system and its different dimensions/perspectives. TOGAF from the Open Group (Open Group's TOGAF, 2017), the currently most popular EA standard, relies on the concept of "catalogs", "matrices" and "diagrams" organized as viewpoints that allow stakeholders (e.g. EA architects) to define the various views they need.

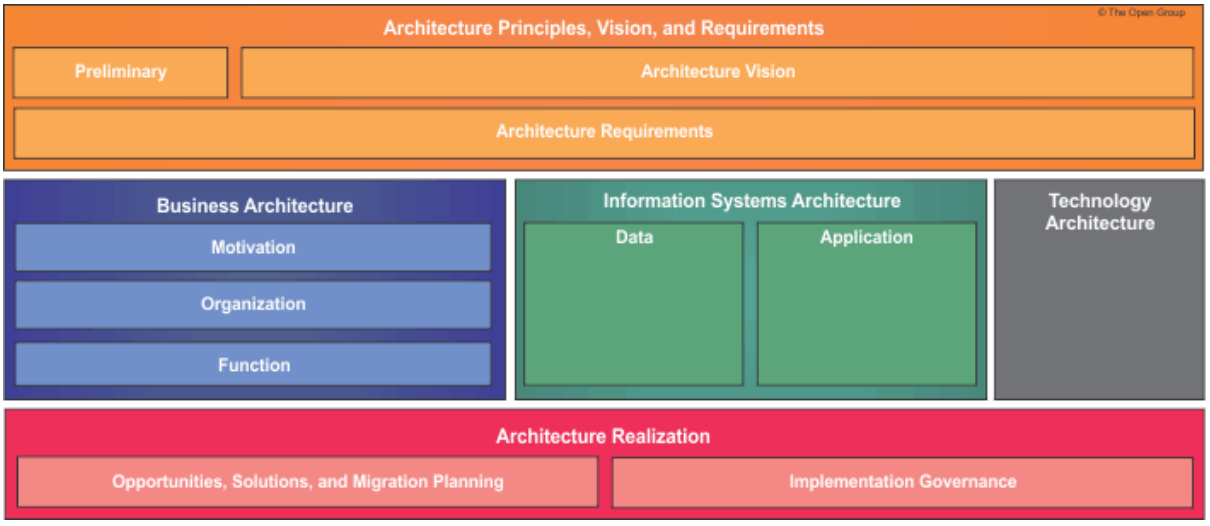

Figure 6: Overview of TOGAF's Content Metamodel (from The Open Group)

From Figure 6, it can be easily seen as the core metamodel of TOGAF (called Content Metamodel) is structured to represent different interrelated views over the same enterprise. For example, the Strategy/Vision, Requirements, Business, Technology or Information System artifacts allow to capture the corresponding models/views.

Some Domain-Specific Languages (DSLs) or Domain-Specific Modeling Languages (DSMLs) have been designed to provide different views over the same modeled systems. For instance, this is the case of languages used in industrial control software, such as those used in trains, cars and airplanes. An industrial control software is a type of software typically used in industries such as transportation, chemical, automotive, and aerospace to provide supervisory and regulatory control (Stouffer, 2011). This type of software is vital to the control of critical infrastructures.

An example of such a language is IEC 61131-3 that is a popular standard for industrial control software used in industry because of its textual and graphical notations and its circuit-like nature (Öhman, 1998). FBD is one of five languages for logical or control configurations defined within the

IEC 61131-3 standard (IEC, 2014) that was originally published by the International Electrotechnical Commission (IEC) in 1993. Other languages defined by IEC are Ladder Diagram (LD), Structured Text (ST), Instruction List (IL) and Sequential Function Chart (SFC). These languages can be separated into graphical (FBD, LD, ST) and textual (IL, SFC) languages. These languages represent different views of the IEC 61131-3 program model. As shown in Figure 7, the standard defines the five views depending on the textual or graphical representation used to define a program model. All views define common elements from all languages, such as project configuration, resources, associated tasks or variables.

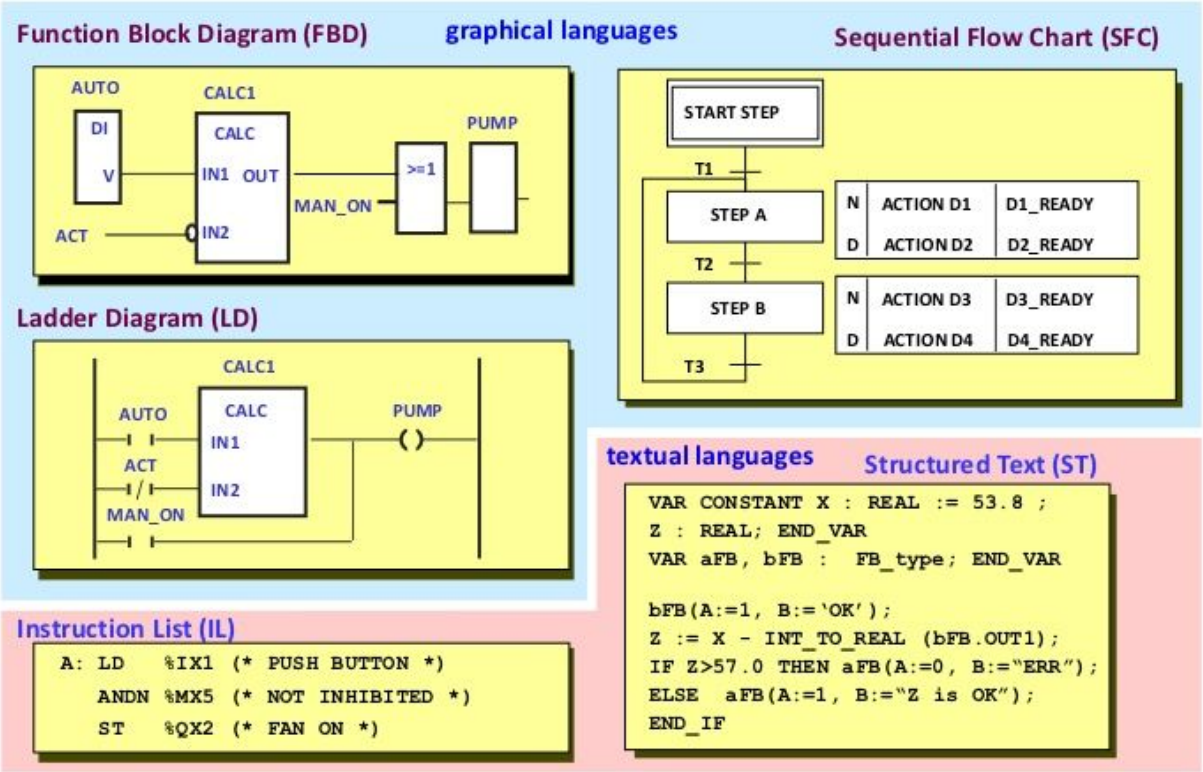

Figure 7: Five different views in IEC 61131-3

Function Block Diagram (FBD). FBD shows a graphical view of the program. These diagrams can be defined using standardized functions or created by a user in a library. A function is graphically represented as a block together with input and output variables and connections between blocks.

Ladder Diagram (LD). A Ladder Diagram represents a program as collection of graphic symbols in a network, representing a logic diagram with input and output parameters represented by contacts and coils. Functions can be used to connect these elements among them.

Instruction List (IL). IL is a textual low-level programming-like view of a program similar to an assembly language. Each of the instructions is composed of a set of operands and an operation indicating which operation is performed. Only one instruction per line is allowed in IL. This representation can be more difficult to debug and can result in complex function representations.

Structured Text (ST). The ST language is defined by IEC as a high-level programming view that can be used to describe the logic of a program in a C-like language. ST uses operators such as assignment, logical branching, and loops.

All of these views share IEC-61131 common model elements. All variables and function calls are defined using these common modelling elements. Different languages within the IEC 61131-3 standard can be used to represent the same program model. The conversion between these IEC 61131-3 languages is used to enhance the user experience in developing, debugging and testing program models.

3.3. Inter-Model Views between Modeling Languages

In addition to intra-model views, view/viewpoint mechanisms can also be considered in the context of several modeling languages and related models. There are several approaches that have been used for defining inter-model views. Some of them were initially designed to this intent, while some other have been re-applied in this context. In this subsection, we are going to provide an overview of such approaches.

3.3.1. Model Composition

Model composition approaches allow defining correspondences between different existing models and generating a brand new (composed) model accordingly. As shown on the following figure, such a composed model can then be considered as a view over the different base models, according to the viewpoint specified by the defined correspondences.

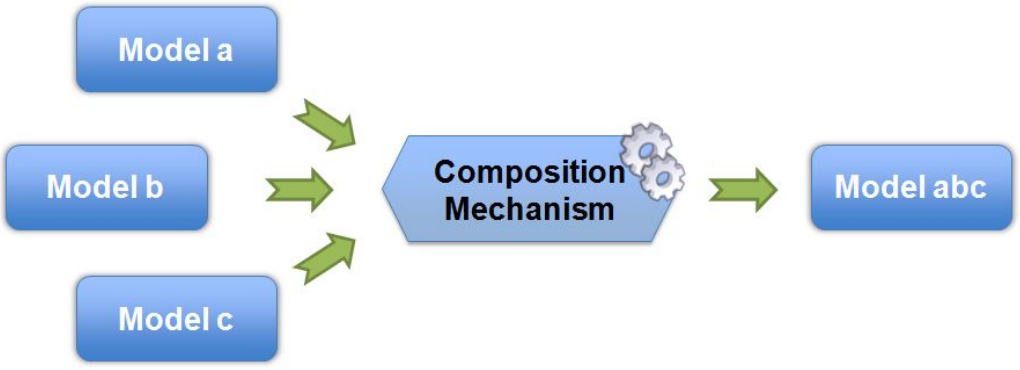

Figure 8: General Principle of (Model) Composition Mechanisms.

Over the past decade or so, many different and/or complementary model composition approaches have been proposed. A large majority of them were not developed with the initial and main objective of providing support for model views. There are approaches (DIDONET DEL FABRO et al. , 2009) (GARCES et al., 2009) (KOLOVOS, 2009) (KOLOVOS et al., 2006) that can be used to somehow simulate views using different link types between models, e.g. traces, correspondences or merging rules. As said before, these approaches do not explicitly focus on model views but rather provide general capabilities which are needed to globally reason about interconnected models. For instance, languages such as the Epsilon Comparison Language and the related Epsilon Merging Language from the same family (KOLOVOS et al., 2006) may help during the composition process, e.g. by facilitating the identification of the elements to merge.

3.3.2. Model Views

Besides all the previously presented techniques and implementing approaches, there are some more dedicated view approaches for models. In (JAKOB et al., 2006) the authors introduce, based on

Triple Graph Grammars, the possibility to have non-materialized views by extending base metamodels using inheritance. In a successor work (ANJORIN et al., 2014), they present an approach for materialized views not modifying the local models, but requiring to explicitly populate the views with elements via model transformations. EMF Facet (EMF FACET, 2017) is another approach to define non-materialized views that are read-only. A kind of similar approach to EMF Facet exists (HEGEDUS et al., 2012), interconnecting models by augmenting the base metamodels with derived features which are computed via incremental model queries.

Another Eclipse/EMF-based solution is Kitalpha (KITALPHA, 2017) that notably proposes a framework to define "views" on models inspired from the use of viewpoints in architectural modeling. The notion of view is very general in Kitalpha and covers more or less completely the following aspects: abstract syntax, notations (such as icons), concrete syntax (textual and graphical), rules (e.g., check, transformation), services and tools.

ModelJoin (BURGER et al., 2014) is a domain-specific language and tool for the creation of views on heterogeneous models. It comes with a declarative language with a human-readable textual concrete syntax that bears similarities to that of SQL. ModelJoin follows a generative approach as it systematically produces both a new target metamodel (the viewpoint) and transformations that create new target models (the views) from the input existing models used to build up the view.

Contrary to ModelJoin, EMF Views (BRUNELIERE, PEREZ, WIMMER, CABOT, 2015) relies on a generic model virtualization framework for realizing both viewpoints and views. It is an approach and corresponding Eclipse/EMF-based tool that provides capabilities for specifying and obtaining views on top of models which potentially conform to different metamodels. It has interesting properties in terms of transparency (a view is seen and handled as a regular model), synchronization (a view always reflects the content of the base models), non-intrusiveness (a view is not altering the base models) and scalability (a view does not duplicate any actual model element). More details on EMF Views, as a tool to be reused and possibly improved in the context of MegaM@Rt2, are provided in Appendix of this document.

Quite interestingly, many of the model view approaches mentioned in these paragraphs can also be used to deal with intra-model views (cf. Section 3.2) in addition to deal with inter-model views (or eventually a combination of both). As a concrete example, EMF Views has already been applied both to deal with heterogenous models federation/integration (BRUNELIERE, PEREZ, WIMMER, CABOT, 2015) and to support the extension and/or refinement of already existing single models (BRUNELIERE, PEREZ, DESFRAY et al., 2015).

4. Global Model Management and Megamodeling

The state-of-the-art in terms of model management and/or megamodeling is quite varied, going from research approaches and prototypes to commercial solutions. This section intends to summarize it covering these two dimensions.

4.1. Megamodeling Principles and Techniques

The idea of *megaprogramming* was introduced in order to propose a solution to the construction of large-scale software systems (WIEDERHOLD et al., 1992). Using a similar general principle, the related idea of *megamodeling* was proposed in order to cope with the accidental complexity that has been observed when building real-life MDE/MBE solutions targeting practical problems (BEZIVIN et al., 2004). Modeling approaches mainly suggest basing the engineering processes on different (meta)models and related chains of model transformations implementing the various activities (Model-to-Model, Model-to-Text or Text-to-Model). A single transformation is often quite easy to handle but, as soon as we tackle complex situations (such as in MegaM@Rt2), large sets of interrelated modeling artefacts are often required to build a full solution. The standard programming tools are usually not well-adapted to manage such modeling artefacts. Thus, it has been proposed to apply modeling itself in order to deal with the accidental complexity that may be generated by complex modeling processes. The overall objective is to offer a global view on and support for the handling of the involved (modeling) resources. This is so-called Global Model Management (GMM) that relies on the core concept of a Megamodel. Next subsections give some base definitions of GMM / Megamodeling as well as more details on already available approaches.

4.1.1. Some Definitions

In what follows, we provide global definitions of the main different concepts related to Modeling / MDE and that are actually useful in the context of megamodeling:

- **System** - A delimited part of the world (the “real world”) considered as a set of elements in interaction. It can integrate both physical and cyber/digital elements and processes. It can be represented in terminal models (M1).
- **Model** - A representation of a given system. For each question of a given set of questions, the model will provide exactly the same answer that the system would have provided in answering the same question.
- **Terminal Model (M1)** - A model such that its reference model is a metamodel, i.e. it conforms to its reference metamodel. It is a representation of a “real world” system according to a given metamodel.
- **Metamodel (M2)** - A model such that its reference model is a metametamodel, i.e. it conforms to its reference metametamodel. It specifies the concepts and relationship types that can be used in corresponding terminal models.
- **Metametamodel (M3)** - A model that is its own reference model, i.e. it conforms to itself. It specifies the meta-concepts and meta-relationship types that can be used to define different metamodels (and also itself).

It is important to notice that the generic term “**model**” is very often used to actually talk about “terminal models” (M1 level). A common practice is to only talk about models and their respective metamodels, considering that the metametamodel is fixed and so is kind of hidden to the end users.

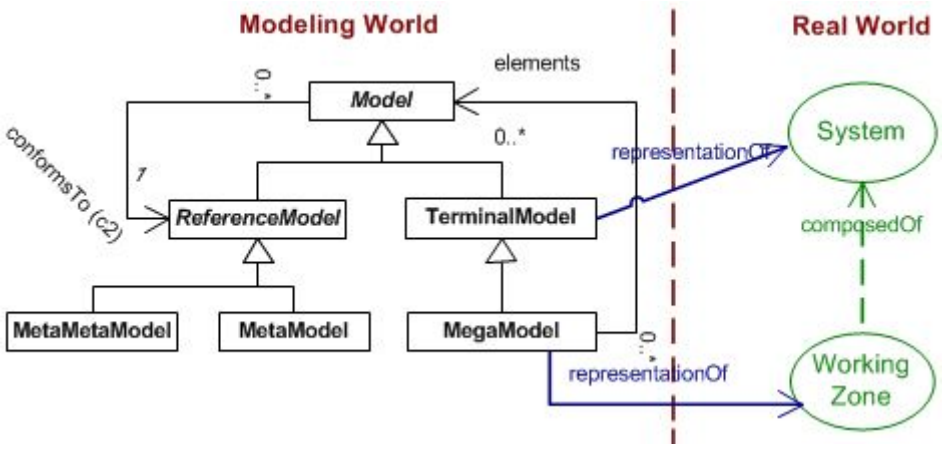

Figure 9: Example Metamodel of Megamodel (excerpt from the AM3/MoScript solution)

Based on these general definitions, the concept of megamodel can then be introduced in order to provide some kind of overall registry for modeling artifacts and their possible interrelationships in the context of Global Model Management:

- **Megamodel** - a model such that its (model) elements are actually 1) references (e.g. pointers, proxies) to modeling artifacts used in/by the modeled systems and 2) relationships existing between these different modeling artifacts. As any model, a megamodel conforms to a given metamodel that specifies the various types of artifacts (models, transformations, etc.) and relationships that can be actually represented. An example of such a metamodel of megamodel is described in the Figure right before.

4.1.2. Existing Approaches

There are already different approaches that have been developed based on the core concept of a megamodel or similar. While sharing significant common features, the overall understanding and expected usages of the built megamodels may differ from one solution to another (HEBIG et al., 2012).

As mentioned before in this document, the first occurrences of the term “megamodel” appeared more than a decade ago (BEZIVIN et al., 2004). The definition has then evolved and has been refined several times according to the approaches that have been progressively developed. For instance, quite early there were attempts to provide a stronger theoretical background behind the notion of megamodel (FAVRE, NGUYEN, 2005). Notably, related efforts have been continued later in order to provide a more formal typing system for the modeling artifacts that are represented in a megamodel (VIGNAGA et al., 2013).

In parallel, technical solutions based on these concepts have been designed and developed in order to propose a more practical support for megamodeling. For instance, the Eclipse/EMF-based AM3 prototype was one of the first to rely on megamodels (ALLILAIRE, 2006). Later on, it has been extended with a proper DSL, called MoScript, intending to facilitate the querying and manipulation of the built megamodels as well as of the represented modeling artifacts (KLING et al., 2011). Some elements of this general AM3/MoScript approach may be reused and/or adapted in the context of MegaM@Rt2, cf. the corresponding Appendix entry in this document.

Another approach has been developed based on the very similar concept of “macromodel” (SALAY et al., 2009). In this solution and related work, they notably insist on the graphical nature of such

macromodels and on attaching detailed information to the represented relations between modeling artifacts. More recently, interesting work has been performed on making megamodels more general and on considering the problem also from an overall Software perspective (and not only from a Modeling perspective anymore). For example, this resulted in the MegaL megamodeling language (LAEMMEL, 2014) which intends to model the architecture of a whole software systems in terms of all the involved software artifacts (including code artifacts for instance, and not only models), software languages (e.g. programming ones) and related technologies (e.g. used frameworks, transformations, generators).

MDEForge (Di Rocco et. al. 2015) is another recent approach that relies on the concept of megamodel for managing stored interrelated modeling artifacts. In particular, MDEForge is an extensible modeling platform that consists of a set of core services that permit to store and manage typical modeling artefacts and tools. Atop of such core services it is possible to develop extensions adding new functionalities to the platform. All the services can be used by means of a Web access and by a REST API that permits to adopt the available model management tools as software-as-a-service. It is important to remark that all the artifacts stored and managed by the MDEForge platform are related to the users that created them. The system permits also to make artifacts public, private, or limit their visibility to specific users.

4.2. Industrial Solutions

As stated before, there are also different existing commercial solutions to deal with model management activities. This subsection details a bit more the Modelio/Constellation approach and compares it to other commercial solutions.

4.2.1. Model decomposition

A model can be composed of (hundred, thousand, million) element(s). At some point any technologies will not be able to correctly (in term of performance) handle huge models. A first approach consist in introducing one or many intermediate layer between the Model and the Element concept. Figure 10 depicts a Model decomposed in one additional layer called Fragment. Such model is now composed of set of fragments, which in turn consist of sets of elements. With this kind of architecture, when a user work on a specific model he is not obliged to load all the fragments but only the relevant ones. Each fragment can also be stored in a different storage according to the performance needs.

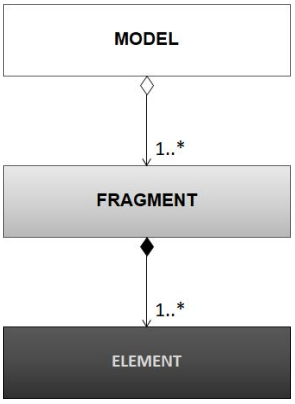

Figure 10: Decomposing models in fragments

4.2.2. Access right

Most of the existing file systems are able to assign access rights to specific users and/or user groups. Such systems control the ability of users to view, change, navigate, and execute the contents of the file system. Similarly, it is possible to apply the same ideas by assigning user and/or user group access rights to model elements (which can be even spread over different files). This approach can reduce the size of necessary resources with respect to user rights. For example, if a given user does not have the rights for accessing a specific set of elements then the used model management system will not load such elements. Even in the case of a read only access, the technology can use a lazy loading technique in order to reduce the required resources.

4.2.3. Decomposition and user access combination

The combination of the two approaches previously described allows to assign fragment rights to specific users and/or user groups. Figure 11 shows how such a combination can look like e.g., by using different storage technologies depending on the access rights to be managed (with a possible performance enhancement of the considered global model management system).

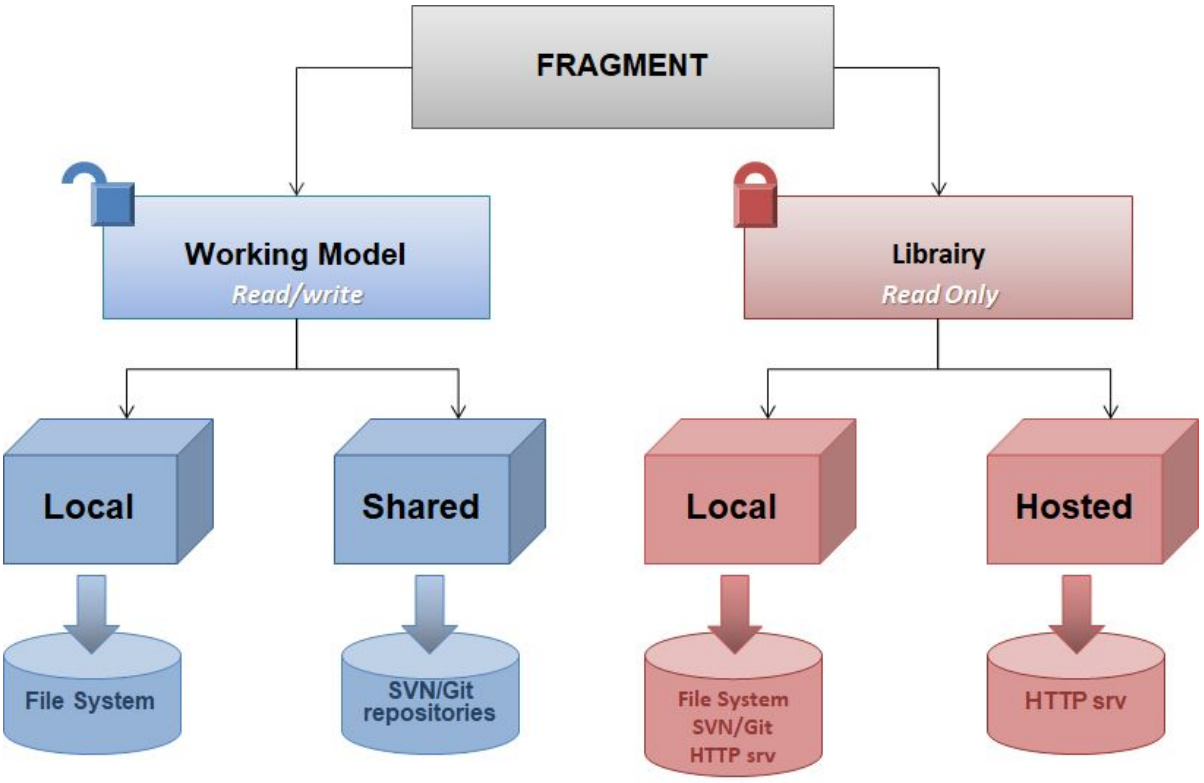

Figure 11: Decomposing models in fragments

4.2.4. Tools Comparison

In Modelio, fragments are viewed as independent and coherent parts of a larger model that are packaged into a single file. These fragments allow developers working on different Modelio projects to set up effective mechanisms for the delivery/reception of model parts. Fragment increase the efficiency of large teams, as they reduce the size of the different model extracts involved in a project, and the number of people working on any given model part. This enables each development teach to

carry out independent work according to their own deadlines, without impact the work of the other project participants.

Modelio also introduces the novel concept of a "Constellation" de-centralized repository approach: It allows models to to be distributed and shared in a vast and immediate way, while avoiding the necessity of having a centralized repository. The distribution of models is carried out through fragments. Constellation can be viewed as a network (or web) of fragments, that constitute autonomous groups of model elements. In a typical model Constellation, several servers coexist and share model fragments, by governing and controlling models access to the wide community. The fragments managed by a server can be accessible from other servers by allowing the most openness and the widest sharing. Conversely, fragments can be visible to only a restricted community, in accordance with strict rules and security and confidentiality constraints.

The following model element aggregation structures can therefore be identified using the Modelio Constellation Approach:

- **Configuration units** are at lowest level granularity. They are groups of model elements that are individually managed by the user. While their definition could vary, in reality they constitute a management unit for the user, who manages their version and configuration. They typically correspond to a Class, a Use Case, a Component, a Process or a Package. A configuration unit can also be locked to control and avoid concurrent accesses to it: it is a user's work unit.
- **Fragments** are groups of higher level model elements. Defined in the form of packages, they group configuration units. Their primary function is to *autonomously store* a part of a model. Their components can be linked to model elements belonging to other model fragments. It should be noted that a model element is also not definitively linked to a fragment. It can be moved from some fragments to other ones, according to the organizational needs of the designers.
- **Models** are groups of fragments with a defined goal (e.g. the model of an application or of a system architecture) and they are at the highest granularity. Models identify the fragments which compose it and group them in a specific context. Fragments exist independently of models, and can be referenced by several models.

In Modelio, each fragment has its own independent storage, and can have several access modes (e.g. local, shared, versioned and configured, library, web, secure web). The model elements in a fragment can be linked to other model elements from different fragments. Thus, a model groups fragments together and has a range, which includes several distributed fragments. Using Modelio, which acts as a fragment and model browser, the user can edit a model, which is a set of fragments whose components are interlinked. These fragments can be transparently local or remote via the web or any other network. Additionally, users can edit or browse a model of a very large scale, without having to worry about the actual location of the handled elements, which are scattered across different fragments in the Constellation.

It should be noted that as with internet browsers, access to other fragments via the web cannot be guaranteed. For instance. remote elements might be unavailable, the internet connection can be interrupted, etc..The architecture of Constellation guarantees that links from source elements to remote elements remain visible, and that the reconnection with absent elements happens transparently, as with broken links and absent pages in internet browsers.

The following table compares the Modelio/Constellation solution with other professional solutions on the same market.

Tool	Model Decomposition	Access Rights	Configuration Management	Model Libraries
Modelio - Constellation (Softeam)	Model Elements, Fragments, Models.	Yes, defined at Fragment level: for e.g. read/write or read only	Yes, can be coupled with a configuration management tool such as SVN	Yes. Packages and their content can be exported in the form of Model libraries (read only)
Enterprise Architect - EA (Sparx Systems)	Model Elements, Models.	Yes, given at repository, file, and model element levels.	Yes, can be coupled with a configuration management tool such as SVN	Yes, based on the management tool capabilities. EA allows packages (and their content) versioning.
MagicDraw (No Magic)	Model Elements, Models.	Yes, defined at the project level.	Yes, can be coupled with a configuration management tool such as SVN	Yes, based on the management tool capabilities.
Visual Paradigm (VP)	Model Elements, Models.	Yes, at the project level several configuration are possible.	Yes, can be coupled with a configuration management tool such as SVN	Yes, based on the management tool capabilities. VP allows to create branches.
Rational Software Architect - RSA (IBM)	Model Elements, Fragments, Models.	Yes, by using dedicated extension (Rhapsody Design Manager)	Yes, can be coupled with a configuration management tool such as SVN	Yes, RSA is able to version models.
Eclipse Papyrus	Model Elements, Models.	Yes, at model and/or model element level (via the Papyrus customization features).	Yes, can be coupled with solutions for configuration management (SVN/Git, CDO, EMFStore, etc.)	Yes, based on the management tool capabilities (for storage, versioning, collaboration, etc.)

Table 2: Comparison between different professional model management solutions

5. Inter-model/DSL Traceability and Interoperability

This section describes an overall state-of-the-art of the approaches and techniques for providing efficient support for traceability, integration and communication between heterogeneous artifacts at different system levels, especially between design models (as produced in WP2) and runtime models (as produced in WP3). Corresponding traceability links can be represented via megamodels and/or model views (cf. Section 4 and 3 respectively), and can be computed via some model querying techniques (cf. Section 2). Furthermore, such links can be exploited for supporting feedback loops aimed at improving the system design with respect to functional and non-functional attributes.

5.1. Traceability approaches in MDE

In this section, we introduce relevant existing approaches for traceability in the MDE context. . In particular, we refer to [PAIGE ET AL. 11, AIZENBUD-RESHEF ET AL. 2006], where different kinds of traceability approaches have been classified.

5.1.1. Model-to-model transformation approaches

In MDE, a number of model transformation tools as well as other model management operations, support the automated generation of trace-links.

The ATLAS Transformation Language (ATL) [JOUAULT ET AL, 2006] uses higher-order transformations to add trace information in the transformation model. The ATLAS ModelWeaver has been used to specify trace links and automatically generate trace-links that conform to a specific weaving metamodel [DIDONET DEL FABRO ET AL, 2009].

The Epsilon framework [KOLOVOS ET AL, 2008] also provides traceability support through an external trace model that can be accessed explicitly in Epsilon's workflow mechanism (which is based on ANT). An Epsilon program, such as a transformation or a model merging operation, can expose trace information (in the form of a trace model) and this information can be accessed by other model management tasks (such as validations) or even non-MDE tasks, such as visualizations generated with GraphViz.

The KerMeta framework also provides facilities for capturing trace-links from transformations, through its Traceability Model Development Kit [VOJTISEK, 2009]. KerMeta also provides a generic traceability metamodel and helpers for optimizing the trace model (which may be a particular issue when managing large collections of trace links).

In the MOF QVT Request for Proposals (RFP) issued by OMG in 2002, traceability is defined as an optional requirement [OMG's MOF QVT, 2007]. The specification describes three model transformation languages that can be used: Relations, Core, and Operational Mappings. In the Relations and Operational Mappings languages, trace-links are created automatically without user intervention. In the Core language, a trace class must be specified explicitly for each transformation mapping. The QVT Operational (procedural) implementation that is part of the Eclipse M2M project [ECLIPSE FOUNDATION, 2009] is an example of this kind of support. The implementation does not store trace models as external files, that would be inter-exchangeable between tools. This can be seen as a disadvantage of the tool [KURTEV ET AL, 2007].

There is also an existing approach [ENOIU, 2016] [ENOIU, 2013] for translating embedded software represented in the FBD model to a timed automata representation for model checking,

simulation and testing. The translation is used in the CompleteTest framework [ENOIU, 2016] which is based on the formal definition of the FBD model. FBD specifications are validated by means of automatically generated tests. In particular, logic coverage criteria are used to define what test cases are needed and a model checker is used to generate test traces. An FBD model consists of the following elements: composite programs, basic blocks, library blocks, connections, ports, and timing constraints. The transformation considers that each modeling element, except for the composite programs, has a set of ports through which it can exchange data. Ports are associated by a set of data types, which are used for data representation, e.g., integer with a specific range. A Port is associated with the same type of data as the associated internal variable. For an FBD program the input ports may only be accessed at the beginning of each computation, and output ports are only written at the end of the computation. Therefore, the behavior is augmented with an external interface. The interface of a block consists of ports and the execution order information. An input port has an associated variable holding the current data values. The internal computation of a block starts with reading all input ports. This internal data is used together with the behavioral model during execution, before writing the variables to the output ports.

The CompleteTest model-to-model transformation is based on the PLCOpen language implemented as an XML profile that provides the ability to describe FBD programs using this profile. The PLCOpen language provides both structural and graphical information needed for implementing the actual translation. The model-to-model transformation generates PLCOpen files in an XML format. The FBD model conforms to the PLCOpen profile and meta-model. The structural translation maps an FBD program into timed automata. The structure of the timed automata model is the basis of the model to text transformation into the UPPAAL input model. The modeling elements of an FBD program used in the translation represent the structure of the model, the behavior, and the timing information. The meta-model elements provide concepts used in component based design. A Block element can be translated with Type, ExecutionOrder and Model elements. Blocks can be composed using connections and ports. Furthermore, a Block element can have a behavioral description as a Model element. The model provided after the translation represents the model annotated with triggering and timing information with assumed functionality.

5.1.2. Bidirectional model transformation approaches

Bidirectional traceability needs to be implemented both forward and backward (e.g., from design to runtime models and from runtime models back to design). Such bidirectional traceability helps determine that all design decisions have been completely addressed and that all models at runtime level can be traced to a valid source even in case of changes.

Triple graph grammar (TGG) based approaches to model transformation provide explicit means for capturing trace-links between source and target model elements through correspondence models [GIESE ET AL., 2006]. Correspondence models are used in the process of generating a TGG specification of a model transformation.

As said before, the QVT-relations, that is the relational and bidirectional language belonging to QVT, generates bidirectional trace links automatically during the transformation execution.

Janus Transformation Language (JTL) [CICCHETTI ET AL. 2010] [ERAMO ET AL. 2015] is a constraint-based and relational model transformation language specifically tailored to support bidirectionality, change propagation and traceability (see Appendix A.6).

5.1.3. Model-to-text transformation approaches

Existing model-to-text languages have support for generating trace-links. In the MOF Models to Text Standard [OMG's MOF, 2006], trace-links are defined to be explicitly created by the use of a trace block inserted into the code, allowing model elements to be traced directly to blocks of text. This approach provides user-defined and user-customisable trace-link definitions; this is specifically useful for adding traces to parts of the code that are not easy to generate automatically. A drawback of the approach is a cluttering of the transformation code with explicit information related to traceability.

A complementary approach, as taken in MOFScript [OLDEVIK ET AL. 2005], is to automate the generation of traces based on model element references.

Acceleo Pro Traceability [ACCELEO 2009] is a traceability tool developed by Obeo that handles traceability links between model elements and code and vice versa. This tool enables round trip support; updates in the model or the code are reflected in the connected artifacts.

5.1.4. Traceability metamodels

In terms of support for representing and storing trace links, common practice in MDE is to store them in a model based on some form of traceability metamodel. In [JOUAULT 2005], a simple metamodel tailored to transformation traceability is presented. In the context of the Eclipse Modeling Project, Eclipse Capra[8] provides traceability features allowing the creation of trace links between arbitrary artefacts, as long as an adapter for these artefacts is available. Once these trace links are established, Capra offers features to manage and even to visualize them. Thus, developers have the possibility to traverse the relationships established through the links and understand how the different artefacts of the considered system are connected.

A more sophisticated approach is to define a generic metamodel, such as the Unified Traceability Scheme discussed in [LIMON 2005] or the traceability framework for software product lines presented in [SOUSA 2008]. This metamodel, with proper extensibility mechanisms, should be able to encode any type of traceability links. Another option is to define a core traceability metamodel that encodes a basic set of common features and then define extensions for different types of traceability links. Although this can be implemented with generic weaving tools, the semantics of metamodel extensions are not yet clearly defined. Further discussion on the advantages and disadvantages of a generic metamodel versus a small core that can be extended for each traceability domain can be reduced to a general discussion about generic modeling languages (such as the UML), versus domain-specific languages, a subject clearly outside the scope of this paper.

In terms of storing trace-links that are established by an operation that manipulates a model, [KOLOVOS ET AL. 2006] observe that this can be done in two ways: either by embedding trace-links in the models, or by storing them externally in a separate new model. The first approach gives a human-friendly view of the trace-links, but it only supports trace-links between elements in the same model. The external has the advantage of having the trace information separated from the model and therefore avoids polluting the models and allows definition of traces between different models.

[8] https://projects.eclipse.org/projects/modeling.capra

5.2. Traceability among lifecycle activities

The approaches described in Section 5.1 are currently implemented in several software development tool contexts. In particular, among the existing software modeling and development tools, there are subsets focused on the traceability management. They aim to help the development of complex software system by ensuring, for instance, that design and runtime aspects are traceable and traceability links are updated accordingly to changed entities.

Usually, throughout the lifecycle of a software product, different tools from different vendors are used, where each of these tools addresses specific activities, e.g., requirement management, test management, bug tracking, etc. Two of the major challenges when using different tools are the integration of the involved tools as well as the traceability among the different data artifacts used and managed by the tools (SAADATMAND,2014).

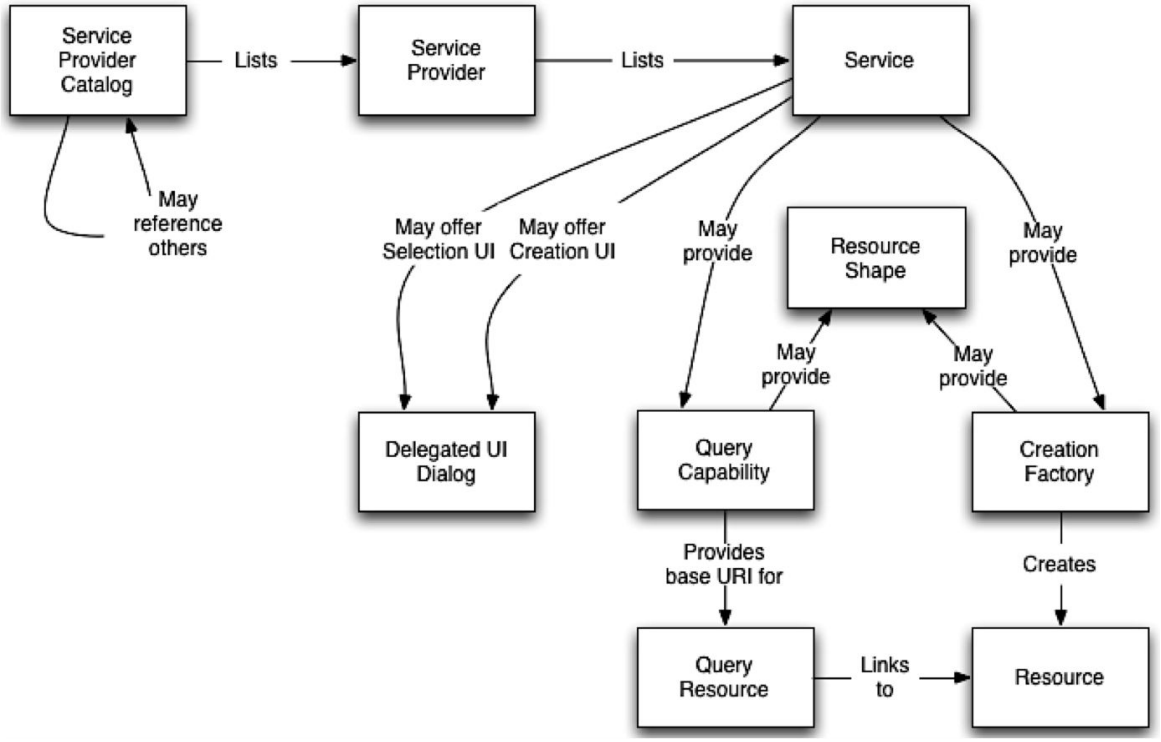

Figure 12: OSLC Core concepts and relationships (OSLC, 2017)

Open Services for Lifecycle Collaboration is an open community for creating specifications for integrating life cycle activities to alleviate the abovementioned issues (OSLC, 2017). OSLC is supported by several companies and organizations such as IBM, Siemens, SHELL, NASA Jet Propulsion Laboratory, and General Motors, to name a few. OSLC community is organized in the form of different work groups focusing on different life cycle integration scenarios. These scenarios, for which specifications are produced, are referred to as *domains* (OSLC, 2017; OSLC PRIMER), e.g., Requirements Management domain (RM), Architecture Management domain (AM), Change Management domain (CM), Quality Management domain (QM), etc. The specification for each domain is built upon one core specification, namely OSLC Core, which defines the basic concepts and rules and ensures consistency among different domain specifications. The figure right before shows the core concepts and their relationships as they are defined in the OSLC Core specification. Using OSLC specifications different lifecycle tools and artifacts can be integrated via standard interfaces.

OSLC tries to be minimal by providing a small set of resources and properties for describing artifacts involved in a specific domain. The figure right before shows the OSLC core common properties used for the Test Case resource definition contained in the QM. Accordingly, one or many testers can be defined as *creators* of a specific test case; each test case is specified by means of its *identifier*, which can be any arbitrary String value; date of creation and date of modification can be specified too using the *created* and *modified* properties. Further properties can be defined if needed, although OSLC deprecates this practice for standardization purposes.

OSLC is based on the concept of Linked Data, *Resource Description Framework*, and HTTP protocol. The figure after shows the application of these concepts and their relationship within OSLC: each artifact is described as an HTTP resource, identified by means of a *Uniform Resource Identifier* (URI), accessed and manipulated with the GET, PUT, POST and DELETE HTTP methods. In other words, OSLC fully supports CRUD operations (Create, Read, Update and Delete) on lifecycle artifacts such as stories, requirements, defects and tests. Considering resource provision and retrieval concepts in an integration scenario, a tool can play two roles, namely *consumer* and the *provider*. That is, for exchanging an artifact, a provider has to encode an artifact's properties in an HTTP resource, and post and provide it under a specific URI; knowing the retrieving URI, a consumer can access the HTTP resource representing the artifact, and simply fetch it from the URI itself. It is important to note that a tool can also be, at the same time, the provider and consumer of different resources.

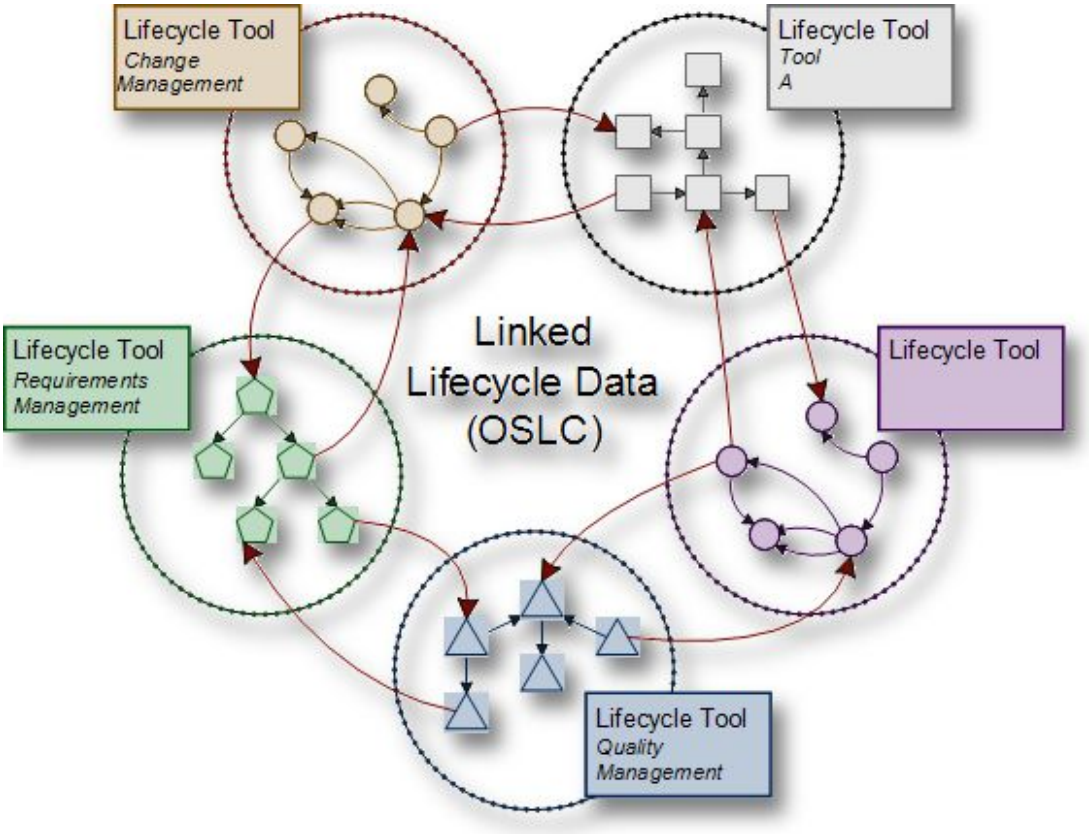

Figure 13: Linked Data in OSLC (OSLC, 2017)

In summary, OSLC specifications serve as a new standard for the integration of tools used in different phases of software development. It enables to establish relationships among different data artifacts throughout the lifecycle of an application. OSLC aims to provide seamless integration of

lifecycle management tools and it enables to have explicit relationships among data artifacts from the early development phases, i.e., requirement (SAADATMAND, 2014).

5.3. Traceability relationships between heterogeneous models

Large-scale software systems are often described using heterogeneous collections of interrelated models (generally represented by megamodels, cf. Section 4 for more details on this term and related approaches) that need to use traceability relations to assess change impact and interoperability between models. There are different approaches to support traceability in the large.

[BARBERO et al, 2007] proposes using global model management to store global traceability relationships. There is a clear separation of classical traceability (traceability in the small) and traceability in global model management (traceability in the large). Classical traceability is considered as the ability to define weaving models that are used to relate model elements of different models. The approach aims at a global model management by putting traceability-related information into a megamodel. As introduced before in this deliverable (cf. Section 4), a megamodel contains relationships between models, for instance transformation models, weaving models, UML models, or metamodels. Therefore, the common megamodel is adapted by replacing simple traceability links with traceability models that have source and target relationships to different models in the mega model. In their particular case, a traceability model is a weaving model implementing traceability between models by defining mappings between model elements. Traceability information is further automatically established by weaving instantiated models into a traceability model.

[SALAY et al. 2008, 2009] is a sophisticated approach joining classical traceability and traceability in global model management called macromodeling (cf. also Section 4). This is realized by a multi model, which contains models and traceability links in between. A traceability link is defined by a meta model, which contains meta model elements and associations of the metamodels it relates to as well as meta model morphisms to map-related metamodel elements to metamodel elements of the meta model of the traceability model. Traceability links are automatically maintained by checking the meta model morphisms on the instances of the models.

[SEIBEL et al, 2010] presents an approach to integrate global model management and classical traceability approaches in the context of MDE. It provides a comprehensive traceability approach by means of a combination of high-level traceability models (mega models) and low-level traceability models. It defines hierarchical dependencies between high-level and low-level modeling artifacts and between traceability links at different levels to glue both traceability models into a combined traceability model. Additionally, the approach provides efficient maintenance of traceability information based on our combined traceability model. The maintenance process is split into two processes called localization and execution. Localization is the process of comprising the creation and deletion of traceability information whereas execution is the process of (re-)establishing the validity of already existing traceability links.

[BEYHL et al, 2013] presents a framework for capturing and maintaining artifacts and traceability links between them using the common format of hierarchical megamodels.

5.4. Feedback collection and analysis

Traceability links can be collected and used for supporting a feedback loop aimed at improving the system design with respect to functional and non-functional attributes. In machine learning, traceability

indicates connecting the data to the mathematical model. To understand the connection from the mathematical model to both the functional and non-functional characteristics requires a deeper analysis of the mathematical model and understanding its connections to either functional or non-functional characteristics, or both. As an example, this can be achieved through careful modification of the specific set of data and evaluating its impact to the functional and non-functional characteristics. Per se, this section only presents a high-level overview of machine learning and deep learning.

5.4.1. Machine learning and deep learning

From the perspective of machine learning and deep learning, an aspect of traceability can be conceived from the perspective of computationally-desirable sparse solutions. Considering high-dimensional data, in which a single atomic piece of data contains numerous individual data elements, typically results in attempting to calculate optimal locations (within the data) that are computationally inconceivable, i.e., the complexity of data overwhelms the computational capacity available or practically feasible (e.g., the computation time exceeds all reasonable expectations.) Fundamentally, machine learning and deep learning methods and tools are data agnostic, therefore they can basically utilize any type of data. Traceability feedback collection belongs to the broad category of time-series data, in which a sequence of data contains the time order of the data. In machine learning and deep learning approaches, the specific selection of the mathematical model is obtained by through the traditional steps of formulating the specification of the dataset (in this case the traceability feedback collection), location of the correct cost function (which indicates the quality of the mathematical model), obtainment of the correct optimization procedure (that enables the starting point of the mathematical model to reach its optimum in the cost function, and the collections of mathematical models to commence with.

In machine learning and deep learning, sparsity means that the solution (data structure, e.g., vector, matrix, etc.) contains numerous zeros. This results in a computationally lighter result as the addition and multiplication of zeros either does not modify the value (former case) or renders it to zero (latter zero.) Sparse solutions can be attained through multiple approaches, such as regularization (specifically with the l1-norm) and dimensionality reduction [GOODFELLOW ET AL, 2016; KOLLER & FRIEDMAN, 2009; MURPHY, 2012; HASTIE ET AL, 2009].

Overall, regularization is the process of introducing additional information (a penalty factor according to the complexity of the proposed solution) to solve a problem or to inhibit overfitting. Typically, in machine and deep learning, a loss function (a function which evaluates the authentic data against the learned solution (by machine or deep learning)) is optimized to locate the best available solution. It is notable, however, that the intention is NOT to memorize or merely copy the solution provided by the authentic data. The ultimate intention is to successfully and correctly generalize to previously unseen data, hence learning the authentic function of relationship between input and output data. The loss function may be penalized (also known as regularized) to inhibit solutions which are too complex. In statistics, Occam's razor states that "Among competing hypotheses, the one with the fewest assumptions should be selected." This can be interpreted as 'the simplest reasonable solution is often the best.' The utilization of the l1 norm results in a solution with multiple zeros. This can be used, for instance, in linear models (ax+b).

Dimensionality reduction incorporates the reduction of the amount of variables which are considered in the machine or deep learning solution. Often the sheer amount of dimensions in the data results in calculations which cannot be completed. The selection of key feature (data elements) and

their extraction (from the rest of the data) gives approximations to be optimized which result in successful and correct solutions. Typical, tangible approaches for dimensionality reduction include PCA (Principal Component Analysis,) LDA (Linear Discriminant Analysis,) and autoencoders. All these approaches attempt, by different means, to explain the comprehensive data set with only a selected amount of key features resulting in an approximation which is adequate.

5.4.2. Performance Feedback Loop

Since a couple of decades, non-functional validation research area has exploited Model-Driven Engineering (MDE) techniques for integrating analysis approaches in the software development process . In particular, early (model-based) validation of performance requirements is needed because performance problems may be so severe that, if late discovered, they can require considerable changes at any stage of the software life-cycle. If performance targets are not met, then a variety of negative consequences (such as user dissatisfaction, lost income, etc.) can impact on significant parts of a project. These factors motivate the activities of modeling and analyzing performance of software systems early in the life-cycle.

Software Performance Engineering (SPE) (SMITH, 2007) has promoted a proactive approach for the satisfaction of performance requirements, aimed at producing performance models early in the development cycle and using quantitative results of model solutions to provide design feedback to the user, in terms of changes (refactorings) that point out architectural design alternatives with the purpose of meeting performance requirements.

The forward path from software to performance models has been fully addressed in the last decade, so that consistent approaches are today available for automating this transformation task (KOZIOLEK, 2010). An open problem, in the context of SPE, remains the interpretation of analysis results and the generation of architectural feedback back from performance models to software models (WOODSIDE ET AL., 2007).However, some work has been done in the last few years to tackle this problem, spanning on different techniques, instruments and tools. Particularly interesting are the ones grounded on performance antipatterns detection (and removal), since have emerged as powerful instruments for performance-driven refactoring of software models.

Wert et al. (WERT ET AL., 2014) introduced heuristics for measurement-based detection of five well-known performance antipatterns in inter-component communications.

Xu (XU, 2008) presented an approach where performance antipatterns are detected on a Layered Queuing Network (LQN) obtained from a software model by means of a bi-directional transformation (ERAMO ET AL., 2015). Refactoring takes place on the performance model, and a corresponding refactored software model is obtained by exploiting transformation bi-directionality.

Parsons et. al. (PARSONS ET AL., 2008) introduced Enterprise Java Beans performance antipatterns. The latter are represented as rules loaded into a detection engine, and the application monitoring leads to reconstruct system run-time design and properties. The detection rules are matched on the obtained software model in order to identify the antipatterns.

Diaz-Pace et al. (DIAZ-PACE ET AL., 2008) presented a framework which assists the software architect during the design to create architectures that meet quality requirements, exploiting simple performance models is used to predict performance metrics for the new system with improved modifiability.

Cortellessa et al. (CORTELLESSA ET AL., 2010) used the Object Constraint Language to define performance antipatterns rules that can be detected on UML (OMG's UML, 2017) models annotated with the OMG's Modeling and Analysis of Real-Time Embedded Systems™ (a.k.a. MARTE) profile (OMG's MARTE, 2017).

Trubiani et al. (TRUBIANI ET AL., 2011) proposed an approach to automatically detecting performance antipatterns within Palladio architectural models (BECKER ET AL., 2009) and suggesting architectural alternatives that might help to overcome the identified performance problems.

De Sanctis et al. (DE SANCTIS ET AL., 2017) investigated the effectiveness of performance antipatterns in the context of software architectures based on Architectural Description Languages (ADLs), by proposing an approach for the automatic detection of four performance antipatterns in Æmilia (BERNARDO ET AL., 2002), i.e. a stochastic process algebraic ADL for performance-aware component-oriented modeling of software systems.

Arcelli et al. (ARCELLI ET AL., 2012) introduced a role-based modeling language in order to characterise performance antipattern problems and their solutions as (source and target) role models. Performance antipatterns can be detected by matching source role models with the software model, and a corresponding solution can be applied to the latter by model differencing techniques applied between source and target role models (ARCELLI ET AL., 2013).

As emerging from above, although performance antipatterns have emerged as powerful instruments for performance-driven refactoring of software models, existing approaches based on them rely on a variety of paradigms, languages, metamodels (CORTELLESSA, 2013). This led to fragmentation and lack of interoperability that not only make difficult identifying the most suitable approach for each specific problem, but also limit the potential impact of such approaches. Such fragmentation issue can be tackled by relying on the EPSILON platform (KOLOVOS ET AL., 2010), which provides an ecosystem of task-specific languages, interpreters, and tools for MDE; Some of those (i.e., Epsilon Pattern Language, Epsilon Validation Language, Epsilon Wizard Language) allow to check properties and apply refactoring on models when the former are not satisfied, thus resulting particularly suitable for providing support to the definition of performance antipatterns detection rules and refactoring actions on design models (ARCELLI ET AL., 2017). Moreover, the different execution semantics of the EPSILON languages enable different model refactoring support. Such support, in practice, is perceived by the final users as a set of different refactoring tools in their hands.

Finally, appropriate model-based trace analysis capabilities can provide useful support that actually brings an added-value in the context of performance. For example, they can reveal significant information for the identification of performance antipatterns (CORTELLESSA ET AL., 2014), such as performance indices coming from runtime up to the design model (HEGER ET AL., 2017). However, they can also be exploited to support model transformations among different artifacts (ALHAJ ET AL., 2013) and to propagate design model changes after performance anti-patterns detection to the other artifacts, i.e, the runtime model, down to the implementation (MAOZ, 2009).

6. Discussion

In what follows, we summarize and discuss some of the main challenges and related problems that have been identified in the state-of-the-art summarized in the previous sections of this document.

Model Storage and Querying

In most of the existing model persistence solutions, scalability is achieved by using a client-server architecture that provides an additional API that has to be integrated in client code to access the model (e.g. to create the server, open a new connection, commit changes, etc). Furthermore, the choice of the data-store is totally independent of the expected model usage (for example complex querying, interactive editing, or complex model-to-model transformation): the persistence layer offers generic scalability improvements, but it is not optimized for a specific scenario. For example, a graph-based representation of a model can improve scalability by exploiting databases' facilities to handle complex relationships between elements, but will have poor execution time performance in scenarios involving repeated atomic value accesses (e.g. querying a log model). As another example, a given model partitioning policy can be a good fit to a modeling task and be inefficient for another one (e.g. design model updated vs runtime model monitoring).

Previous works on model persistence (Benelallam et. al. 2014, Gómez et. al. 2015, Daniel et. al. 2016) have shown that providing a well-suited data store for a specific modeling scenario can dramatically improve performance of modeling applications (Shah et. al. 2014). However, these multidatabase architectures have to be complemented with additional technologies to be able to handle modeling workflows involving large models (Kolovos et. al. 2013) such as model versioning, collaborative editing, and efficient model queries and transformations.

Model Viewpoints and Views

There are some commonly shared limitations as far as model viewpoints and views approaches are concerned. For instance, they systematically require an explicit definition of the viewtype and do not offer the possibility of deriving it at run-time when computing the view. Another typical missing feature is the verification support: most approaches do not support designers in the scalable specification of the viewpoints and corresponding views. For instance, they do not alert the users regarding the applicability or executability of these definitions to build actual views. The few that somehow ensure these properties are doing it more as a side-effect, because the underlying view mechanism itself is strict enough to prevent from some possible issues. Moreover, graphical languages are hardly used, while they could be useful to allow less technical users defining their own viewpoints/views (as tools like graphical query builders for databases have proven to be).

We can highlight a few more challenges worth to be investigated when relevant during MegaM@Rt2. Some of them are actually well-known recurring problems in any technical space where views are used (e.g., in the database domain). Some others are more specifically related to our modeling context.

- *View Updating Problem* - Fully updating a view is not always possible (depending on the applied modifications and on used operators used to get the view). Indeed, some combinations may not result in a deterministic translation of the view update into a set of modifications on the base model elements. A pragmatic solution would be to provide a uniform

way to support several different model view update strategies. As we have observed, current solutions follow a more conservative approach where they basically restrict the changes as soon as they become complex to handle.

- *Incremental View Maintenance* - When (some of) the model view is automatically computed, a major problem is to incrementally update it after changes on the base models. Solutions typically ignore or provide very limited support to this feature. Always completely recomputing the whole view may be too costly and/or trigger undesirable side-effects in some cases. As introduced in the previous item, specific view update strategies could be implemented to provide the needed incremental support to deal with such scenarios (e.g. by relying on incremental model transformation techniques).
- *Concrete Syntax Generation* - The (direct or indirect) definition of the abstract syntax of the view type is a key element in most approaches. Nevertheless, most of them do not offer explicit support to specify the concrete syntax part. To make such views more easily usable by end-users, we should be able to display the view content graphically (and not just showing it by using default tree-like browsers). In order to achieve this, we could generate a default concrete syntax based on the concrete syntax(es) associated to the base metamodels/languages. We could also manually build one explicitly for a given viewtype.
- *Security Aspects* - Views are typically used as a security mechanism to prevent people from accessing data they are not authorized to see and/or modify. This requires the availability of an access-control mechanism enabling to give read/write permissions (on specific model views) to particular categories of persons. Many approaches provide profiles or DSLs to annotate models with security characteristics of the system being modeled. However, they do not allow assigning explicit permissions to the model access itself. This could be possibly addressed by the use of model views in such contexts.

Global Model Management and Megamodeling

The first generation of modeling tools were based on one unique metamodel and gave few features in term of collaboration and model sharing (primarily based on file exchange). With the development of the Model Driven Engineering (MDE), many models, defined according to several metamodels, are now specified, used, modified, and shared between several teams or organizations across the world.

The need to be able to manage efficiently these amount and variety of models ask for solutions. By using recent development in file management, by fragmenting these models in a specific way, or by defining methodologies, specific issues can be solved for specific use cases (e.g. when accessing a given model, cf. Model Storage and Querying approaches as described before). However a global management able to handle various models distributed in a decentralized system for any kind of processes or methodologies still need to be defined.

This generic global management raises several challenges mainly concerning a decentralized edition of models (e.g. design models to be updated remotely), performance and security issues (e.g. limited access required to some data in runtime models), among several teams and/or locations.

Inter-model/DSL Traceability and Interoperability

The development of complex systems requires the use of approaches and techniques for providing efficient support for traceability, integration and communication between heterogeneous artifacts belonging to different system levels. In particular, the usefulness of traceability links increases if maintained between design models (as produced in WP2) and runtime models (as produced in WP3).

In MDE, a number of tools support the automated generation of trace links. In particular, model transformation tools (such as, ATL, Epsilon, QVT) used to specify and generate trace links that conform to a specific metamodel. Bidirectional transformation tools (such as, TGGs, JTL, QVT-R) provide explicit means for capturing trace links between source and target model elements during the transformation execution, whereas existing model-to-text languages (such as, MOF, Acceleo) provide support for trace links generation.

Since large-scale software systems are described using heterogeneous collections of interrelated models, a global traceability (that handles traceability information between models as a whole) is needed. Classical traceability approaches (traceability in the small) handles the trace information between model elements. Current solutions to traceability in the large mix local and global traceability. Other solutions use traceability megamodels (as also discussed in Section 4) that need to use traceability relations to assess change impact and interoperability between models.

Usually, throughout the lifecycle of a software product, different tools from different vendors are used, where each of these tools addresses specific activities, e.g., requirement management, test management, bug tracking, etc. Two of the major challenges when using different tools are the integration of the involved tools as well as the scalable traceability among the different data artifacts used and managed by the tools. In this context, Open Services for Lifecycle Collaboration (OSLC) is an open community for creating specifications for integrating life cycle activities to alleviate the above mentioned issues.

Finally, traceability links can be exploited for supporting a feedback loop aimed at improving the system design with respect to functional and non-functional attributes. Particularly interesting are non-functional attributes such as performance; with this respect, Software Performance Engineering (SPE) promotes a proactive approach for the fulfillment of performance requirements, aimed at producing performance models early in the development cycle and using quantitative results of model solutions to provide design feedback to the user, e.g. in terms of performance antipatterns occurring in the system and changes that point out architectural design alternatives with the purpose of meeting performance requirements. To this aim, the Epsilon platform can be adopted for providing support to the definition of performance anti-patterns detection rules and refactoring actions on design models, and different execution semantics can be exploited in order to enable different types of model refactoring support. The two capabilities above subsume an evolution of the design model, hence traceability links are needed to support the co-evolution of all the artifacts involved in the feedback loop.

7. Conclusion

The present deliverable D4.1 is the initial step in WP4. It aimed to provide an overall state-of-the-art in terms of model management and traceability solutions. To this end, it enumerated and described common principles and approaches related to model storage, querying, handling and linking with others models and modeling artifacts. This notably concerns the available support for model views and/or so-called megamodels. It also presented relevant traceability and interoperability features and solutions. In general, the document includes both existing research approaches as well as some commercial tools/environments. It ended with a discussion on current important limitations and challenges related to the treated topics, and with an initial list of technical solutions provided by the project's partners and relevant in the context of the present document/topics. All along the deliverable, a particular importance has been given to aspects related to the scalability of the available solutions (in terms of model storage, collaborative model management, and model traceability and interoperability).

The main purpose of this deliverable was to prepare the work for then being able to collaborate together on the specification of the Model Management & Traceability framework that has to be developed and deployed in MegaM@Rt2. This will be the main purpose of the coming 6 months and of the deliverable D4.2 (which is due to M12). The underlying goal was also to help identifying some of the key issues to be faced and tackled during the future implementation of this framework. This will be the objective of deliverables D4.3 and D4.4 which are due to M20 and M32, respectively. Among others, the following big challenges have been identified and introduced in this D4.1: scalable model storage and querying, well-synchronized and verified model views, performant and decentralized global model management, efficient integration of inter-model traceability and interoperability support.

Appendix A: Baseline Tools

This appendix provides a comprehensive list of the baseline tools provided by all the partners (who are tool providers) involved in this deliverable. They have been selected based on their relevance in the context of MegaM@Rt2 and with the topics addressed in this deliverable (cf. the different sections from 2 to 5). More technical solutions from the project's partners are available similarly from the two other state-of-the-art deliverables, namely D2.1 and D3.1, covering respectively the design and runtime aspects of MegaM@Rt2.

A.1. AM3 (and the MoScript extension)

Summary Sheet

Short Description	The goal of AM3 (AtlanMod MegaModel Management) is to provide a practical support for modeling in the large. We base this activity on the concept of a "megamodel", i.e. a model in which all model engineering resources available in given projects are registered. The basic idea is that there is no unique metamodel for megamodels. Instead, the user may use existing metamodels from a library or invent their own ones. We know the common artifacts (models, metamodels, transformations, semantic correspondences, etc.) but many others may be contextually defined: the AM3 tool is designed to be metamodel-agnostic in order to handle this contextual variability. The goal of MoScript is to complement AM3 with a textual DSL improving its model management task orchestration expressivity. With MoScript, users can automate model management tasks by means of OCL based scripts. For instance, user may write queries (based on model content, structure, relationships, and behavior derived through on-the-fly simulation) to retrieve models from model repositories, manipulate them (e.g., by running transformations on sets of models), and store them back in the repository. MoScript also allows to populate and update the megamodel automatically by doing reverse engineer of simple modeling artifact repositories.
License	Eclipse Public License - v1.0
Documentation Resources	https://wiki.eclipse.org/AM3 https://wiki.eclipse.org/MoScript
Source Code	Cf. links right above.
Maturity	Research Prototype (legacy, no recent developments)
Contact	hugo.bruneliere@imt-atlantique.fr

Decription

The AM3 prototype has been initially developed in the context of the MODELPLEX IST-FP6 European project. From the prototype's Wiki page (cf. the link provided in the table right before), plenty

of resources can be found describing both the overall approach and the corresponding technical solution: project deliverables, research papers, source code, demos, etc. This notably include the AM3 core metamodel of megamodel as well as different metamodel extensions realized to support more advanced traceability features, in the context of (ATL) model transformations notably. The Figure hereafter shows the AM3 Megamodeling perspective opened in an Eclipse workbench, coming with its Megamodel Navigator as well as related Model- and Model Element-level traceability views.

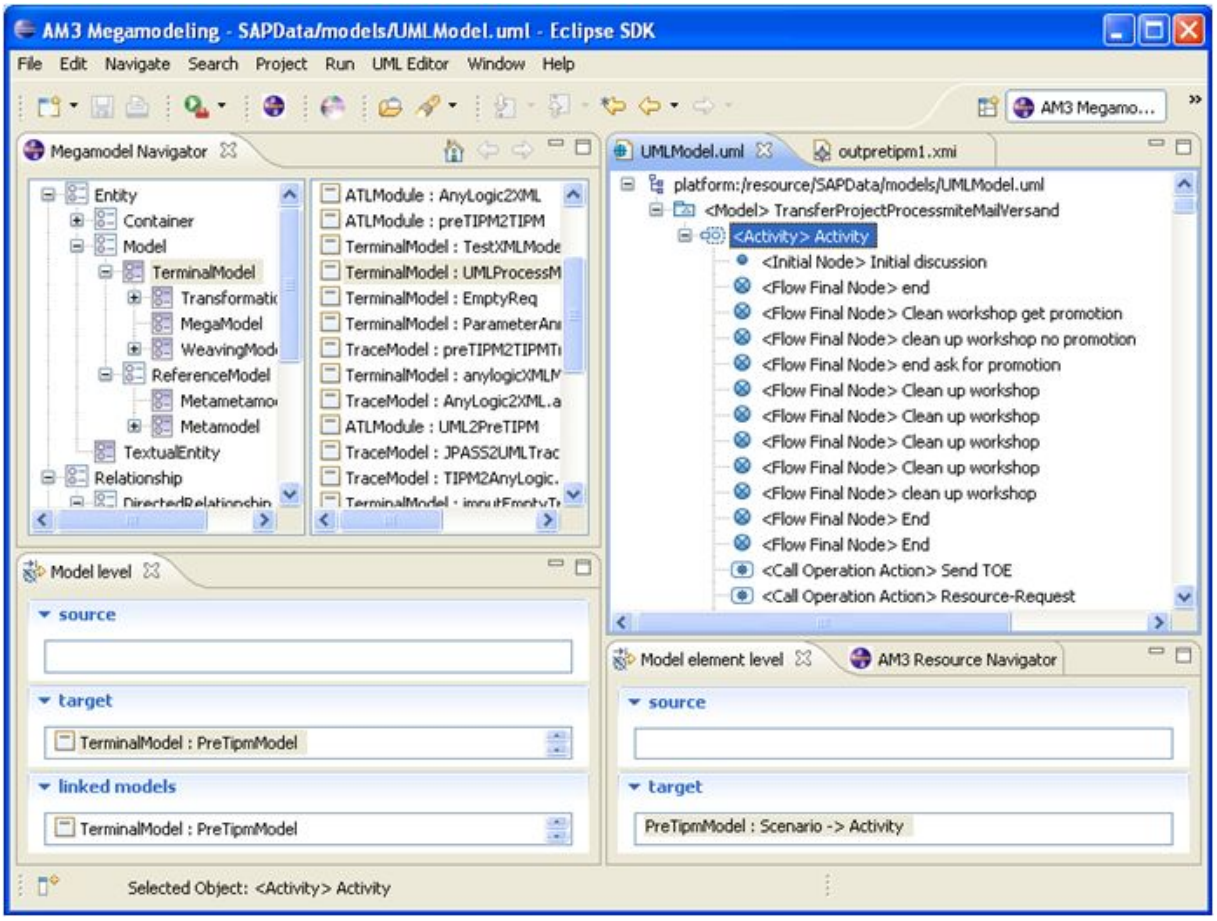

Figure 14: AM3 Megamodeling Base Environment

As stated earlier, in addition to the base megamodeling features, some more advanced support has been developed in order to provide automated megamodel discovery capabilities as well as ATL model transformation execution and traceability features. The Figure hereafter shows the dedicated ATL Transformation editor that allows specifying, triggering and then tracing the execution of ATL model-to-model transformations on models registered within a given megamodel.

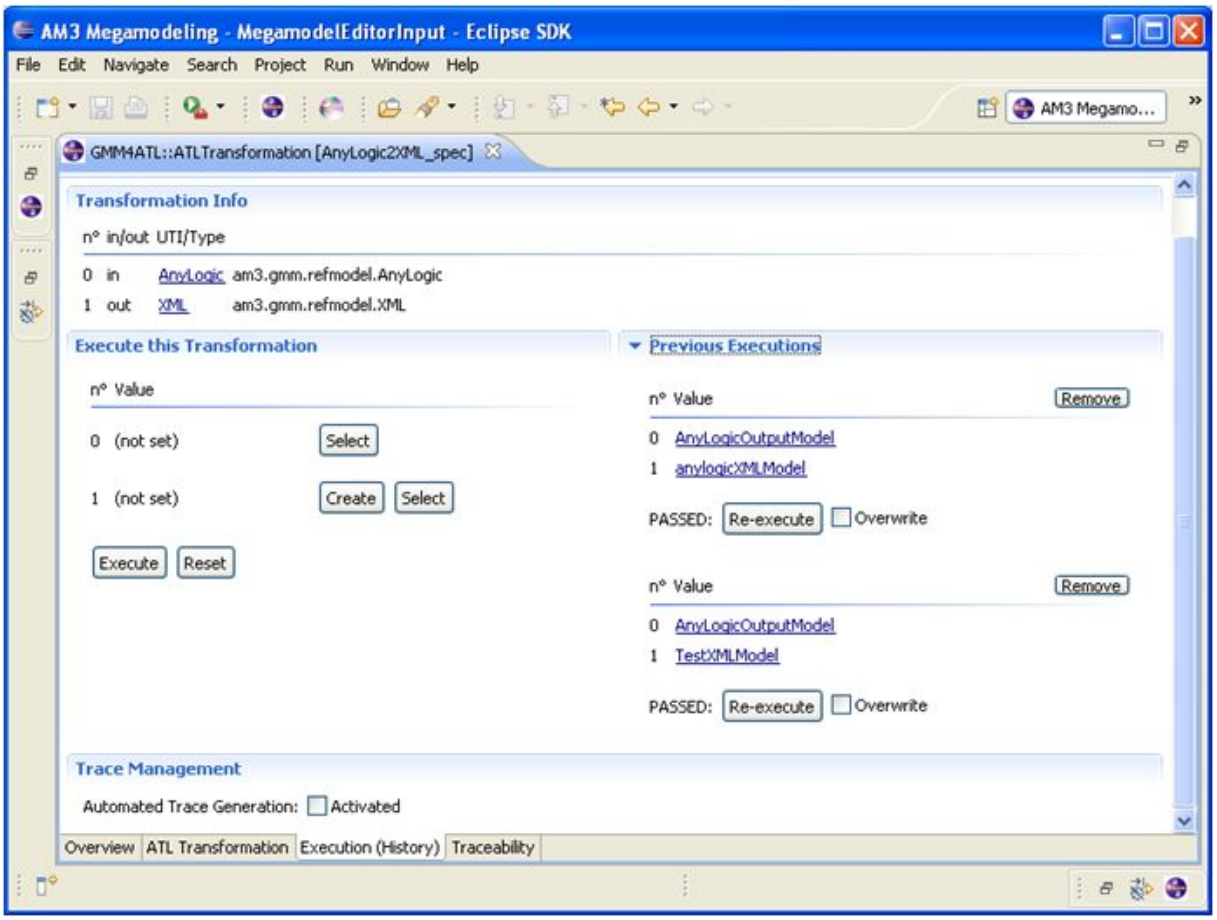

Figure 15: AM3 Megamodeling Editor for Transformation & Traceability

The MoScript DSL has been added to the AM3 Megamodeling features in order to provide users with an easy way to specify, execute and share scripts that manipulate the (modeling) artifacts described in an AM3 megamodel. The Figure hereafter shows a simple example of such a MoScript program retrieving a model transformation from a given megamodel and launching the execution of this transformation on a input model also retrieved from this same megamodel.

```
MoScriptPrgm.mscr
program MoScriptDemo

using {
    tr  : !Transformation = OclUndefined
    fam : !Model = OclUndefined
    per : !Model = OclUndefined
    models : Map(String, OclAny) = OclAny
    names : Sequence(String) = OclAny
}

do {
    -- Find transformation
    tr <- thisModule.getTransfById('/Families2Persons/Transformation/Familie

    -- Find model
    fam <- thisModule.getModelById('/Families2Persons/Families/Sample/sample

    -- Run transformation
    models <- tr.applyTo(Map{('IN', fam)});

    per <- models->get('OUT');

    -- Query resulting model
    names <- per.allContentInstancesOf('Person')->collect(e | e.fullName);
```

Figure 16: Example of a MoScript Program

More resources on MoScript, and notably the complete language definition and related tooling support, are available from the link provided in the table before.

A.2. Constellation (Modelio)

Summary Sheet

Short Description	Modelio Constellation enables models/projects governance and centralized administration, the establishment of indicators and the definition of automated procedures - reports, audit, code generation and continuous integration, document portal update… Constellation supports enterprise federations, and repositories federation whatever their heterogeneity. The web administration interface provides a convenient management tool to project managers or repository administrators. Capabilities: • Models & projects governance • Federated enterprises: support of complex organizations • Global traceability (organization/system scale) • Models sharing and reuse • Participants and models organization • Easy models/projects administration • Flexibility for distributed teams and repositories, cloud and nomadic work
License	Proprietary

Documentation Resources	https://www.modeliosoft.com/en/products/modelio-constellation.html
Source Code	Proprietary
Maturity	TRL9 (Commercial on the market)
Contact	andrey.sadovykh@softeam.fr

Decription

Modelio's new "Constellation" repository technology is based upon two widespread approaches supporting cooperation and information and contribution sharing:

- The web, whose omnipresence and flexibility are required by everyone.
- The distribution of models through "fragment" . This approach is widely used for code in open source projects and software development, which are a major source of sharing and re-use of existing "librairies" or "components".

In the following example, two models, "Model 1" and "Model 2", are composed of three available fragments. Note that the Fragment 2 model fragment is referenced by both Model 1 and Model 2.

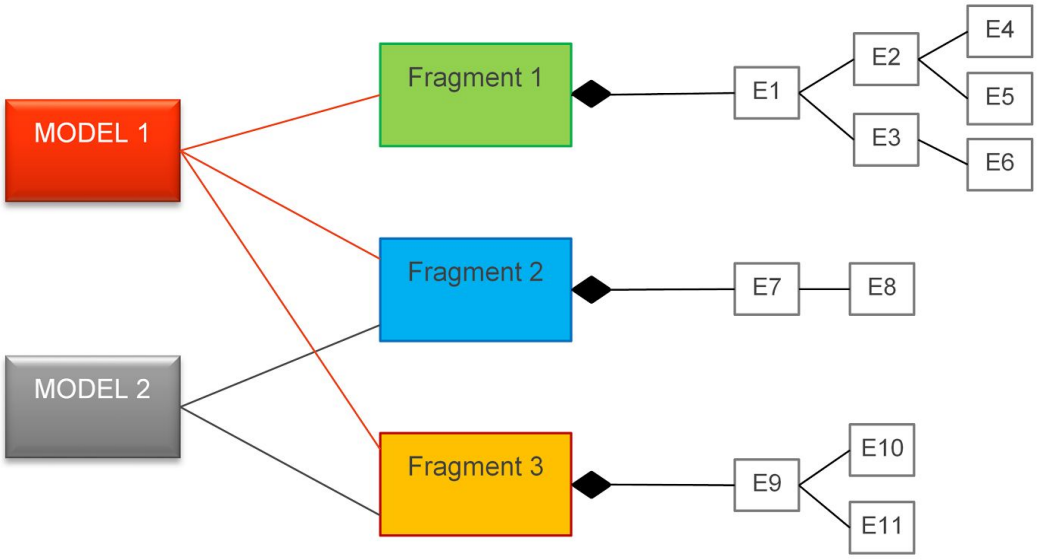

Figure 17: An example of models, fragments, and model elements

Like the web, there is no "central server" with Constellation. Model elements belong to a fragment and models reference these fragments. As depicted below, fragments are tightly coupled with repositories and repositories are where model elements are physically stored.

Figure 18: Model fragment and Repository

Modelio supports several repository technologies. Each technology has specific characteristics that must be understood when choosing one of them for a particular repository.This organization is highly decentralized, which allows the most open and agile cooperation modes.

State	Visible	Joinable	Modifiable	Comments
New	No	No	No	Any user
Active	Yes	Yes	Yes	Enrolled users only
Frozen	Yes	Yes	No	Enrolled users only
Closed	No	No	No	Any user

Table 3: Repository technologies

A.3. EMF Views

Summary Sheet

Short Description	EMF Views is a solution based on the idea of adapting the concept of database views to models. it directly relies on a generic model virtualization framework, that can be connected to language(s) enabling users to specify model views grouping elements coming from different models. Thanks to this model virtualization backend, the produced views behave as any other regular models and are not materialized but rather computed on-demand. EMF Views is also about providing language support for the user definition of model views, that allows a better view specification sharing between both users and tools. A SQL-like language has already been developed to this intent. Another specialization of the tool has also been implemented in order to deal with the particular problem of (meta)model extension, notably proposing a dedicated metamodel extension language.
License	Eclipse Public License - v1.0

Documentation Resources	http://atlanmod.github.io/emfviews
Source Code	https://github.com/atlanmod/emfviews
Maturity	Research Prototype (with new ongoing developments)
Contact	hugo.bruneliere@imt-atlantique.fr

Decription

EMF Views allows specifying viewpoints gathering concepts from one or several existing metamodels, and obtaining related views on corresponding sets of models. In order to do so, it relies on a generic model virtualization API that is applied similarly at both metamodel and model levels (to represent viewpoints and view respectively). The Figure hereafter shows the overall architecture of the EMF Views solution as well as the different artifacts which are involved in a given viewpoint/view creation and manipulation process.

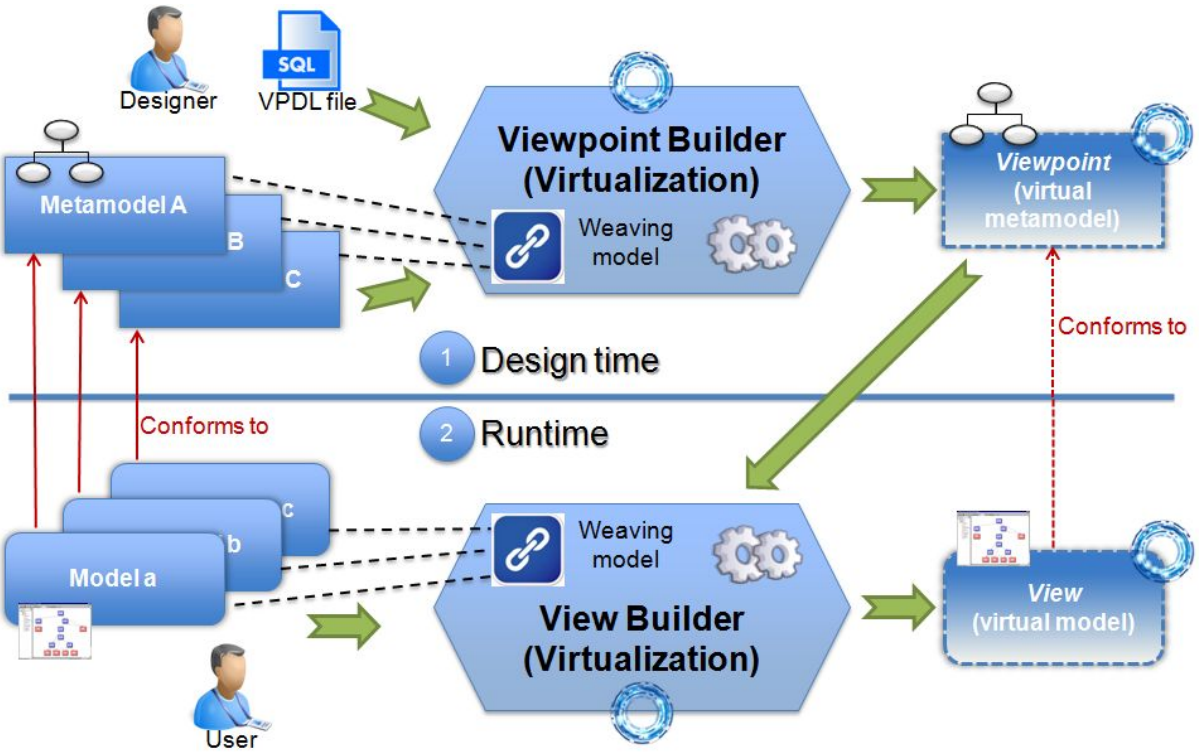

Figure 19: Overview of EMF Views and its underlying Virtualization API

From a ViewPoint Definition Language - VPDL file (cf. what comes next), a virtual metamodel representing the viewpoint is computed automatically based on the contributing metamodels. This virtual metamodel / viewpoint is then used to obtain a corresponding virtual model / view from a set of corresponding contributing models. In both cases, the required extra-data (and notably the inter-model relationships) is stored in a separate weaving model. The Figure hereafter shows an example of view, built with EMF Views, that combines elements coming from three different models (and concepts

coming from three different metamodels): namely TOGAF - Enterprise Architecture, BPMN - Business Process and ReqIF - Requirements.

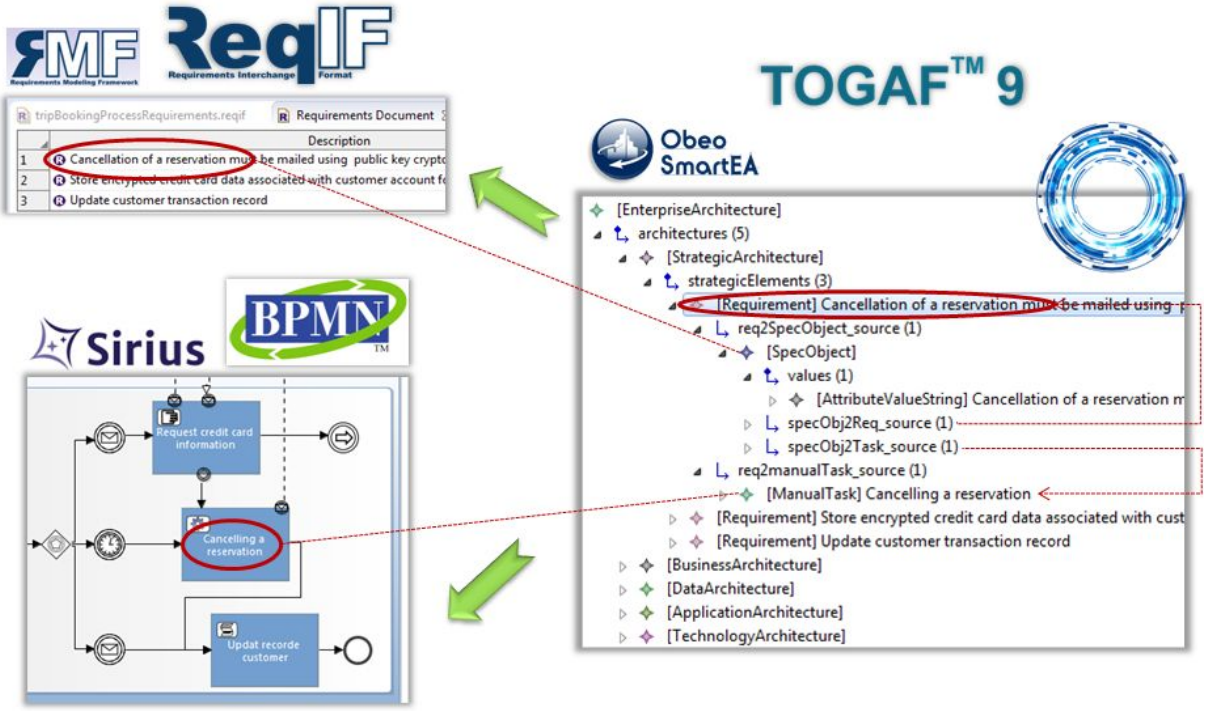

Figure 20: Example of a view in EMF views combining TOGAF, BPMN and ReqIF models

On top of EMF views, it is possible to design and plug different languages that facilitate the specification and computation of viewpoints and corresponding views. A first version of a SQL-like language has already been designed and related tooling support developed. It is called ViewPoint Definition Language or VPDL. The Figure hereafter shows how the viewpoint described right before can be expressed using VPDL.

```
create view myEnterpriseArchitectureViewpoint on
    "http://www.obeonetwork.org/dsl/togaf/contentfwk/9.0.0" as TOGAF,
    "http://www.omg.org/spec/BPMN/20100524/MODEL-XMI" as BPMN,
    "http://www.omg.org/spec/ReqIF/20110401/reqif.xsd" as REQIF
select TOGAF.Process{name},
       BPMN.Process{processType, processCriticality},
       TOGAF.Requirement{rationale}, REQIF.SpecObject{longName}
from TOGAF.Process join BPMN.Process as detailedProcess,
     TOGAF.Requirement join REQIF.SpecObject as detailedRequirement
where TOGAF.Process.name = BPMN.Process.name
      and TOGAF.Process.isAutomated = false
      and REQIF.SpecObject.values
                    ->exists(v | v.theValue = TOGAF.Requirement.name)
```

Figure 21: Example of a viewpoint definition in VPDL

More resources on EMF Views and notably the prototype's source code, its general approach, the VPDL language definition and related tooling support, are available from the links provided in the table before.

A.4. JTL

Summary Sheet

Short Description	Janus Transformation Language (JTL) is a constraint-based and relational model transformation language specifically tailored to support bidirectionality, change propagation and traceability.
License	Eclipse Public License - v 1.0
Documentation Resources	jtl.di.univaq.it
Source Code	jtl.di.univaq.it
Maturity	Research prototype
Contact	Romina Eramo romina.eramo@univaq.it

Decription

Janus Transformation Language (JTL) [CICCHETTI ET AL. 2010; ERAMO ET AL. 2015] is a constraint-based and relational model transformation language specifically tailored to support bidirectionality and change propagation.

Most current bidirectional approaches, such as like QVT-R and TGGs, prescribe writing a bidirectional transformation as a declarative consistency relation between two metamodels. Since most interesting examples of bidirectional transformations are non-bijective, restoring consistency is an inherently ambiguous process. In fact, reversing non-injective mappings may give place to multiple choices, requiring that the transformation designer decide on a general consistency-restoration strategy at design-time. However, when this is not possible, current tools typically consider only one particular strategy out of the many possible alternatives, of which developers have little or no control. For instance, a process model can be transformed into an unordered set of activities. Then, adding a new activity to the set cannot be univocally back-propagated to the process model, as there are multiple ways to accommodate the activity in the process. A possible consistency-restoration strategy could be to append the activity to the end or beginning of the process. Omitting such a general rule renders the result almost unpredictable with current tools.

Recently, novel approaches to bidirectionality have been proposed for dealing with non-deterministic transformations. Among them, JTL provides a relational semantics relies on Answer Set Programming (ASP) [GELFOND ET AL, 1988]: given a change to one source, JTL uses the DLV [LEONE ET AL, 2004] constraint solver to find a consistent choice for the other source; there might be multiple choices. Thus, the responsibility of choosing the right model among the generated ones is left to the designer.

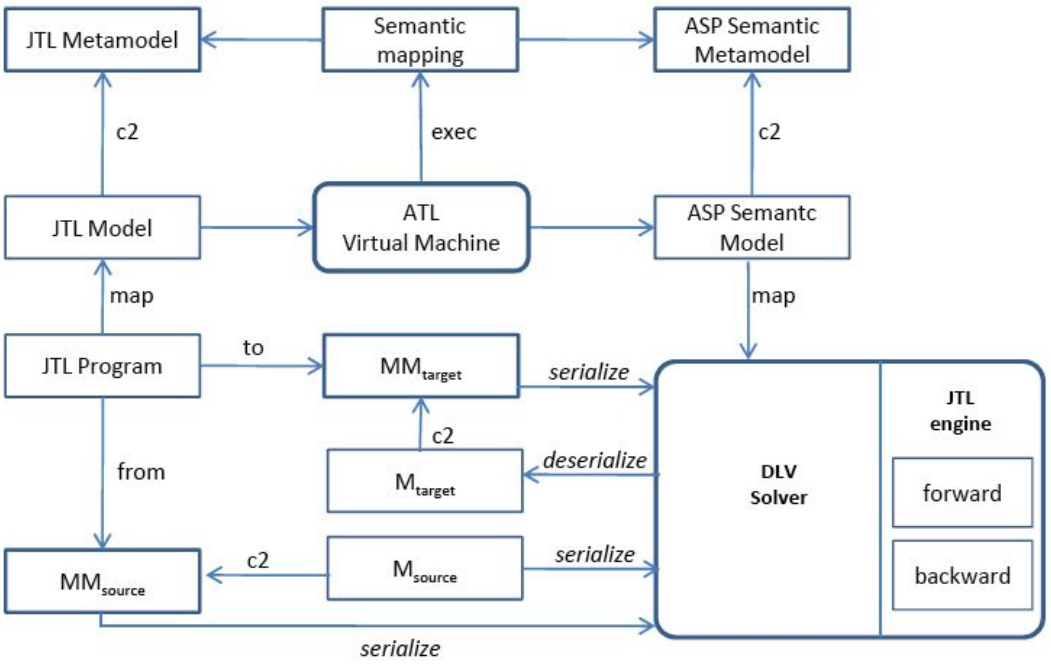

Figure 22: Overview of the JTL approach

In JTL, the tracing information mechanism stores relevant details about the linkage between source and target model elements at execution-time (including the applied transformation rules). To better understand how JTL works, let us consider the following definitions:

Definition. A *Trace link l* is a relationship between one or more source model elements and one or more target model elements; i.e., it describes how a target element has been generated starting from a source one by applying a transformation rule during the execution process. It consists of a tuple $l = (e_m , e_n , r)$ with em and en model elements in m in M and n in N, respectively, such that en is obtained from e_m by applying the rule r in R.

Definition. A *Trace model t* is a structured set of trace links. One or more trace models are generated as a result of model transformation, both forward and backward. In particular, in case of non-deterministic transformation, for each generated target model a correspondent trace model is obtained. Each element in the trace model represents the linkage describing how a target element has been generated starting from a source one.

Within JTL, the traceability mechanism is an intrinsic characteristic of the ASP-based engine. Trace links are extrapolated during the transformation execution and made explicit by the framework. Thus, trace models are explicit and maintained as models conforms to the JTL Trace Metamodel, as defined in its Ecore format within EMF. Trace models can be stored, viewed and manipulated (if needed) from the designer. Within JTL, trace models are re-used during the transformation execution. In particular, trace model, can be given as input of the transformation in order to (re-) establish consistency, manage ambiguities and guarantee the correctness of the transformation.

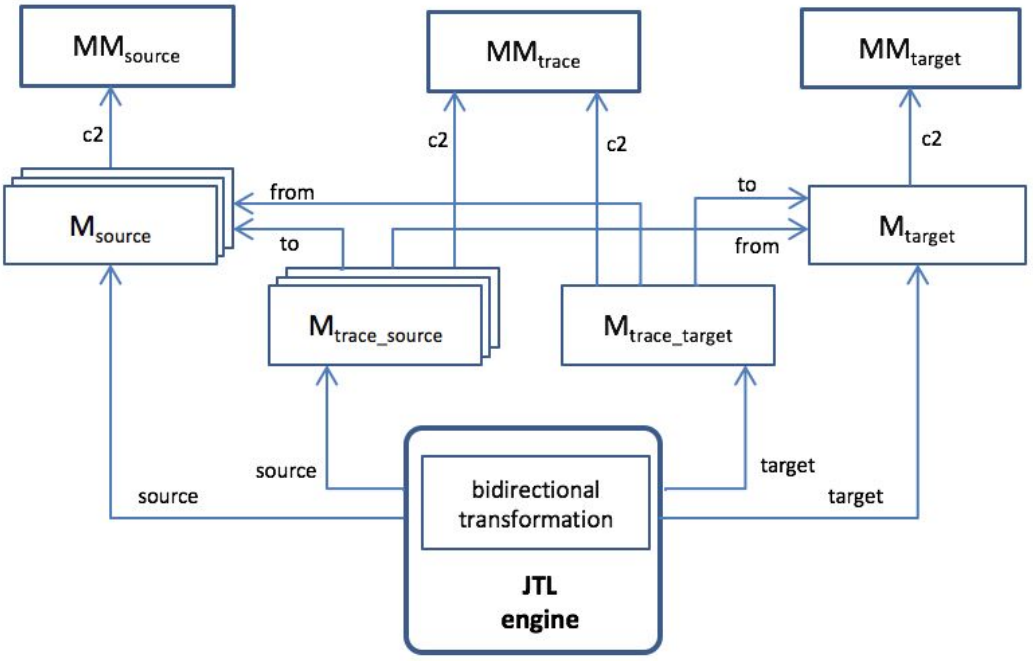

Figure 23: Traces in the JTL approach

A.5. NeoEMF

Summary Sheet

Short Description	NeoEMF is a persistence framework designed to take benefit of the scalability of NoSQL databases to handle very large models efficiently. NeoEMF aims at providing low-level model representations that fit a given modeling scenario (for example interactive editing, complex queries, or model transformations). The framework currently supports three model-to-NoSQL database mappings (graph, map, and wide column), and relies on a lazy-loading architecture to optimize memory consumption.
License	Eclipse Public License - v1.0
Documentation Resources	www.neoemf.com https://github.com/atlanmod/NeoEMF/wiki
Source Code	https://github.com/atlanmod/NeoEMF
Maturity	Research Prototype
Contact	neoemf@googlegroups.com

Decription

NeoEMF is a multi-database model persistence framework that aims to provide the appropriate model-to-database mapping according to a given modeling scenario. The framework provides three built-in NoSQL implementations, each one optimized for a specific modeling task:

- NeoEMF/Map: a key-value store connector that is designed to efficiently compute repeated atomic accesses on a model, which are typically generated by high-level modeling API. The connector is implemented for MapDB and BerkeleyDB, and has shown positive results in terms of execution time and memory consumption compared to existing persistence solutions (GOMEZ et. al. 2015).
- NeoEMF/Graph: a graph database connector that takes benefit of the rich query languages usually provided by this database family to compute complex model queries efficiently. This connector is the basis of the Mogwaï query framework (DANIEL et. al. 2016), that computes OCL expressions by generating low-level graph database queries, bypassing the modeling API and improving execution time and memory consumption.
- NeoEMF/Column: a wide-column store connector that provides advanced distribution and concurrency capabilities, enabling the development of distributed MDE-based applications.It exploits the wide availability of distributed clusters in order to distribute intensive read/write workloads across datanodes. The distributed nature of this persistence solution is used in the ATL-MR (BENELALLAM et al. 2015) tool, a distributed implementation of the ATL engine based on MapReduce.

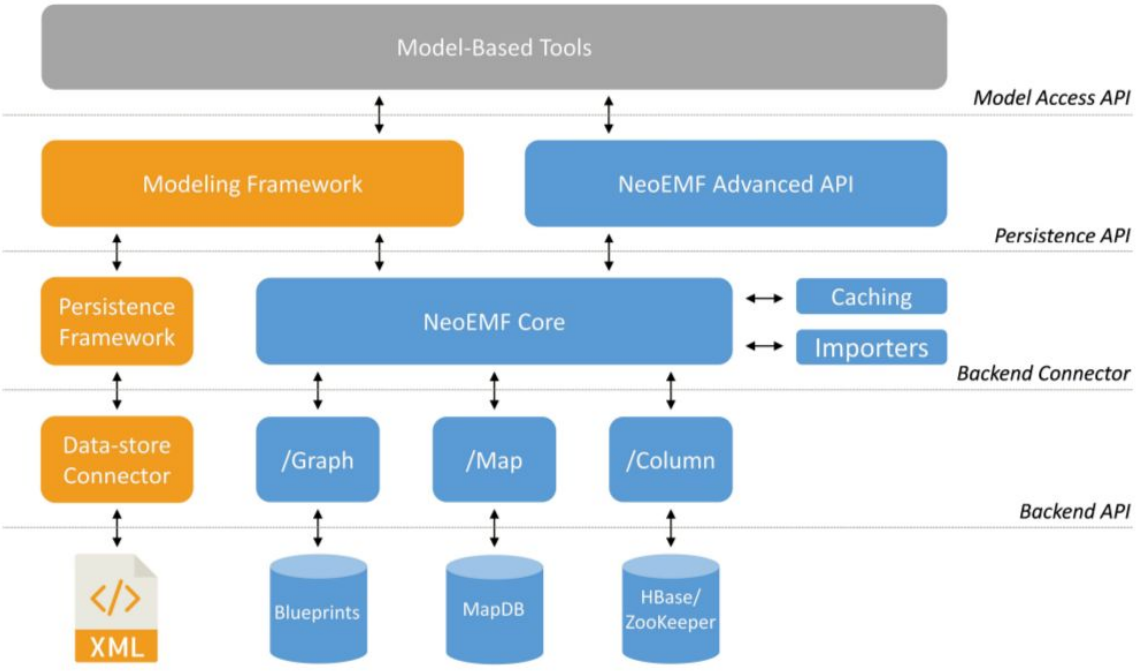

Figure 24: NeoEMF Integration in the Modeling Ecosystem

The figure right before describes the architecture of NeoEMF in a typical modeling environment. Modelers typically manipulate models using *Model-based Tools*, which provide high-level modeling features such as a graphical interface, interactive console, or query editors. These features internally rely on a *Model Access API* to navigate the models, perform CRUD operations, check constraints, etc. The modeling framework delegates the operations to a persistence manager using its *Persistence API*, which is in charge of the (de)serialization of the model. This *Persistence API* can be complemented by a low-level connector that interacts directly with the data-store API. This generic

architecture is used in popular modeling frameworks such as EMF and KMF (FOUQUET et. al. 2012) which typically provide a default XML connector to store models.

The NeoEMF *core* component implements the *Persistence API*, and provides a set of methods allowing the modeling framework to interact with it as a regular persistence layer. This design makes NeoEMF both transparent to the client application, and the modeling framework itself, that simply delegates the calls without taking care of the actual storage.

Once the NeoEMF *core* component has received the request of the modeling operation to perform, it forwards the operation to the appropriate *Backend Connector* (*/Map*, */Graph*, or */Column*), which is in charge of handling the low-level model-to-database mapping of the model. These connectors translate modeling operations into *Backend API* calls, store the results, and reify database records into high-level modeling framework elements when needed. NeoEMF also embeds a set of default caching strategies that are used to improve performance of client applications, and can be configured transparently at the EMF API level.

NeoEMF is open source and publicly available on Github[9]. The framework is released as a set of maven artifacts, and an update site is available to ease the tool integration in Eclipse-based platforms.

A.6. PADRE

Summary Sheet

Short Description	PADRE (Performance Antipatterns Detection and Refactoring with Epsilon) is a model refactoring tool that enables performance problems detection and solution within an unique supporting environment. Performance problems are represented by performance anti-patterns (SMITH et al., 2001; SMITH et al., 2002; SMITH et al., 2003), and possible solutions to them, in terms of design model refactoring actions, are enabled with the aim of removing anti-patterns and improving performance. The Epsilon Epsilon Platform (KOLOVOS et al., 2010) represents the unifying environment supporting the performance optimization process. In particular, PADRE relies on some task-specific languages of the Epsilon Platform to codify (i) declarative conditions that can be used to notify the user about performance problems occurrences, and (ii) imperative blocks that support software model refactoring based on violations of those conditions. The different execution semantics of each language enable different supports to model refactoring, i.e. different scenarios on the trade-off between automation and human participation to the refactoring process (ARCELLI ET AL., 2017).
License	Eclipse Public License - v1.0
Documentation Resources	Not available yet

[9] https://github.com/atlanmod/NeoEMF

Source Code	https://github.com/SEALABQualityGroup/performance-driven-software-model-refactoring-framework
Maturity	Research prototype
Contact	Daniele Di Pompeo (daniele.dipompeo@graduate.univaq.it) Davide Arcelli (davide.arcelli@univaq.it) Vittorio Cortellessa (vittorio.cortellessa@univaq.it)

Decription

PADRE is a model refactoring tool that enables performance problems detection and solution within an unique supporting environment. More specifically, PADRE is an Eclipse-integrated tool grounded on a subset of task-specific languages provided by the Epsilon Platform, aimed at supporting a model-based performance optimization process driven by the detection and solution of performance anti-patterns. In its current version, PADRE supports 10 refactoring actions for UML models and the detection of 7 performance anti-patterns. By exploiting the different execution semantics of three task-specific languages of Epsilon (Epsilon Pattern Language – EPL, Epsilon Validation Language – EVL, Epsilon Wizard Language – EWL), three types of refactoring sessions are provided (detailed in the following table), all aimed at detecting performance anti-patterns and applying refactoring actions on UML models, but with different scenarios on the trade-off between automation and human participation to the refactoring process.

Epsilon language	**Model refactoring support**
EPL	Batch refactoring sessions allow to sequentially execute predefined set of anti-pattern detection rules and refactoring actions. The process can be repeated once (i.e., standard mode) or until no more performance anti-pattern occurrences are found (i.e., iterative mode).
EVL	User-driven multiple refactoring sessions allow to execute interactive anti-pattern detection and refactoring sessions. A list of detected anti-pattern occurrences in the performance-oriented software model is first presented to the user; Then, the EVL engine enables a number of available refactoring actions as applicable. Each refactoring is applied to a current temporary version of the software model and, when the user stops the session, the current version is finalized and represents the session output.
EWL	User-driven single refactoring sessions are directly integrated in Eclipse-based Graphical Modeling Frameworks, e.g. Papyrus. As an element is selected in the modeling environment, anti-pattern occurrences are immediately detected with respect to the selected element type. Then, the EWL engine enables the anti-pattern solutions, among which the user can select the one to apply to the software model, thus producing

	a refactored model, which represents the session output. A subsequent element selection would trigger a new refactoring session.

Table 4: Refactoring support provided by PADRE

Finally, PADRE also supports the automated translation of the code for performance anti-patterns detection rules and refactoring actions among EPL, EVL, and EWL, based on a set of mappings at level of language abstract syntax. This feature reduces the effort in Epsilon code writing. In fact, it allows the user to manually write code only for one of the three languages (e.g., EPL), and then to generate the code for the other two in order to exploit the different kinds of refactoring support that their execution semantics provide.

In MegaM@Rt2, PADRE will be used in its current version for supporting the performance feedback loop for UML design models with MARTE annotations. Given its extensible nature, PADRE will be extended to other modelling notations involved in MegaM@Rt2.

A.7. MDEForge

Summary Sheet

Short Description	MDEForge is an extensible Web-based modeling platform specifically conceived to foster a community-based modeling repository, which underpins the development, analysis and reuse of modeling artifacts. Moreover, it enables the adoption of model management tools as software-as-a-service that can be remotely used without overwhelming the users with intricate and error-prone installation and configuration procedures.
License	Eclipse Public License - v1.0
Documentation Resources	https://github.com/MDEGroup/MDEForge/wiki (still at early stages)
Source Code	https://github.com/MDEGroup/MDEForge
Maturity	Research prototype
Contact	Davide Di Ruscio (davide.diruscio@univaq.it) Juri Di Rocco (juri.dirocco@univaq.it) Alfonso Pierantonio (alfonso.pierantonio@univaq.it)

Decription

MDEForge consists of a number of services that can be used by means of both a Web access and programmatic interfaces (API) that enable their adoption as software as a service. In particular, core services are provided to enable the management of modeling artifacts, namely transformations, models, metamodels, and editors. Atop of such core services, extensions can be developed to add new functionalities.

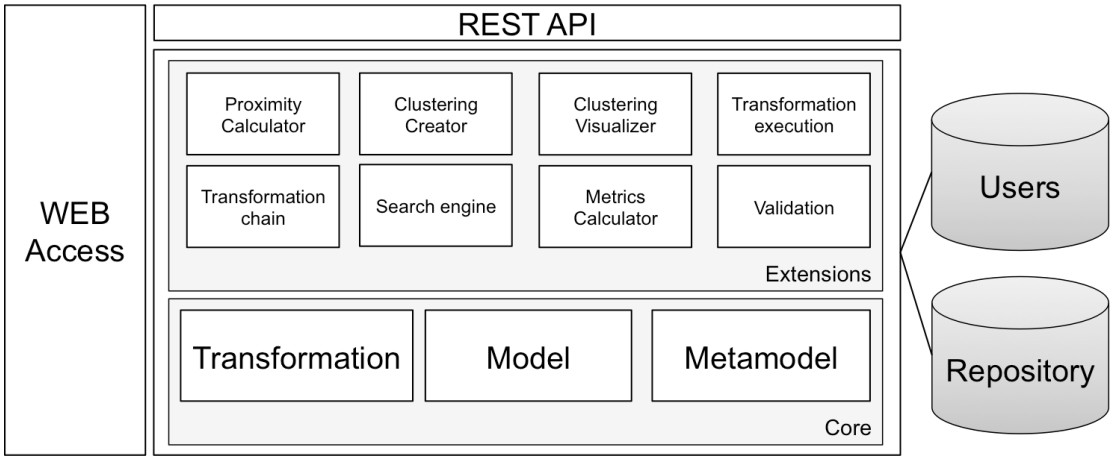

Figure 25: Overview of the MDEForge approach

MDEForge has been designed for:

- *Developers of modeling artifacts*: we envision a community of users that might want to share their tools and enable their adoption and refinement by other users. To this end the platform provides the means to add new modeling artifacts to the MDEForge repository.
- *Developers of MDEForge extensions*: one of the requirements that was identified when started the development of MDEForge is about the modularity and extensibility of the platform. To this end a set of core services was identified and that can be used to add new functionalities by means of platform extensions. In this respect, experienced users might contribute by proposing new extensions to be included in the platform.
- *End-users*: A Web application enables end-users to search and use (meta)models, transformations, and editors available in the MDEForge repository. Experienced users might use the REST API to exploit the functionalities provided by the platform in a programmatic way. For instance, tool vendors might exploit the functionalities provided by their tools by exploiting some of the transformations available in the MDEForge repository.

The Repository component shown in Figure 25 plays a key role in the MDEForge platform and it has been developed in order to store artifacts according to the metamodel shown in Figure 26. In particular, the repository has been developed with the aim of managing any kinds of modeling artifacts (see the metaclass Artifact in Figure 26). Each artifact refers to the corresponding type, e.g., model, transformation, metamodel, etc. The specification of the relation between a given artifact and the corresponding type is done by means of the Relation elements. In turn, each relation is typed by means of a corresponding *RelationType* element. By means of such modeling constructs it is possible e.g., to specify the conformsTo relation between a model m1 and the corresponding metamodel MM1 as shown in Figure 27. Similarly, it is possible to specify any kinds of modeling elements together with their relations. For example, it is possible to represent also the execution engine of a given model transformation stored in the repository.

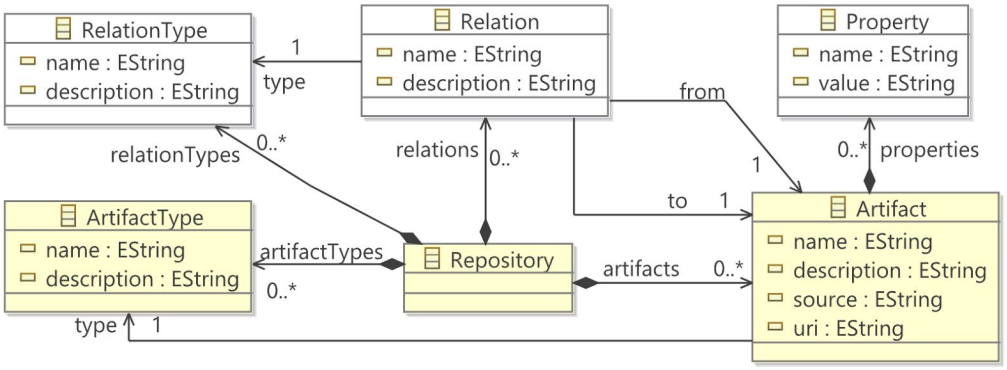

Figure 26: Fragment of the MDEForge metamodel

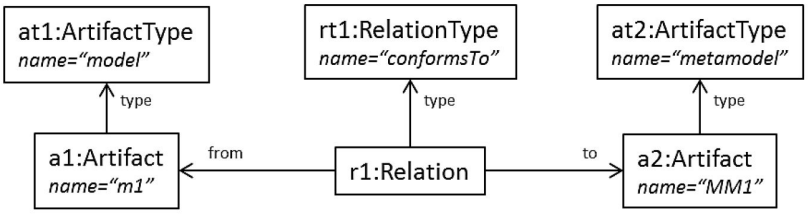

Figure 27: Simple content of the MDEForge Repository

References

ACCELEO PROTRACEABILITY. http://www.acceleo.org/pages/additionnal_products/en (2009)

ABITEBOUL, S.; BONNER, A. (1991). Object and Views. ACM SIGMOD Record, 20(2):238–247.

AIZENBUD-RESHEF, N., NOLAN, B., RUBIN, J., SHAHAM-GAFNI, Y.: Model traceability. IBM Syst. J. 45(3), 515–526 (2006)

ALHAJ, M.; PETRIU, D.C. (2013). Traceability Links in Model Transformations between Software and Performance Models. In Model-Driven Dependability Engineering: 16th International SDL Forum (SDL). Springer, 2013, pages 203–221.

ALLILAIRE, F.; BEZIVIN, J.; BRUNELIERE, H.; JOUAULT, F. (2006) Global Model Management in Eclipse GMT/AM3. In Eclipse Technology eXchange workshop (eTX) - a ECOOP 2006 Satellite Event.

ANGLES, R.; GUTIERREZ, C. (2008). Survey of Graph Database Models. ACM Computing Surveys, 40(1):1–39.

ANJORIN, A.; ROSE, S.; DECKWERTH, F.;SCHURR. (2014). Efficient Model Synchronization with View Triple Graph Grammars. In ECMFA 2014.

ARCELLI, D.; CORTELLESSA, V.; TRUBIANI, C. (2012). Antipattern-based model refactoring for software performance improvement. In Proceedings of the 8th International ACM/SIGSOFT Conference on Quality of Software Architectures (QoSA). ACM, 2012, pages 33–42.

ARCELLI, D.; CORTELLESSA, V.; DI RUSCIO, D. (2013). Applying Model Differences to Automate Performance-Driven Refactoring of Software Models. In 10th European Workshop on Computer Performance Engineering (EPEW). Springer, 2013, pages 312–324.

ARCELLI, D.; CORTELLESSA, V.; DI POMPEO, D. (2017). Performance-driven software model refactoring. In Journal of Information and Software Technology. To appear. DOI:https://doi.org/10.1016/j.infsof.2017.09.006

ATKINSON, C.; STOLL, D.; BOSTAN, P. (2010). Orthographic Software Modeling: A Practical Approach to View-Based Development. In ENASE 2010.

BARBERO M., DEL FABRO M.D., BEZIVIN J., Traceability and Provenance Issues in Global Model Management. In ECMDA-TW'07: Proc. of 3rd Workshop on Traceability, pages 47-55. SINTEF, June 2007.

BARMPIS, K.; KOLOVOS, D. (2013). Hawk: Towards a Scalable Model Indexing Architecture. In 1st BigMDE Workshop, pages 6-9. ACM.

BARMPIS, K.; KOLOVOS, D. (2012). Comparative Analysis of Data Persistence Technologies for Large-Scale Models. In 1st Extrem Modeling Workshop, pages 33-38. ACM.

BAUER, C.; KING, G (2005). Hibernate in action. Manning Greenwich CT.

BECKER, S.; KOZIOLEK, H.; REUSSNER, R.H. (2009). The Palladio component model for model-driven performance prediction. In Journal of Systems and Software 82(1):3–22.

BENELALLAM, A.; GOMEZ, A.; SUNYE, G.; TISI, M.; LAUNAY, D. (2014). Neo4EMF, a Scalable Persistence Layer for EMF models. In 10th ECMFA Conference, pages 230-241. Springer.

BENELALLAM, A.; GOMEZ, A.; TISI, M.; CABOT, J. (2015). Distributed model-to-model transformation with ATL on MapReduce. In the 8th SLE Conference, pages 37-48. ACM.

BERNARDO, M.; CIANCARINI, P.; DONATIELLO, L. (2002). Architecting families of software systems with process algebras. In ACM Transactions on Software Engineering and Methodology 11(4):386–426.

BEYHL T., HEBIG R., GIESE H., A Model Management Framework for Maintaining Traceability Links. Software Engineering (Workshops) 2013: 453-457.

BEZIVIN, J.; JOUAULT, F.; VALDURIEZ, P. (2004). On the Need for Megamodels. In 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications: Best Practices for Model-Driven Software Development workshop.

BISHOP, C. (2009). Pattern Recognition and Machine Learning. Springer.

BURGER, E. (2013). Flexible Views for Rapid Model-Driven Development. In 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling.

BURGER, E.; HEN, J.; KRUSE, S.; KUSTER, M.; RENTSCHLER, A.; HAPPE, L. (2014). ModelJoin. A Textual Domain-Specific Language for the Combination of Heterogeneous Models. Tech. Rep., Karlsruhe Institute of Technology, Faculty of Informatics.

BRUNELIERE, H.; CABOT, J.; DUPE, G.; MADIOT, F. (2014). MoDisco: A Model Driven Reverse Engineering Framework. In Journal of Information and Software Technology, 56(8):1012-1032.

BRUNELIERE, H.; PEREZ, J.G.; DESFRAY, P.; KHELLADI, D.E.; HEBIG, R.; BENDRAOU, R.; CABOT, J. (2015). On Lightweight Metamodel Extension to Support Modeling Tools Agility. In the European Conference on Modelling Foundations and Applications (ECMFA), pages 62-74. Springer.

BRUNELIERE, H.; PEREZ, J.G.; WIMMER, M.; CABOT, J. (2015). EMF Views: A View Mechanism for Integrating Heterogeneous Model. In the 34th International Conference on Conceptual Modeling (ER), pages 317-325. Springer.

CICCHETTI, A.; CICCOZZI, F,; LEVEQUE, T. (2011). A Hybrid Approach for Multi-View Modeling. In ECEASST 2011.

CICCHETTI, A.,DI RUSCIO, D., ERAMO, R., PIERANTONIO, A. JTL: a bidirectional and change propagating transformation language, in SLE10, pp. 183–202, 2010.

CORTELLESSA, V.; DI MARCO, A.; ERAMO, R.; PIERANTONIO, A.; TRUBIANI, C. (2010). Digging into UML models to remove performance antipatterns. In Proceedings of the 2010 Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems (QUOVADIS@ICSE). ACM, 2010, pages 9–16.

CORTELLESSA, V. (2013). Performance Antipatterns: State-of-art and Future Perspectives. In 10th European Workshop on Computer Performance Engineering (EPEW). Springer, 2013, pages 1–6.

CORTELLESSA, V.; DI MARCO, A.; TRUBIANI, C. (2014). An approach for modeling and detecting software performance antipatterns based on first-order logics. In Software and System Modeling, 13(1): 391–432.

DANIEL, G.; SUNYE, G.; BENELALLAM, A.; TISI, M.; VERNAGEAU, Y.; GOMEZ, A.; CABOT, J. (2016). NeoEMF: a Multi-database Model Persistence Framework for Very Large Models. In the 19th MoDELS Conference (Demonstration and Poster Sessions), pages 1-7. CEUR-WS.

DANIEL, G.; SUNYE, G.; CABOT, J. (2016). Mogwaï: a Framework to Handle Complex Queries on Large Models. In the 10th RCIS Conference, pages 225-237. IEEE.

DANIEL, G.; JOUAULT, F.; SUNYE, G.; CABOT, J. (2017). Gremlin-ATL: a Scalable Model Transformation Framework. In the 32nd ASE Conference, to appear. IEEE/ACM.

DE CARLOS, X.; SAGARDUI, G.; MURGUZUR, A.; TRUJILLO, S.; MENDIALDUA, X. (2015). Model Query Translator: a Model-Level Query Approach for Large-Scale Models. In the 3rd MODELSWARD Conference, pages 62-73. IEEE.

DE CARLOS, X.; SAGARDUI, G.; TRUJILLO, S. (2014). MQT, an Approach for Run-Time Query Translation: From EOL to SQL. In the 14th OCL Workshop, pages 13-22. CEUR.

DE SANCTIS, M., TRUBIANI, C., CORTELLESSA, V., DI MARCO, A., FLAMMINJ, M. A model-driven approach to catch performance antipatterns in ADL specifications. Information & Software Technology 83: 35-54 (2017)

DIAZ-PACE, A.; KIN, H.; BASS, L.; BIANCO, P.; BACHMANN, F. (2008). Integrating Quality-Attribute Reasoning Frameworks in the ArchE Design Assistant. In Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures (QoSA). Springer-Verlang, 2008, pages 171–188.

DIDONET DEL FABRO, M.; VALDURIEZ, P. (2009). Towards the Efficient Development of Model Transformations using Model Weaving and Matching Transformations. Software and Systems Modeling, 8:305–324.

DI ROCCO, J., DI RUSCIO, D., IOVINO, L., PIERANTONIO, A., 2015. Collaborative Repositories in Model-Driven Engineering (2015), in: IEEE Software, 32:3(28-34)

ECLIPSE FOUNDATION. Eclipse model-to-model transformation. http://www.eclipse.org/m2m/. Last accessed: June 2009

EMF FACET (2017). The Eclipse EMF Facet project. Accessed July 07, 2017. https://projects.eclipse.org/projects/modeling.emft.emf-facet

ENOIU, E.P., CAUSEVIC, A., OSTRAND, T.J., WEYUKER, E.J., SUNDMARK, D. and PETTERSSON, P., 2016. Automated test generation using model checking: an industrial evaluation. International Journal on Software Tools for Technology Transfer, 18(3), pp.335-353.

ENOIU EP, SUNDMARK D, PETTERSSON P. Model-based test suite generation for function block diagrams using the UPPAAL model checker. In Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Mar 18, 2013 (pp. 158-167). IEEE.

ERAMO, R., PIERANTONIO A., ROSA, G., Managing uncertainty in bidirectional model transformations, in Procs of the Int. Conference on Software Language Engineering, SLE 2015, pp. 49–58, 2015.

FAVRE, J.M.; NGUYEN, T. (2005) Towards a Megamodel to Model Software Evolution through Transformations. In Electronic Notes in Theoretical Computer Science 127(3):59-74.

FOUQUET, F.; NAIN, G.; MORIN, B.; DAUBERT, E.; BARAIS, O.; PLOUZEAU, N.; JEZEQUEL, J-M. (2012). An Eclipse Modeling Framework Alternative to Meet the Models@Runtime Requirements. In the 15th MoDELS Conference, pages 87-101. Springer.

GARCES, K.; JOUAULT, F.; COINTE, P.; BEZIVIN, J. (2009). A Domain Specific Language for Expressing Model Matching. In IDM 2009.

GELFOND M., LIFSCHITZ V. The Stable Model Semantics for Logic Programming. In Procs of ICLP, pages 1070–1080, 1988.

GIESE, H., WAGNER, R.: Incremental model synchronization with triple graph grammars. In: MoDELS. LNCS, vol. 4199. Springer, Berlin (2006)

GOLDSCHMIDT, T.; BECKER, S.; BURGER, E. (2012). Towards a Tool-Oriented Taxonomy of View-Based Modelling. In Modellierung 2012.

GOMEZ, A.; SUNYE, G.; TISI, M.; CABOT, J. (2015) Map-based Transparent Persistence for Very Large Models. In the 18th FASE Conference, pages 19-34. Springer.

GOMEZ, A.; BENELALLAM, A.; TISI, M. (2015) Decentralized Model Persistence for Distributed Computing. In the 3rd BigMDE Workshop, pages 42-51. CEUR-WS.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. (2016). Deep Learning. MIT Press.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics.

HEBIG, R.; SEIBEL, A.; GIESE, H. (2012) On The Unification of Megamodels. Electronic Communications of the EASST 45.

HEGEDUS, A.; HORVATH, A.; RATH, I.; VARRO, D. (2012). Query-Driven Soft Interconnection of EMF Models. In MODELS 2012.

HEGER, C.; VAN HOORN, A.; MANN, M.; OKANOVIC, D. (2017) Application Performance Management: State of the art and Challenges for the Future. In Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE). ACM, 2017, pages 429–432.

IEC. “International Standard on 61131-3 Programming Languages”. In: Programmable Controllers. IEC Library, 2014

JAKOB, J.; KONIGS, A.; SCHURR, A. (2006). Non-materialized Model View Specification with Triple Graph Grammars. In ICGT 2006.

JOUAULT, F.: Loosely coupled traceability for ATL. In: Proc. ECMDA Workshop on Traceability (2005)

JOUAULT, F., ALLILAIRE, F., BÉZIVIN, J., KURTEV, I., VALDURIEZ, P.: ATL: a QVT-like transformation language. In: OOPSLA Companion, pp. 719–720 (2006)

KEITH STOUFFER, JOE FALCO, and KAREN SCARFONE. "Guide to Industrial Control Systems (ICS) Security". In: Special Report. Vol. 800. 82. NIST, 2011, pp. 16–16.

KITALPHA (2017). "The Eclipse Kitalpha project". Accessed July 31, 2017. https://polarsys.org/kitalpha/

KLING W.; JOUAULT, F.; WAGELAAR, D.; BRAMBILLA, M.; CABOT, J. (2011) MoScript: A DSL for Querying and Manipulation Model Repositories. In International Conference on Software Language Engineering (SLE 2011), pages 180-200. Springer.

KOLOVOS, D.; PAIGE, R.; POLACK, F. (2006). The Epsilon Object Language (EOL). In ECMDA-FA 2006, pages 128-142. Springer.

KOLOVOS D. (2009) Establishing Correspondences between Models with the Epsilon Comparison Language. In ECMFA 2009.

KOLOVOS, D.; PAIGE, R.; POLACK, F. (2006). Merging Models with the Epsilon Merging Language. In MODELS 2006.

KOLOVOS, D.; ROSE, L.; MATRAGKAS, N.; PAIGE, R.; GUERRA, E.; CUADRADO, J.; DE LARA, J.; RATH, I.; VARRO, D.; TISI, M.; CABOT, J. (2013). A Reasearch Roadmap Towards Achieving Scalability in Model Driven Engineering. In 1st BigMDE Workshop, pages 1-10, ACM.

KOLOVOS, D.; ROSE, L.; PAIGE, R.; GARCIA-DOMINGUEZ, A. (2010). The epsilon book. Structure, vol. 178:1–10.

KOLOVOS, D.S., PAIGE, R.F.: Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon), http:// www.eclipse.org/gmt/epsilon, 2008

KOLOVOS, D.S., PAIGE, R.F., POLACK, F.: On-demand merging of traceability links with models. In: Proceedings of the ECMDA Workshop on Traceability (2006)

KOLLER, D.; FRIEDMAN, N. (2009). Probabilistic Graphical Models: Principles and Techniques. MIT Press.

KOZIOLEK, H. (2010). Performance Evaluation of Component-Based Software Systems: A Survey. Performance Evaluation. 67(8):634–658.

KRAMER, M. E.; BURGER, E.; LANGHAMMER, M. (2013). View-centric Engineering with Synchronized Heterogeneous Models. In 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling.

KURTEV, I., DEE, M., GOKNIL, A., VAN DEN BERG, K.: Traceability based change management in operational mappings. In: Proceedings of the ECMDA Workshop on Traceability (June 2007)

LAMMEL, R.; VARANOVICH, A. (2014) Interpretation of Linguistic Architecture. In European Conference on Modeling Foundations and Applications (ECMFA 2014), pages 67-82. Springer.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI S., SCARCELLO, F. The DLV System for Knowledge Representation and Reasoning, 2004.

LIMON, A., GARBAJOSA, J.: The need for a unifying traceability scheme. In: Proceedings of the ECMDAWorkshop on Traceability (2005)

MAOZ, S. (2009). Using Model-Based Traces as Runtime Models. In Computer 42(10):28–36.

M. OHMAN, S. JOHANSSON, and K.E. ARZEN. “Implementation Aspects of the PLC standard IEC 1131-3”. In: Journal on Control Engineering Practice. Vol. 6. 4. Elsevier, 1998, pp. 547–555.

MURPHY, K. (2012). Machine Learning: A Probabilistic Perspective. MIT Press.

OLDEVIK, J., NEPLE, T., GRONMO, R., AAGEDAL, J., BERRE, A.: Toward standardised model-to-text transformations. In: ECMDA. LNCS, vol. 3748. Springer, Berlin (2005)

OMG’s UML (2017). "The Unified Modeling Language, Object Management Group". Accessed July 31, 2017. http://www.uml.org/

OMG’s MOF models to text transformation language; final adopted specification. OMG Document 08-01-16, http://www.omg.org/spec/MOFM2T/1.0/PDF (2006)

OMG’s MOF QVT draft specification. OMG Document ptc/2007/07/07, http://www.omg.org/cgibin/doc?ptc/ 2007-07-07 (2007)

OMG’s MARTE (2017). "UML Profile For MARTE™: Modeling And Analysis Of Real-Time Embedded Systems™". Accessed September 26, 2017. http://www.omg.org/spec/MARTE/

Open Group’s TOGAF (2017). "The Open Group Architecture Framework". Accessed July 31, 2017. https://www.opengroup.org/togaf/

OSLC: Open Services for Lifecycle Collaboration, http://www.open-services.net , Accessed: July 2017.

OSLC Primer - Online book, http://open-services.net/uploads/resources/OSLC_Primer_-_Learning_the_concepts_of_OSLC.pdf

PAGAN, J.; CUADRADO, J.; MOLINA, J. (2015). A Repository for Scalable Model Management. In Journal of Software and Systems Modeling, 14(1):219-239. Springer.

PAIGE, R.F, DRIVALOS, N., KOLOVOS, D.S., FERNANDES K.J., POWER, C., OLSEN, G.K., ZSCHALER, S. Rigorous identification and encoding of trace-links in model-driven engineering. Software and System Modeling 10(4): 469-487 (2011)

PARSONS, T.; MURPHY, J. (2008). Detecting Performance Antipatterns in Component Based Enterprise Systems. In Journal of Object Technology 7:55–91.

ROMERO J.R.; JAEN, J.I.; VALLECILLO A. (2009). Realizing Correspondences in Multi-Viewpoint Specifications. In EDOC 2009.

SAADATMAND, MEHRDAD; BUCAIONI, ALESSIO. (2014). OSLC Tool Integration and Systems Engineering - The Relationship Between The Two Worlds. In the 40th Euromicro Conference on Software Engineering and Advanced Applications.

SALAY, R. MYLOPOULOS, J. EASTERBROOK, S.: Managing models through macromodeling. In: ASE '08: Proceedings of the 2008 23rd IEEE/ACMInternational Conference on Automated Software Engineering, pp. 447–450. IEEE Computer Society, Washington, DC (2008)

SALAY, R.; MYLOPOULOS, J.; EASTERBROOK, S. (2009) Using Macromodels to Manage Collections of Related Models. In Advanced Information System Engineering, pages 141-155/ Springer.

SCHEIDGEN, M.; ZUBOW, A.; FISCHER, J.; KOLBE, T. (2012). Automated and Transparent Model Fragmentation for Persisting Large Models. In the 15th MoDELS Conference, pages 102-118. Springer.

SEIBEL A., NEUMANN S., GIESE H., Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. Softw. Syst. Model 9(4), 493-528 (2010)

SHAH, S; WEI, R; KOLOVOS, D; LOUIS, M; PAIGE, R. (2014). A Framework to Benchmark NoSQL data stores for large-scale model persistence. In the 17th MoDELS Conference, pages 586-601. Springer

SOUSA, A., KULESZA, U., RUMMLER, A., ANQUETIL, N., MITSCHKE, R., MOREIRA, A., AMARAL, V., ARAÚJO, J.: A model-driven traceability framework to software product line development. In: Proceedings of the ECMDA Workshop on Traceability (2008)

SMITH, C.U. (2007). Introduction to Software Performance Engineering: Origins and Outstanding Problems. In Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, ser. Springer Berlin Heidelberg, 2007, pages. 395–428.

SMITH, C.U.; WILLIAMS, L.G. (2001). Software Performance AntiPatterns; Common Performance Problems and their Solutions. In 27th International Computer Measurement Group Conference (CMG), Computer Measurement Group, 2001, pages. 797–806.

SMITH, C.U.; WILLIAMS, L.G. (2002). New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In 28th International Computer Measurement Group Conference (CMG), Computer Measurement Group, 2002, pages. 667–674.

SMITH, C.U.; WILLIAMS, L.G. (2003). More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In 29th International Computer Measurement Group Conference (CMG), Computer Measurement Group, 2003, pages. 717–725.

TRUBIANI, C.; KOZIOLEK, A. (2011). Detection and solution of software performance antipatterns in palladio architectural models. In Second Joint WOSP/SIPEW International Conference on Performance Engineering (ICPE). ACM, 2011, pages 19–30.

VIGNAGA A.; JOUAULT, F.; BASTARRICA, M.C; BRUNELIERE, H. (2013) Typing Artifacts in Megamodeling. Software and Systems Modeling 12(1):1-15. Springer.

VOJTISEK, D. (2009): Traceability MDK for KerMeta. http://www.kermeta.org/mdk/traceability/. Last accessed: June 2009

WERT, A.; OEHLER, M.; HEGER, C.; FARAHBOD, R. (2014). Automatic Detection of Performance Anti-patterns in Inter-component Communications. In Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA). ACM, 2014, pages 3–12.

WHITTLE, J.; HUTCHINSON, J.; ROUNCEFIELD, M. (2014). The State of Practice in Model-Driven Engineering. In IEEE Software, 31(3):79-85.

WIEDERHOLD, G. (1986). Views, Objects and Databases. IEEE Computer, 19(12):37–44.

WIEDERHOLD, G.; WEGNER, P.; CERI, S. (1992). Toward Megaprogramming. Communications of the ACM, 35(11): 89-99.

WOODSIDE, M.C.; FRANKS, G.; PETRIU, D.C. (2007). The Future of Software Performance Engineering. In International Conference on Software Engineering (ICSE), Workshop on the Future of Software Engineering (FOSE). IEEE Computer Society, 2007, pages 171–187.

XU, J. (2008). Rule-based Automatic Software Performance Diagnosis and Improvement. In Proceedings of the 7th International Workshop on Software and Performance (WOSP). ACM, 2008, pages 1–12.