Exploring model repositories by means of megamodel-aware search operators

Francesco Basciani
University of L'Aquila
L'Aquila, Italy
francesco.basciani@univaq.it

Davide Di Ruscio
University of L'Aquila
L'Aquila, Italy
davide.diruscio@univaq.it

Juri Di Rocco
University of L'Aquila
L'Aquila, Italy
ljuri.dirocco@univaq.it

Ludovico Iovino
Gran Sasso Science Institute
L'Aquila, Italy
ludovico.iovino@gssi.it

Alfonso Pierantonio
University of L'Aquila
L'Aquila, Italy
alfonso.pierantonio@univaq.it

ABSTRACT

Great strides have been made in the development of tools and techniques for advance model management over the last decade. Despite the use of model repositories is gaining traction in industry, their use is still hampered by the limited understanding of the underlying platform semantics. Consequently, the all-important goal of reusing artefacts has led to an enduring quest for ways to search and retrieve artifacts more efficiently and accurately. Arguably, a contributory factor limiting the use of current search engines is the poor alignment between the query languages and the lattice of relations among the different and heterogeneous artifacts in the repository.

In this paper, a novel approach to model search is presented. By leveraging the repository structure into *megamodels*, well-formed search operators have been conceived in order to permit designers to reliably explore and browse model repositories. An experimental investigation has been conducted by implementing the approach in the MDEForge platform by employing the Lucene search library.

1 INTRODUCTION

The pervasiveness of modelling techniques in everyday software practice has escalated the importance of reliable model repositories [11]. Consequently, the availability of efficient and accurate ways to retrieve artifacts is becoming of high relevance. Thus, relying on sound and well-formed models for discovering and reusing existing artifacts is key to preserving productivity benefits related to model-based processes [18]. While such advantages are particularly attractive, substantial shortcomings have been identified as problems that are inherent to the way repositories are modelled themselves. Arguably, a contributory factor limiting the use of current search engines is the poor alignment between the query languages and the lattice of relations among the different and heterogeneous artifacts in the repository. In particular, the diversity and numerosity of models stored in a repository require query mechanisms based on a finer-grained level of understanding of the repository. For instance, in order to locate an artifact, it might be useful to be able to predicate over both repository-wide attributes, including artifact types, metamodels, domain types, and maturity levels, and metamodel elements, such as classes and structural features.

This article outlines a novel approach to model search that leverages the repository structure into a megamodel. The approach provides designers with dedicated operators to explore the model repository without requiring the knowledge of low-level details about the underlying platforms to formulate the queries. The approach has been implemented atop of MDEForge [4] by employing Lucene [16] to feature efficient text search.

Structure of the paper. The paper is structured as follows. Next section presents a motivating scenarios. Section 3 makes an overview of existing model search approaches. Next section introduces the approach, which is demonstrated in Sect. 5. Finally, Sect. 6 concludes the paper and discusses future work.

2 MOTIVATING SCENARIOS

In this section, we discuss explanatory scenarios that involve models, metamodels, and model transformations. The goals are:

- highlighting the need for proper methods and tools supporting the exploration of model repositories managing different kinds of interrelated modelling artifacts;
- showing that even the implementation of simple search scenarios can be error-prone and time-consuming if not adequately supported.

For each scenario, a corresponding query implemented in OCL (and executed in Java) is presented. For the sake of clarity, the queries assume the availability of modelling artifacts stored in a local user folder.

Scenario 1. In this scenario, the modeler is interested in metamodels that contain a specific metaclass defined in terms of its name and structural features. For instance, metamodels for graphs can be found by searching the terms *nodes* and *edges*. Listing 1 shows a query whose definition is given in OCL at line 4. Since it has to be evaluated on all the artifacts locally stored, at lines 5-12 all the metamodels in a specified

The research described in this paper has been partially supported by the CROSSMINER Project, EU Horizon 2020 Research and Innovation Programme, grant agreement No. 732223.

folder are retrieved. Then, for each corresponding package the *checkConstraint* method is executed. Line 6 checks the artifact is a metamodel by looking at its extension (*ecore*). The method *checkConstraint* executes the OCL predicate by considering the parameters given as input, namely *className, attrName* and *refName*. Each OCL query result is added to the *result* list, which will contain the list of metamodels satisfying the query.

Listing 1: Sample query supporting *Scenario 1*

```
1 public  List<File> search(String folderString, String className
      , String attrName, String refName) {
2  File folder = new File(folderString);
3  List<File> results = new ArrayList<Artifact>();
4  String query = "EClass.allInstances()->exists(e␣|␣e.name='"+
       className+"')␣and␣EAttribute.allInstances()->exists(e␣|␣
       e.name='"+attrName+"')␣and␣EReference.allInstances()->
       exists(e␣|␣e.name='"+refName+"')"
5  for (final File fileName : folder.listFiles()) {
6   if (getFileExtension(fileName).equals("ecore")) {
7    List<EPackage> epList = getEPackages(filename)
8    for (EPackages package : epList)
9     if(checkConstraint(package, query))
10     results.add(fileName);
11  }
12 }
13 return result,
14}
```

It is worth noting that even such simple search requires writing a query (with three nested levels) that is a tedious and error-prone activity, despite its simplicity (e.g., no relation among artifacts is involved).

Scenario 2. In this scenario, the designer is interested in finding transformations able to generate specific elements out of source models of a given type. For instance, the developer is working on some model transformations able to generate Petri net models out of BMPN specifications. To this end, she would like to get inspired by existing transformations (if any) in order to understand how to develop the mappings between BPMN *tasks* and corresponding Petri net modules. Listing 2 contains an OCL query that at line 4 looks for transformations mapping the concepts expressed in the parameter *inPatternName* into one instance of *outPatternName*. The query is evaluated for each *atl* file, see *checkContraint* at line 8. The outcome is stored in the list *result*, see line 9.

Listing 2: Sample query supporting *Scenario 2*

```
1 List<File> search(String folderString, String outPatternName,
      String inPatternName) {
2  File folder = new File(folderString);
3  List<File> result = new ArrayList<File>();
4  String query = "SimpleOutPatternElement.allInstances()->exists
       (e␣|␣e.type.name␣=␣'" + outPatternName+"')␣and␣
       SimpleInPatternElement.allInstances()->exists(e␣|␣e.type
       .name␣=␣'" + inPatternName+"')"
5  for (final File fileName : folder.listFiles())
6   if(getFileExtension(fileName).equals("atl")){
7    ATLModel atlModel = injectTrasformation(fileName.getName());
8    if (checkConstraint(atlModel.getRoot(), query))
9     result.add(fileName);
10  }
11}
```

Scenario 3. In this scenario, the designer is interested in models conforming to a specific metamodel. For instance, the modeler would like to reuse and refine the architectural specification (given in a some specific ADL) of an already implemented software system. Then, Listing 3 contains a sample query implemented in Java that makes use of EMF[1] methods. In particular, the query returns all the models conforming to a metamodel *MM* denoted by its *nsUri* (see lines 5-11). In particular, the *getNsURiFromModels* method (line 6) extracts package information by using the EMF *.eClass()* and *.eResource()* methods.

Listing 3: Sample query supporting *Scenario 3*

```
1 public  List<File> search(String folderString, String nsUri) {
2  File folder = new File(folderString);
3  List<File> results = new ArrayList<Artifact>();
4
5  for (final File fileName : folder.listFiles()) {
6   List<EPackage> epList = getNsURiFromModels(filename);
7   if(containsUri(epList, nsUri))
8    results.add(fileName);
9  }
10 return result,
11}
12f.eClass().eResource().getURI()
```

Again, despite the simplicity of the requirements the designer must face a certain accidental complexity due to the technological setting that requires the familiarity with the EMF framework and its corresponding APIs.

3 BACKGROUND

A number of existing approaches for model searching are summarized in Table 1. For each of them, the following characteristics are considered:

- *Supported modeling artifact:* it refers to the kinds of artifacts that the considered approach is able to manage;
- *Query mechanism:* it refers to how query are specified; e.g., there are approaches that allow users to specify queries with query strings; others adopt more structured languages like OCL;
- *Megamodel-awareness:* some approaches consider also relations among different kinds of artifacts, where relations give place to *joins* that traverse the repository; for instance, like searching for all metamodels supported by existing editors that are source metamodels of a transformation that generates models conforming to a given metammodel; considering the artifact relations requires the approach to be aware of the repository structure typically represented by means of megamodels [9];
- *Indexing supported:* in order to make searches more efficient, approaches may rely on indexing mechanisms.

The first entry in the table is an approach [10] for retrieving UML[2] design models using the combination of WordNet and Case-Based Reasoning. The approach makes use also of a similarity distance to 'approximate' the result. Moogle [14, 15] is a model search engine that relies on the use of metamodeling information for creating indexes allowing the execution of complex queries. It also delivers the results in a readable way by removing irrelevant strings from the actual model

[1] https://www.eclipse.org/modeling/emf/
[2] https://www.omg.org/spec/UML/

	Supported modeling artifact	Query mechanism	Megamodel-awareness	Indexing supported
Gomes P. et al. [10]	UML models	Query-by-example	No	Yes
Lucrédio D. et al. [14, 15]	Any	Text search	No	Yes
Konstantinos B. et al. [2, 3]	Any	OCL-like language	Partial	Yes
Kessentini et al. [12]	Metamodel	Query-by-example	No	No
Bislimovska B. et al. [6]	WebML Models	Query-by-example	No	Yes
Bozzon A. et al. [7]	Any	Text search	No	Yes
Kling W. et al. [13]	Any	OCL-like language	Yes	No
Ángel. et al. [1]	Model and Metamodel	Text search	No	Yes

Table 1: A sample of approaches for model search. The list is inevitably non–exhaustive and merely intended to reflect the kinds of approaches that are available

file. Moogle can search for models conforming to different languages, as long as there is a well-defined metamodel which can be provided to Moogle. The approach is limited to a single architectural layer without considering for example relationships among artifacts. For the indexing mechanism Moogle relies on the existing Apache SOLR[3] search engine.

Hawk is a framework aiming at providing scalable techniques for large-scale model querying and transformations. In [2, 3] the authors compare the conventional and commonly used persistence mechanisms in MDE with novel approaches such as the use of graph-based NoSQL databases. Moreover, they present an extensible model indexing framework at the base of the developed tool. The proposed framework collects models stored in file-based version control systems and persists them in indexes, while not altering the original files. The *node* concept, represents metamodel types and contain their name, they are linked with relationships to their model. This mechanism directly supports conformance navigation but we are not sure if can be adapted to fully support a megamodel-based relationship navigation. The default query mechanism is based on native APIs e.g., related to Neo4J[4] and OrientDB[5].

In [12], the authors propose a search-based metamodel matching mechanism combining structural and syntactic metrics to generate correspondences between metamodels. The approach considers metamodel matching as an optimization problem. They adopted a global search to generate an initial solution and, subsequently, a local search, namely simulated annealing, to refine the initial solution generated by the genetic algorithm. The approach starts by generating a set of possible matching solutions between the source and target metamodels randomly. Then, these solutions are evaluated using a fitness function based on structural and syntactic measures.

In [6], the authors investigate the adoption of different techniques for indexing and searching model repositories, by focusing on WebML [8] models. Keyword-based and content-based techniques are employed to provide users with a query-by-example paradigm. In [7], authors proposes the adoption of information retrieval techniques. They identify relevant design dimensions and several options (project segmentation, index structure, query language and processing, and result presentation) and presents an architecture for the automatic model-driven project segmentation, indexing and search, without requiring any manual model annotation. The proposed approach is agnostic on the type of modelling artifact. In [13], the authors propose MoScript, an OCL-based scripting language that permits to query a model repository by relying on the metadata available in a dedicated megamodel.

The authors in [1] propose EXTREMO, a tool developed as an Eclipse plugin, able to gather heterogeneous information from different technological spaces (like ontologies, RDF, XML or EMF). EXTREMO represents them uniformly in a common data model that enables an uniform querying, by means of an extensible mechanism, which can make use of services, e.g., for synonym search and word sense analysis.

By a brief analysis of Table 1, it clearly emerges that only few approaches leverage the relationships among the artifacts as first-class entities to be used when exploring an existing model repository. Most of the analyzed approaches are based on indexing techniques for the sake of better performance. Some approaches have recognized the importance of being generic and supporting the management of any kind of modeling artifacts. Among the considered approaches, the preferred query mechanism follows the 'query-by-example' model.

In the next section, the proposed approach is presented. It aims at being generic in order to manage any kind of modeling artifact. Moreover, it reduces the accidental complexity by allowing modelers to efficiently search repositories by means of dedicated search operators that prevent the designer from becoming familiar with specific languages, systems, and technologies.

4 PROPOSED APPROACH

Conventional wisdom on managing large repositories suggests that technical merit of the query model is key to success. In fact, as shown in Sect. 2 very simple requirements may lead to complex queries embedded in components that involve several languages and tools. We aim to address this shortcoming by proposing a technique that is based on search operators that abstract from the underlying machinery. For instance, Gmail provides users with a list of domain-specific operators[6] that can be used when searching throughout mails, e.g., the operator `has:attachment` returns all messages with an attachment. Multiple operators can be combined in complex search string, as for instance `"to:david has:youtube"` that

[3] http://lucene.apache.org/solr/
[4] https://neo4j.com
[5] https://orientdb.com/
[6] https://support.google.com/mail/answer/7190?hl=en

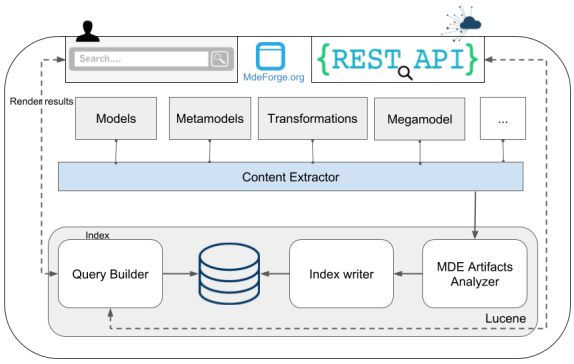

Figure 1: Architecture overview of the proposed model search infrastructure

filters messages sent to a specific recipient with a YouTube video. The interesting idea about this is that it is based on a very simple syntax that does scale well, in the sense that *i)* it is platform agnostic as it does not require specific expertise to be used; and *ii)* the notation remains concise despite searches can get complex. Thus, the same mechanism has been adopted for searching through model repositories. For instance, let us assume we are interested in retrieving all transformations that consumes models conforming to a *Family* metamodel, the expression `fromMM:Family` returns them. Now, if we restrict our attention to only those transformations that return `Person` models, we can write fromMM:Family toMM:Person. An excerpt of the available operators is in Table 2 together with descriptions and typing requirements.

The approach has been implemented by integrating the *Lucene* [16] search engine in the MDEForge [5] platform. A detailed overview of the technologies, model search infrastructure, and operators is given in the next section.

4.1 Overview of the technology baseline

MDEForge is an extensible modeling framework that consists of services for storing, managing, analysing any kind of modeling artefacts. Extensions can be developed and integrated in the platform by starting from the services exposed by the platform. Restful APIs allow implementors to design complex modelling life-cycles that are in turn used as software-as-a-service. The persistency layer provided by MDEForge is at the base of this work and will be used to retrieve the artifacts that can be searched by means of specific search operators. Interested readers can refer to [5] for more details about the technical details about MDEForge and its megamodel-based architecture.

Lucene is a simple but powerful Java-based and open source search library. It is scalable and its high-performance enables its adoption to index and search virtually any kind of text. Lucene can be used in any application to add search capabilities to it. The library provides the core operations, which are typically required by any search application. The main operations the engine provides can be summarized as: collecting the content, analyzing the artifacts, indexing the documents, providing a search interface, query building, query execution, and showing results.

4.2 Model search infrastructure

Figure 1 illustrates the architecture of the search infrastructure with the components underpinning the integration of MDEForge and Lucene. In particular, Lucene is used to create indexes related to the artifacts stored in the MDEForge repository. The indexing operation can be customized for example, scheduling it every time a new artifact is uploaded. The content to be considered when creating the indexes is retrieved by the *Content extractor* component, which extracts specific information for each type of artifact. For instance, for metamodels the component extracts relevant metamodel characteristics, including package names, nsUri, metaclass names, attribute and reference names, enumerations, literales, and datatypes. Concerning model transformations, the content extractor retrieves for each transformation specific attributes like helper and rule names, etc. Interestingly, the content extractor implements some reflection as well, since it retrieves also information about the megamodel representing all the existing relationships of the ecosystem stored in the repository. For instance, the megamodel explicitly represents the *conformsTo* relation between a model and the corresponding metamodel. Such conformance relation is used to index models with respect to the corresponding types. For instance, if we consider the *Person* metamodel[7], the content extractor will retrieve information so to enable the possibility to query models conforming to the *Person* metamodel by searching for *persons* with a particular name and so on. There are attributes that are common for all the types of artifacts and that are retrieved by the extractor as, e.g., author's name, update time, etc.

Once all the artifacts stored in the repository have been extracted, they are analyzed by the *MDE Artifacts Analyzer*, which enriches the indexes created by the *Index writer* component. When users submit a search string (by means of the Web-based interface or via the available Rest API), a corresponding query is built by the *Query Builder* component in order to retrieve artifacts from the MDEForge repository with respect to the available indexes.

Figure 2 shows a screenshot of the Web-based search page, consisting of three main parts: the search form, the list of available search operators that can be used for specifying the query, and the query results. In the shown example, the search string makes use of three operators, namely `eClass:`, `eReference:`, and `eAttribute:` in order to search for all the metamodels containing metaclasses named *Family*, which in turn contains a reference named *members*, and an attribute named *age*. The execution of that query produced one result consisting of the *Family.ecore* metamodel.

Query results are ranked with respect to a matching score between the query and the found artifacts (in the example shown in Fig. 2 it is 755). The score is determined by the Lucene engine and depends on many factors. In particular,

[7] http://www.eclipse.org/atl/atlTransformations/

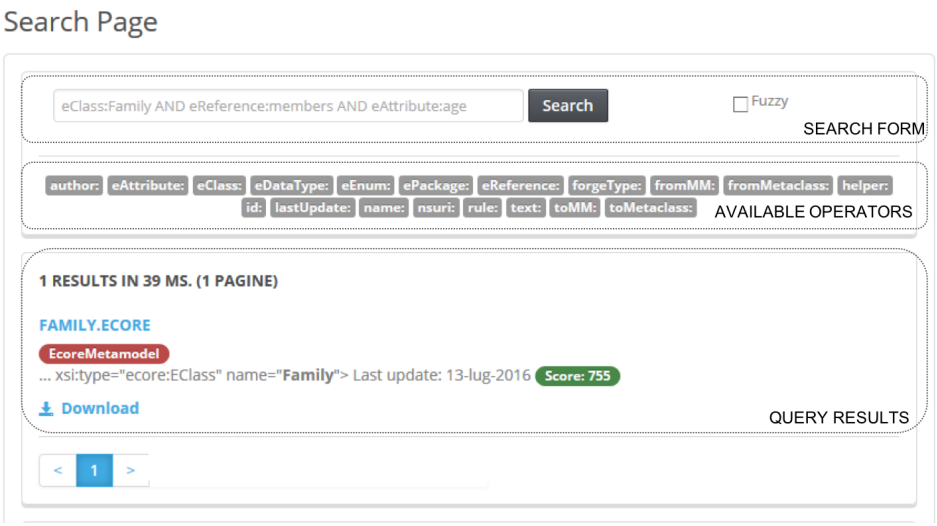

Figure 2: The proposed model search infrastructure at work

Operator name	Artifact	Description
`name:`	Any	It returns all the artifacts matching the name provided by the query tag value
`author:`	Any	It returns all the artifacts provided by a specific author
`conformToMM:`	Metamodel	It returns the models that conform to the metamodel named as the provided value
`eClass:`	Metamodel	It returns the metamodels containing at least one metaclass named as the provided value
`eAttribute:`	Metamodel	It returns the metamodels containing at least one metaclass having an attribute named as the provided value
`eReference:`	Metamodel	It returns the metamodels containing at least one metaclass having a reference named as the provided value
`fromMM:` / `toMM:`	Transformation	It returns all the transformations having as source metamodel the provided value for the fromMM tag and as destination the provided value for the toMM tag
`fromMC:` / `toMC:`	Transformation	It returns all the transformations having a rule transforming the metaclass specified in the value for the first tag into the metaclass specified as value for the last tag

Table 2: Excerpt of the available search operators

to determine that value, Lucene implements a variant of the TF-IDF[17] scoring model.

4.3 Model search operators

As mentioned above, Table 2 shows an excerpt of the operators provided by the proposed approach and that can be used to search throughout a repository as shown in Fig. 2. Typing information are in the second column, i.e., the operator `name:` is evaluated on any type of artifacts, whilst `eClass:` is executed only on metamodels.

The operators can be used in the form `key:`*`value`*, where `key` is the operator specification and *`value`* is the matching term. More complex search expressions can be obtained by a conjunction of operators

$$\{\texttt{key:}\mathit{value}\}^{+}$$

Furthermore, Lucene provides users with additional operators that can be used to compose queries [16].

5 EXPERIMENTS

In this section, a discussion is provided about the application of the proposed approach to a dataset consisting of 2.422 metamodels, 350 models, and 115 transformations[8]. In particular, the approach has been used to specify and execute the queries discussed in Sect.2. The main goal of the experiments is to perform a preliminary experimental assessment of the proposed approach with respect to *i)* its suitability to support the specification of model queries involving different kinds of interrelated artifacts; *ii)* its performance in terms of query execution time. The queries that have been employed in the experiments are the following:

Q1: Get all the metamodels having a metaclass named *className*, which in turn contains an attribute named *attrName*, and a reference named *refName*;

Q2: Get all the model transformations transforming metaclasses named *metaclassName1* to generate target *metaclassName2* elements;

Q3: Get all the models conforming to the metamodel named *metamodelName*.

The queries are written with the proposed approach and in OCL in order to provide a comparison. Table 3 shows

[8]These are publicly available artifacts; as it often happens it is not easy to retrieve models from publicly accessible repositories, while metamodels are typically easier to find.

	Proposed approach		OCL based approach	
	exec.time (ms)	**query string**	**exec.time (ms)**	**#loc**
Q1	39	`eClass:`*`className`* `AND eAttribute:`*`attrName`* `AND eReference:`*`refName`*	12641	≈70
Q2	59	`fromMetaclass:`*`metaclassName1`* `AND toMetaclass:`*`metaclassName2`*	7701	≈40
Q3	24	`conformToMM:`*`metamodelName`*	8102	≈60

Table 3: Performed experiments

relevant data related to the application of both versions of the queries. The specification of the queries by means of the proposed operators is given in the third column of the same table, whereas the number of lines of code (#loc) of the OCL-based solutions is shown in the last column. The #loc values are obtained by considering the length of the code shown in Sect. 2 and the lines of the auxiliary methods that have been implemented for supporting the execution of those queries. For both approaches, the query execution times are reported in milliseconds (see the second and fourth columns of the table).

From the experimental data it emerges that the execution time of the OCL-based queries is always considerably higher than that of the proposed approach. In particular, for **Q2** the execution time of the OCL query is eight times higher; this can be explained with the cardinality of metamodels that is much higher of that of the other artifacts. As to the verbosity, the proposed approach outperforms the OCL-based query. It is worth noting that the execution times of the queries specified by means of the proposed approach do not take into account the time needed to create the indexes as discussed in the previous section. However, indexes are typically created off-line by means of batch processes, which do not interfere with the actual execution of queries. Moreover, each time the indexes must be updated because new artifacts are added, this is done incrementally.

6 CONCLUSION AND FUTURE WORK

Modern modelling tools are becoming more and more distributed platforms where artifacts can be persistently stored and coherently dealt with. As a consequence, being able to conveniently search throughout the repository according to specific criteria is key to any reuse practice. The approach presented in this paper proposes simple yet powerful operators to perform complex repository searches. The corresponding search infrastructure permits modelers to explore model repositories in an efficient way by abstracting form the specific platforms and tools used to formulate the queries. The approach has been implemented by integrating the Lucene search engine in the MDEForge platform. Preliminary experiments show that the approach is promising, especially when compared with traditional techniques, even though more accurate comparison criteria and metrics have to be properly defined. Moreover, the adoption of alternative frameworks for model query, like Hawk [2] will be also investigated.

REFERENCES

[1] Mora Segura Ángel, Juan de Lara, Patrick Neubauer, and Manuel Wimmer. 2018. Automated modelling assistance by integrating heterogeneous information sources. *Computer Languages, Systems & Structures* 53 (2018), 90–120.

[2] Konstantinos Barmpis and Dimitris Kolovos. 2013. Hawk: Towards a Scalable Model Indexing Architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering (BigMDE '13)*. ACM, New York, NY, USA, Article 6, 9 pages.

[3] Konstantinos Barmpis and Dimitrios S Kolovos. 2014. Towards scalable querying of large-scale models. In *European Conference on Modelling Foundations and Applications*. Springer, 35–50.

[4] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. 2014. MDEForge: an Extensible Web-Based Modeling Platform. *CloudMDE@MoDELS* (2014), 66–75.

[5] Francesco Basciani, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2014. Automated Chaining of Model Transformations with Incompatible Metamodels. *MoDELS* 8767, Chapter 37 (2014), 602–618.

[6] Bojana Bislimovska, Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. 2014. Textual and content-based search in repositories of web application models. *ACM Transactions on the Web (TWEB)* 8, 2 (2014), 11.

[7] Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. 2010. Searching repositories of web application models. In *International Conference on Web Engineering*. Springer, 1–15.

[8] Stefano Ceri, Piero Fraternali, and Aldo Bongio. 2000. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks* 33, 1-6 (2000), 137–157.

[9] Juri Di Rocco, Davide Di Ruscio, Johannes Härtel, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. 2018. Systematic Recovery of MDE Technology Usage. In *Theory and Practice of Model Transformation*, Arend Rensink and Jesús Sánchez Cuadrado (Eds.). Springer International Publishing, Cham, 110–126.

[10] Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, and Carlos Bento. 2004. Using WordNet for Case-based Retrieval of UML Models. *AI Commun.* 17, 1 (Jan. 2004), 13–23. http://dl.acm.org/citation.cfm?id=992846.992849

[11] Brahim Hamid. 2017. A model-driven approach for developing a model repository: Methodology and tool support. *Future Generation Computer Systems* 68 (2017), 473 – 490.

[12] Marouane Kessentini, Ali Ouni, Philip Langer, Manuel Wimmer, and Slim Bechikh. 2014. Search-based metamodel matching with structural and syntactic measures. *Journal of Systems and Software* 97 (2014), 1–14.

[13] Wolfgang Kling, Frédéric Jouault, Dennis Wagelaar, Marco Brambilla, and Jordi Cabot. 2012. MoScript: A DSL for Querying and Manipulating Model Repositories. In *Software Language Engineering*, Anthony Sloane and Uwe Aßmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 180–200.

[14] Daniel Lucrédio, Renata P. de M. Fortes, and Jon Whittle. 2008. *MOOGLE: A Model Search Engine*. Springer Berlin Heidelberg, Berlin, Heidelberg, 296–310.

[15] Daniel Lucrédio, Renata P de M Fortes, and Jon Whittle. 2012. MOOGLE: a metamodel-based model search engine. *Software & Systems Modeling* 11, 2 (2012), 183–208.

[16] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in action: covers Apache Lucene 3.0*. Manning Publications Co.

[17] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. 133–142.

[18] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. 2015. Collaborative Repositories in Model-Driven Engineering [Software Technology]. *IEEE Software* 32, 3 (May 2015), 28–34.