Software and Systems Modeling
https://doi.org/10.1007/s10270-021-00960-4

SPECIAL SECTION PAPER

An efficient and scalable search engine for models

José Antonio Hernández López[1] · Jesús Sánchez Cuadrado[1]

Received: 22 February 2021 / Revised: 1 October 2021 / Accepted: 22 November 2021

Abstract
Search engines extract data from relevant sources and make them available to users via queries. A search engine typically crawls the web to gather data, analyses and indexes it and provides some query mechanism to obtain ranked results. There exist search engines for websites, images, code, etc., but the specific properties required to build a search engine for models have not been explored much. In the previous work, we presented MAR, a search engine for models which has been designed to support a query-by-example mechanism with fast response times and improved precision over simple text search engines. The goal of MAR is to assist developers in the task of finding relevant models. In this paper, we report new developments of MAR which are aimed at making it a useful and stable resource for the community. We present the crawling and analysis architecture with which we have processed about 600,000 models. The indexing process is now incremental and a new index for keyword-based search has been added. We have also added a web user interface intended to facilitate writing queries and exploring the results. Finally, we have evaluated the indexing times, the response time and search precision using different configurations. MAR has currently indexed over 500,000 valid models of different kinds, including Ecore meta-models, BPMN diagrams, UML models and Petri nets. MAR is available at http://mar-search.org.

Keywords Model repositories · Search engines · Model-driven engineering

1 Introduction

The availability of mechanisms for effective navigation and retrieval of software models are essential for the success of the MDE paradigm [16,26], since they have the potential to foster communities of modellers, improve learning by letting newcomers explore existing models and to discover high-quality models which can be reused. There are several model repositories available [26], some of which offer public models. For instance, the GenMyModel[1] cloud modelling service features a public repository with thousands of available models, although it does not support any search mechanism. Source code repositories like GitHub and GitLab also host thousands of models of different kinds. However, these services do not provide model-specific features to facilitate the discovery of relevant models.

In this setting, search engines are the proper tool for users to be able to locate the relevant models for their tasks. For example, Moogle [42] is a text-based search engine but it does not include model crawlers and it is discontinued. In [15], a query-by-example search engine specific for WebML models is presented. However, it only scales to a few hundreds of models. Other search approaches are based on OCL-like queries [11,38] which can be efficient but only select exact matches. Another aspect is that existing approaches are not useful in practice since they do not provide access to a high number of diverse and public models.

Therefore, to date, there has been little success in creating a generic and efficient search engine specially tailored to the modelling domain and publicly available. Nowadays, when a developer faces the task of finding models, she needs to perform several steps manually. First, she needs to find out herself potential sources for models (e.g. public model repositories). Then, for each source, a dedicated query must be written (or a manual lookup must be done if no query mechanism is available), and candidate results must be collected. The results are typically not ranked according to its

[1] https://www.genmymodel.com/.

Communicated by S. Abrahão, E. Syriani, H. Sahraoui and J. de Lara.

✉ José Antonio Hernández López
joseantonio.hernandez6@um.es

Jesús Sánchez Cuadrado
jesusc@um.es

1 Facultad de Informática, Universidad de Murcia, Murcia, Spain

relevance with respect to the query. Which is more, when results come from different sources only manual comparison is possible. Altogether, searching for models is still an open problem in the MDE community.

In this paper, we present the evolution of MAR, a search engine specifically designed for models. MAR consists of several components, namely crawlers, model analysers, an indexer, a REST API and a web user interface. The crawlers and the analysers allow MAR to obtain models from known sources and prepare them to be consumed by the rest of the components. The indexer is in charge of organizing the models to perform fast and precise searches. At the user level, MAR provides two search modes: keyword based and by-example. Keyword-based search provides a convenient way to quickly locate relevant models, but it does not take into account the structure of models. In the query-by-example mode, the user creates a model fragment using a regular modelling tool or a dedicated user interface (e.g. textual editor) and the system compares such fragment against the crawled models. This allows the search process to take into account the structure of the models. To perform this task efficiently, MAR encodes models by computing paths of maximum, configurable length n between objects and attribute values of the model. These paths are stored in an inverted index [7] (a map from items to documents containing such item). Thus, given a query, the system extracts the paths of the query and access the inverted index to perform the scoring of the relevant models. We have evaluated the precision of MAR for finding relevant models by automatically deriving queries using mutation operators, and systematically evaluating the configuration parameters in order to find out a good tradeoff between precision and the number of considered paths. Moreover, we evaluate its performance for both indexing and searching obtaining results that show its practical applicability. Finally, MAR has currently indexed more than 500, 000 models from several sources, including Ecore meta-models, UML models and BPMN models. MAR is freely accessible at http://mar-search.org.

This work is based on [40], which has been extended as follows:

– An improved crawling and analysis architecture which allows us to grow the number of visited models up to 600,000.
– A new indexing process which is faster and it is now incremental. An indexer for keyword-based search has also been included.
– The UI and the REST APIs have been greatly improved, which are hopefully a better resource for the community.
– Extended evaluation, which also includes the indexing process and an analysis of the effect of the configuration parameters in the search precision and the response time.

Organization. Section 2 motivates the need for this work and presents an overview of our approach and the running example, and it introduces the user interface of MAR. Section 3 presents the crawling and analysis architecture. Sections 4 and 5 explain keyword-based and example-based search, respectively. Section 6 discusses the indexing process. Section 7 reports the results of the evaluation. Finally, Sect. 8 discusses the related work and Sect. 9 concludes.

2 Motivation and overview

In this section, we motivate the need for model-specific search engines through a running example. From this, a set of challenges that should be tackled is derived. Then, we present an overview of our approach.

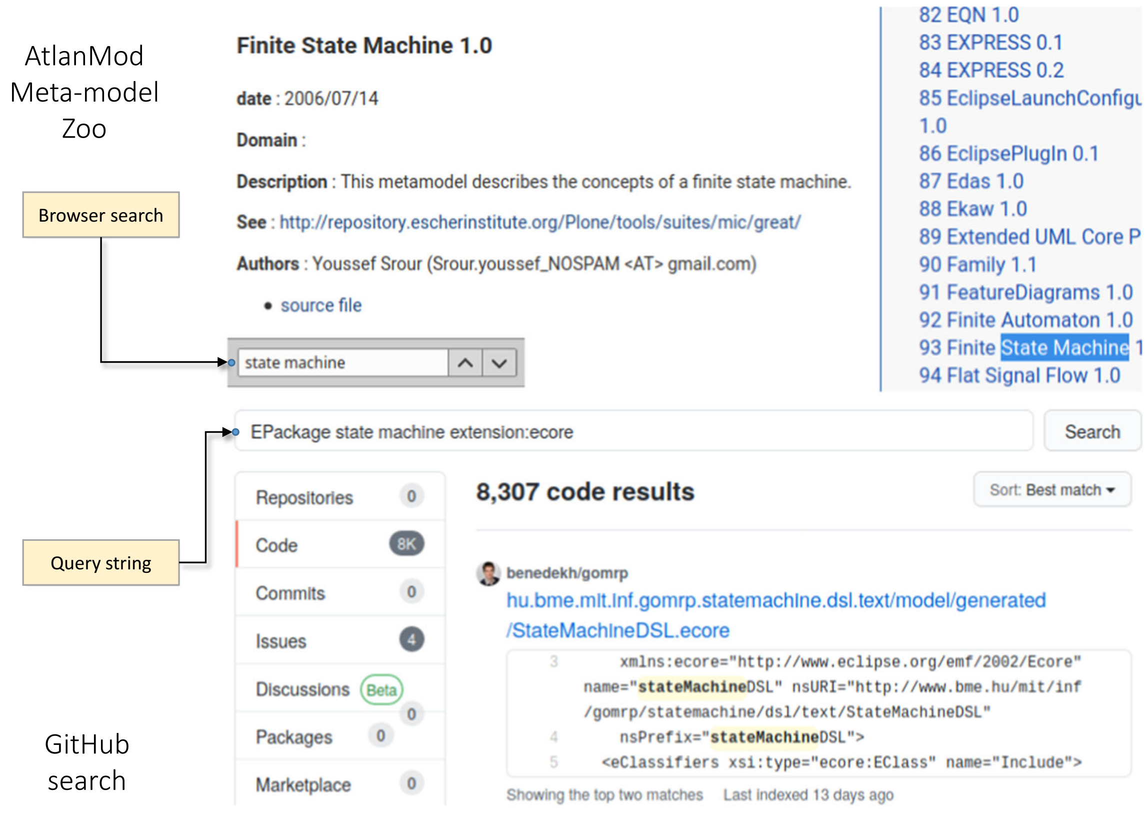
2.1 Motivation

As a running example, let us consider a scenario in which a developer is interested in creating a DSL which reflects the form of state machines, and thus, it will include concepts like *state machine*, *state*, *transition*, etc. The developer is interested in finding Ecore meta-models which can be useful either for direct reuse, for opportunistic reuse [33] (copy–paste reuse) or just for learning and inspiration purposes.

There are several places in which the developer could try to find meta-models. For instance, the ATL meta-model zoo contains about 300 meta-models,[2] but it is only possible to look up models using the text search facility of the web browser. Using this search mechanism the user could find up to five meta-models which seems to represent state machines. This is shown in the upper image of Fig. 1. However, the only way to actually decide which one is closer to what the developer had in mind is to open the meta-models in Eclipse and inspect them.

It is also possible to look up meta-models in GitHub. In this case, a possible approach to find relevant models is by constructing a specific search query. For instance, to search for Ecore meta-models about state machines one would write a query like: `EPackage state machine extension:ecore`. This query will be matched against `.ecore` files which contain the keyword `EPackage` as a hint to find Ecore/XMI formatted files, plus the `state machine` keywords. This is shown in the lower image of Fig. 1. Matched files are not validated (i.e. files may contain a bad format), duplicates are likely to exist, and the user can only see the XMI representation of the file. In addition, the number of results can be relatively large, which exacerbates the problems mentioned before, since the user needs to manually process the models.

[2] https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Ecore.



Finite State Machine 1.0

date : 2006/07/14

Domain :

Description : This metamodel describes the concepts of a finite state machine.

See : http://repository.escherinstitute.org/Plone/tools/suites/mic/great/

Authors : Youssef Srour (Srour.youssef_NOSPAM <AT> gmail.com)

- source file

state machine

82 EQN 1.0
83 EXPRESS 0.1
84 EXPRESS 0.2
85 EclipseLaunchConfigu
1.0
86 EclipsePlugIn 0.1
87 Edas 1.0
88 Ekaw 1.0
89 Extended UML Core P
90 Family 1.1
91 FeatureDiagrams 1.0
92 Finite Automaton 1.0
93 Finite State Machine 1
94 Flat Signal Flow 1.0

EPackage state machine extension:ecore

8,307 code results

benedekh/gomrp
hu.bme.mit.inf.gomrp.statemachine.dsl.text/model/generated
/StateMachineDSL.ecore

```
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
name="stateMachineDSL" nsURI="http://www.bme.hu/mit/inf
/gomrp/statemachine/dsl/text/StateMachineDSL"
    nsPrefix="stateMachineDSL">
  <eClassifiers xsi:type="ecore:EClass" name="Include">
```

Showing the top two matches   Last indexed 13 days ago

Fig. 1 Search approaches for the AtlanMod meta-model zoo (upper part) and GitHub (lower part)

Therefore, there are a number of shortcomings that developers face when trying to discover relevant models, namely:

- *Heterogenous model sources*. Models may be available in several locations, which forces the user to remember all these sources and manually check each one in turn. In contrast, a search engine provides a single entry point to discover models regardless of their locations.
- *Model-specific queries*. The unavailability of query mechanisms that target models specifically means that the user needs to adapt the queries to the underlying platform and to give special interpretation to the results.
- *Ranking results*. Results are typically not sorted using model-specific similarity criteria. In the best cases, keywords are used to drive the search. In some cases, there is no sorting at all (e.g. in the AtlanMod Zoo).
- *Result inspection*. The user would be interested in obtaining information about the model to make a decision about whether it is relevant or not. This information could be specific renderings of the model (or parts thereof), a textual description or simple tags describing the topics of the model.
- *Model validation*. When searching in source code repositories, there is no guarantee that the obtained models are valid, which means that the user needs to perform an additional step of validating the models before using them.

In this setting, the lack of modelling platform able to address these issues has been the main driving force in the creation of MAR. Its design, which is described next, is intended to address these shortcomings.

2.2 Overview

The design of a search engine for models must consider several dimensions [43]. One dimension is whether the search is *meta-model based*, in the sense that only models conforming to the meta-model of interest are retrieved. This is useful when the user wants to focus on a specific type of models and avoid irrelevant results. However, sometimes the user might be interested in exploring all types of models as a first approximation to the search. Another dimension is whether the search engine is *generic* or specific for a modelling language. There exist search engines for specific modelling languages (e.g. UML [31] and WebML [15]), but the diversity of modelling approaches and the emergence of DSLs suggest that search engines should be generic. Approaches which rely on exact matching are not adequate in this scenario since the user is typically interested in obtaining results which are approximately similar to the query. Thus, the nature of the search process requires an algorithm to perform *inexact matching*. Moreover, a *ranking mechanism* to sort the results according to their relevance is needed. The ranking needs to take into account the similarity of the query with respect to each result. A search engine requires an *indexing* mechanism for processing and storing the available models in a manner which is adequate for efficient look up. In addition, a good search engine should be able to handle *large repositories* of models while maintaining a good performance as well as search precision. Another aspect is how to present the results to the user. This requires considering the integration with existing tools and building dedicated web services.

Our design addresses the shortcomings discussed in the previous section and dimensions described above by including the following features, which are depicted in Fig. 2.

- **Model-specific queries**. The queries created by the user target the contents of the models indexed by the search engine. There are two query mechanisms available.
  - **Keyword-based queries**. In this search mode, the user interacts with the system by typing a few keywords and obtaining models whose contents match them (see Sect. 4). This is useful for quick searches but the results are less precise.
  - **Query-by-example**. In this search mode, the user interacts with the system by creating an example model (see Sect. 5). This example model contains elements that describe the expected results. The system receives the model to drive the search and to present a ranked list of results.
- **Generic search**. The engine is generic in the sense that it can index and search models when their meta-model is known. In the case of the query-by-example mode, the search is meta-model based because it takes into account the structure given by the meta-model.
- **Crawling and indexing**. An inverted index is populated with models crawled from existing software repositories (such as GitHub and GenMyModel) and datasets (such as [20]). We use Lucene as our index for keyword-based search (Sect. 4) and the HBase database as our backend for example-based queries (Sect. 6).
- **Ranking procedure**. The results are obtained by accessing the index to collect the relevant models among those available. Each model has a similarity score which is used to sort the results. The IDE or the web frontend is responsible for presenting the results in a user-friendly manner.
- **Frontend**. The functionality is exposed through a REST API which can be used directly or through some tool exposing its capabilities in a user-friendly manner. In particular, we have implemented a dedicated web interface (Sect. 2.3).
- **Metadata and rendering**. The system gathers metadata about the models and analyses them (e.g. how many elements are there in this model?) in order to provide the user with additional information to make decisions. Moreover, model-specific renderings are also provided to facilitate the inspection of the models.

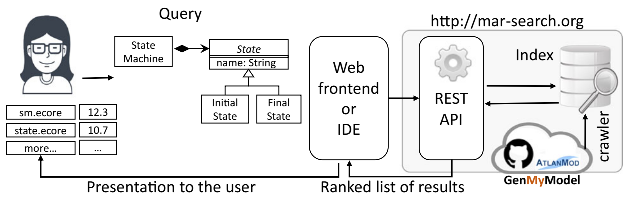


Fig. 2 Usage and components of the search engine

The availability of a platform of this type would allow modellers to discover relevant models for their tasks. We foresee scenarios like the following:

- *Model reuse*. Users who want to find models useful for their projects may use the query-by-example facility to provide a description of the structure that relevant models need to have in order to obtain them.
- *Novel modellers* may be interested in navigating models of certain type to learn from other, possibly more experience, modellers.
- *Researchers* who want to easily gather models (with metadata attached) in order to perform experiments (e.g. machine learning, analytics, etc.).
- *Large enterprises* hosting several model repositories could leverage on a search engine to make their stored models easily browsable (i.e. having a local installation of MAR).

2.3 Using MAR in practice

This section provides a summary of the usage of MAR, intended to give the reader a concrete view of how the features outlined in the previous section are materialized in practice. The goal is to facilitate the understanding of the subsequent sections.

Following with the running example about searching for models related to state machines, we consider two different, but complementary use cases.

- **Keyword-based search**. In this use case, the user wants to skim available models for state machines by quickly typing related keywords. For instance, in Fig. 3 (upper part) the user writes `state machine` as keywords. This

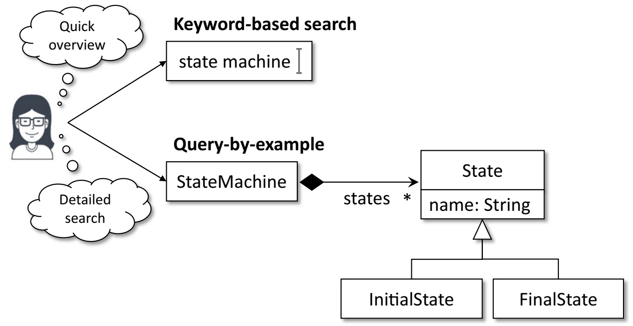


Fig. 3 Keyboard-based search and query-by-example

type of query is more likely to return irrelevant results (e.g. the keyword *state machine* alone may also return models related to factories which use machines). However, they are easy to craft and to refine. An additional advantage is that a keyword-based query can naturally be used to perform queries across model types because it is possible to build an efficient common index for all models. This can be useful to explore for which modelling formalisms there are interesting models available. For instance, a developer might be interested in models representing ATMs and she might discover useful UML state machines and activity diagrams, BPMN models and even Petri nets that model this domain. She may want to reuse, e.g. UML models but use BPMN models and Petri nets as inspiration to complete and refine the UML models.

– **Query-by-example**. In this use case, the user has already in mind the model elements and wants the results to be relatively close to a proposed example. Thus, in our system the user would create a query by crafting an example model which represents the main characteristics of the results that the user expects. Fig. 3 (bottom part) shows a model fragment in which the user indicates that they are interested in state machines in which *Initial* and *Final* states are modelled as classes. It is worth noting that this imposes a requirement on the obtained results. This means that, for instance, models in which the initial and final states are modelled using an enumeration will have a low score.

To illustrate the usage of MAR, Fig. 4 shows the web-based user interface as it is used to perform a query-by-example search.[3] In this case, the user selects that it is interested in crafting an example ❶ and chooses the type of the model, Ecore in this case ❷. Then, a model fragment or example that servers as query must be provided. This can be accomplished by uploading an XMI file or by using a concrete syntax. For Ecore, we support Emfatic syntax ❸.

When the user clicks on the *Submit!* button, the search service is invoked, and it returns the list of results. Each result ❺ contains the file name (which is a link to the original file), a description (if available), a set of topics (if available), statistics (e.g. the number of model elements) and quality indicators (e.g. quality smells [41]).

Many times the number of results can be large. MAR provides a facet-based search facility ❹ with which the user can filter out models by several characteristics, namely: the number of quality smells, elements, topics or change the sort order.

Finally, to facilitate the inspection of models, a rendering is also available ❻.

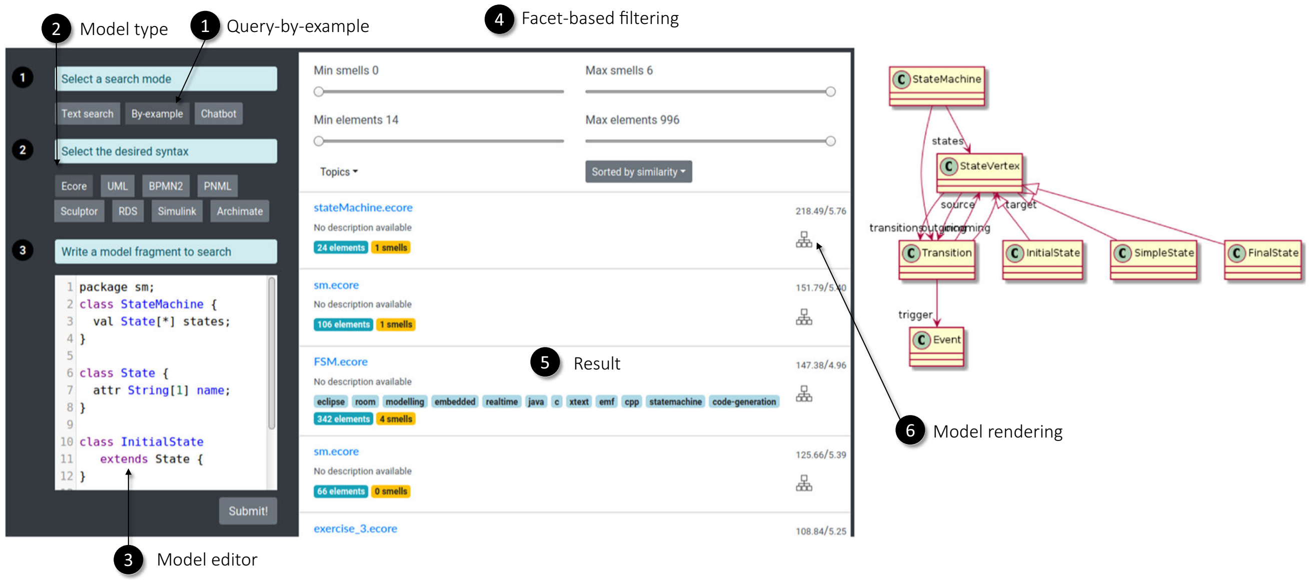
In addition to the user interface, a REST endpoint is also available. This can be useful to empower modelling tools. We foresee scenarios in which the search results are used to provide recommendations in modelling editors (similar to the approach proposed in [53]). For instance, when the user is editing a model, MAR could be called in the background and the results can be used to recommend new elements to be added to the model, model fragments that could be copy-pasted, etc. We also envision usages of the REST API to take advantage of the query facilities in order to implement approaches to recover the architecture of MDE projects [24]. For instance, given an artefact which is not explicitly typed (i.e. its meta-models are not known for some reason), its actual meta-model could be discovered by performing queries with inferred versions of its meta-model.

3 Crawling and analysis

This section describes how MAR extracts models from available resources, and how they are analysed. From a practical point of view, this is a key component of a search engine, since it is in charge of obtaining and feeding the data that will be indexed and searched. Moreover, it poses a number of technical challenges which are also discussed in this section.

Figure 5 shows the architecture that we have built. It has three main parts: crawlers which collect models, analysers which take these models and validate, compute statistics and extract additional metadata from them, and indexers which are in charge of organizing the models so that they can be searched in a fast manner. The goal of this architecture is to automate the crawling and analysis of models as much as possible and to have components that can be run and evolved independently.

[3] The interface for keyword-based search is similar. It is accessed by clicking on the Keywords button.



```
package sm;
class StateMachine {
  val State[*] states;
}

class State {
  attr String[1] name;
}

class InitialState
   extends State {
}
```

stateMachine.ecore 218.49/5.76
No description available
24 elements 1 smells

sm.ecore 151.79/5.40
No description available
106 elements 1 smells

FSM.ecore 147.38/4.96
No description available
eclipse room modelling embedded realtime java c xtext emf cpp statemachine code-generation
342 elements 4 smells

sm.ecore 125.66/5.39
No description available
66 elements 0 smells

exercise_3.ecore 108.84/5.25

Fig. 4 User interface of MAR

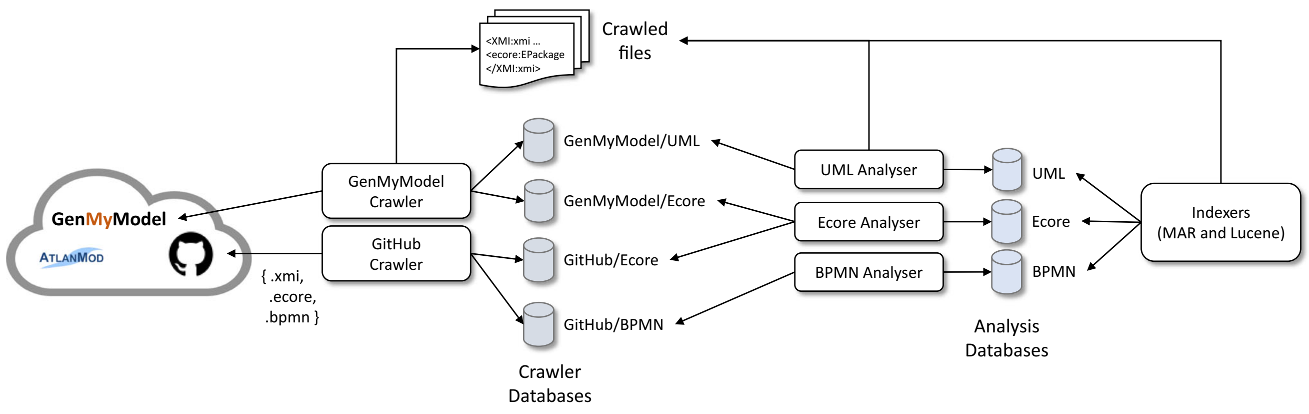


Fig. 5 Crawling and analysis architecture

3.1 Model sources

The first challenge is the *identification of model sources*. Models can be found in different locations, like source code repositories (e.g. GitHub, GitLab), dedicated model repositories (e.g. AtlanMod Zoo) or private repositories (e.g. within an enterprise). There is no standard mechanism for discovering these sources, and therefore this step is manual. Moreover, many times there is an additional step of identification of relevant model types along with the required technology to process them. For instance, to process Archimate models we needed to gather its meta-model[4] and integrate it appropriately in our system. We have included in MAR those sources and types of models that we were already aware of, or that we have learned through our interaction with the MDE community. We plan to include new sources and new types of models as we discover them. An important trait is that our search engine embeds all the knowledge that we have gathered about model sources and how to process them, so that it provides an effective means for reusing this knowledge by simply searching for models.

[4] https://github.com/archi-contribs/eclipse-update-site.

3.2 Crawling model sources

A search engine is more useful when it is able to provide access to a greater amount of resources. Crawling is the process of gathering models from sources such as model repositories, source code repositories and websites. Hence, our crawlers are a key element for MAR. An important challenge is that each repository or source of models has its own API or protocol for accessing its resources, or even no specific API at all (e.g. models listed in plain HTML pages). This means that there is typically little reusability when imple-

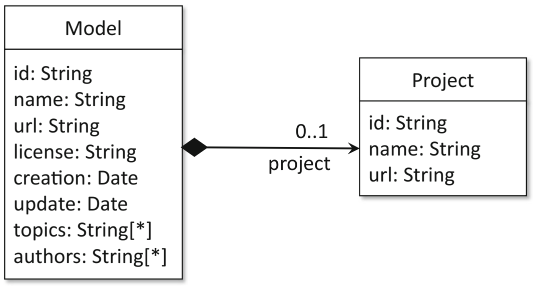


Fig. 6 Data model of the crawlers

menting crawlers, and for each type of repository a crawling strategy is needed.

In our architecture, each crawler accesses the remote resource and downloads the models, which are stored temporarily on a local disk. In addition, the metadata extracted for each model is stored in a database. In our design, for each pair of $(repository, meta - model)$ we have a different database. This means that each crawler knows how to extract a specific type of models (i.e. conforming to some meta-model) from a given repository.

We have implemented three crawlers: for GitHub, GenMyModel and the AtlanMod zoo. All crawlers maintain a database to store the metadata associated with each model. The data model is shown in Fig. 6.

For the AtlanMod meta-model zoo, we have implemented an HTML crawler which extracts the links to the meta-models and the associated information (description, topics, etc.).

For GenMyModel, we use its public API[5] to download the metadata of the public models it stores. The metadata contains references to the location of the XMI files. The steps are as follows:

- Download the complete metadata catalogue, which is available as a collection of JSON files which can be easily downloaded from the API endpoint.
- Download models per type using the metadata catalogue which contain hyperlinks to the actual XMI files.
- Process the metadata files and insert the relevant metadata into our crawler data model.

For GitHub, we use PyGithub[6] to interact with the GitHub Search API. The GitHub API imposes two main limits which requires special treatment. First, there is a maximum of 15,000 API Requests per hour (for OAuth users). Moreover, it is recommended to pause for a few seconds between searches. Secondly, it returns up to 1024 results per search. This means, that we need to add special filtering options to make sure that each search call always returns fewer than 1024 models. Our current approach is to add a minimum and maximum size to the search query, so that in each call we only obtain models within this range. Then, we move this sliding window adjusting the minimum and maximum sizes dynamically according to the number of returned results. In practice, these limitations imply that the GitHub crawler may take several days to perform its job.

Table 1 summarizes the types, number and sources of the models that we have crawled so far. The highest number of models corresponds to Ecore, UML and BPMN models. This is so as they are widely used notations. They have been obtained from GitHub, GenMyModel and the AtlanMod Zoo. We could also obtain a relatively large number of models from Archimate and Petri nets (in PNML format) since they are also well-known types of models. We also downloaded Sculptor models which is a textual DSL to generate applications. This shows that our system is not limited to XMI serialized files. Finally, we converted Simulink models using Massif [4] from a curated dataset [20].

3.3 Model analysis

The following step in the process is model analysis. It refers to the task of processing the gathered models, check their validity and compute additional information such as statistics and quality measures.

The task of checking the validity of a model is more involved than it might seem at first sight, notably if one is interested in a fault-tolerant process. The underlying problem is that given a crawled model, blindly loading and validating it may crash the analyser (i.e. out of memory error, bugs in EMF, invalid formats, etc.). Therefore, the implementation must take this into account and load the model in a separate process, so that it is possible to capture the situation that the process dies. If this is the case, the main execution thread can catch the error and consider the model invalid.

The analysis of a model consists of four main steps, and the results of the analysis are stored in the data model shown in Fig. 7.

- Compute the MD5 hash of the file contents and check if it is duplicated with respect to an already analysed model. If so, mark its status as duplicated.
- Check if the model can be loaded (e.g. using EMF) and check its validity (e.g. using EMF validators). If the model is valid, mark as such so that it can be later used in the indexing process.
- Compute statistics about the model. The most basic measure is the number of model elements, but we also compute other specific metrics for some model types like Ecore, UML, etc.

[5] https://app.genmymodel.com/api/projects/public.

[6] https://github.com/PyGithub/PyGithub.

Table 1 Summary of the type of models crawled by MAR as December 23, 2021

	Source	Crawled	Duplicates	Failed	Indexed	Observations
Ecore	GitHub	67,322	46,199	341	20,782	Based on the standard Ecore meta-model
	GenMyModel	3987	3	27	3957	
	AtlanMod	304	1	4	299	
UML	GitHub	53,082	7282	1699	44,101	Based on the Eclipse UML implementation
	GenMyModel	352,216	143	23,836	328,237	
BPMN	GenMyModel	21,285	0	200	21,085	Based on the EMF BPMN2 meta-model
Archimate	GitHub	496	77	106	313	Based on the Archi meta-model
PNML	GitHub	3291	1576	1044	671	Based on PNML framework
Sculptor	GitHub	188	88	0	88	An application generator based on Xtext
RDS	GenMyModel	91,411	108	515	90,788	Refers to entity/relationship diagrams
						Meta-model semi-automatically inferred
Simulink	Dataset [20]	200	0	0	200	Converted to EMF using Massif [4]
Total	–	593,582	55,477	27,972	510,321	–

https://www.omg.org/spec/BPMN/2.0/
https://github.com/archi-contribs/eclipse-update-site
https://pnml.lip6.fr/

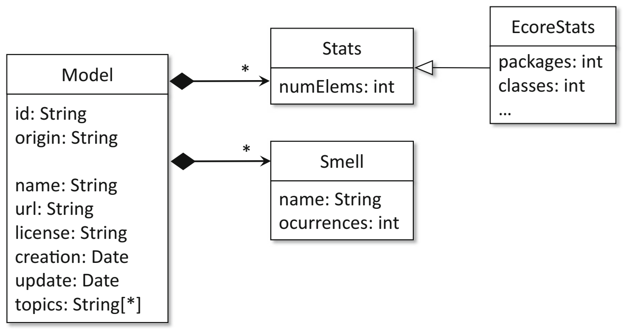


Fig. 7 Data model of the analyser

– Perform a quality analysis. We currently analyse potential smells for Ecore models. For this, we have implemented the smells presented in [41].

Table 1 summarizes the number of duplicated, failed and actually indexed files.

3.4 Supporting different types of models

In MAR, we aim at supporting different types of models crawled from different sources. This is a challenge since it is not possible to analyse and process all types of models using a single approach. For instance, the code to load and analyse PNML files is different from the one used to load and analyse Ecore models. Moreover, the computation of metrics, smells and visualizations is specific to each kind of models.

To partially overcome this issue, the design of MAR is modular and extensible, in the sense that support for new model types can be added just by creating new modules and adding some configuration options. Implementation-wise the addition of a new model type to MAR involves the following steps:

– Identify which modelling framework is needed to load the models. For instance, most models serialized as XMI can be loaded with EMF if its Ecore meta-model is known. Sometimes, the models are serialized using a textual syntax, for instance using Xtext. In this case, the loading phase requires invoking the corresponding Xtext facilities.
– A new analyser must be created, which is in charge of loading the model, invoke the corresponding validator (if available) or implement new validation code. Moreover, the analyser may include the logic to compute metrics and smells. The results are stored in the data model shown in Fig. 7.
– The analyser must be registered in a global registry which contains pairs of *(model type, factory object)*. Given a certain model, its model type is looked up in the registry and the factory object instantiates the validator.
– To support new types of visualizations, a similar approach is used. Given a model type, a visualizer must be registered. A visualizer takes a model, loads it and generates an image. We currently target PlantUML, but other approaches are possible, like integrating the facilities provided by Picto [39].

In general, adding a new model type is relatively straightforward if it is based on EMF or an EMF-based framework (like Xtext) because MAR already provides facilities to implement the above steps.

3.5 Indexers

The models that have been deemed valid by the corresponding analyser are fed into the indexers in order to store them in a form suitable for fast query resolutions. We have two indexers: for keyword-based queries and for example-based queries. These search modes are explained in the two following sections.

4 Keyword-based queries

MAR provides a keyword-based search mode in which the user just types a set of keywords and the system returns models that contain these input keywords. To implement this functionality, we rely on existing techniques related to full-text search, which has been widely studied in the literature [59]. In our context, we assume that the string attributes of the models that we want to expose in the search engine act as keywords. Thus, we consider each model to be a document consisting of a list of keywords. The expectation is that a model contains *names* which reflects its intention and the user can reach relevant models by devising keywords similar to those appearing in the models.

Implementation-wise, we use Apache Lucene [2] to implement this functionality, since it provides facilities for pre-processing documents, to index the documents and to perform the keyword-based search. To integrate the crawled models with Lucene, each model goes through the following pipeline:

1. **Term extraction**. We extract all the string attributes (e.g. ENamedElement.name in Ecore) from the model. For instance, the text document for the model shown in Fig. 3 would be:

   ```
   StateMachine states State name
   InitialState FinalState
   ```

2. **Camel case tokenization**. Each camel case term is divided into lowercase subterms. For example, the previous terms are transformed into:

   ```
   state machine states state name
   initial state final state
   ```

3. **Word pre-processing**. Full-text search engines, like Lucene, typically perform a pre-processing step which include techniques like stop word removal (i.e. remove common tokens like *to* and *the*) and stemming to reduce inflectional forms to a common base form (e.g. transforming a plural word like *states* into its singular form *state*). Using the Porter Stemming Algorithm [49], we obtain:

   ```
   state machin state state name
   initi state final state
   ```

4. **Storing of the bag of terms**. The list of subterms is transformed into pairs $(term, freq)$ where $freq$ is the frequency of $term$ in the list. This step is done internally by Lucene. For example:

   ```
   (state,5) (machin,1) (name,1)
   (initi,1) (final,1)
   ```

   Finally, this set of pairs are stored in the Lucene inverted index.

Moreover, in each document we also include metadata about the corresponding model obtained in the crawling process, such as the description and topics. Then, the documents are indexed in an inverted index managed by Lucene.

Regarding the search process, given a set of keywords as query, Lucene accesses the inverted index and it ranks the models according to their scores with respect to the query (it uses the Okapi BM25 [51,59] scoring function).

A feature of our keyword-based index is that all models are indexed together. This means that a query like *state machine* may return models of different types like Ecore meta-models representing state machines, but also, e.g. UML models in which the keywords may appear. Then, the user could use the UI filters to select the concrete type of models.

The main disadvantage of keyword-based queries is that the structure of the models is not taken into account. Thus, a query like *state transition* could not distinguish whether the user is interested in models containing classes named *State* or *Transition*, or if they refer to references like *states* or *transitions*. Thus, one way to improve the precision of the search is by providing model fragments as queries so that the structure of the models can be taken into account.

5 Query-by-example

MAR provides support for another search mode called *query-by-example*. In this case, the user creates a small example model as the query and the system looks up models which have parts similar to this example. In this section, we will explain how MAR performs approximate structural matching to determine relevant models. This is achieved in three steps:

1. Transforming each model into a multigraph.
2. Extracting paths from the graph.
3. Comparing the paths extracted from the query and the paths extracted from each model in the repository.

5.1 Graph construction

In our approach we transform models into a labelled multigraph with the following form:

$$G = (V, E, f, \{\mu_V^1, \mu_V^2\}, \{\mu_E\})$$

where:

- V is the set of vertices, E is the set of edges and $f : E \longrightarrow V \times V$ is a function which defines the source and the target of each vertex.
- $\mu_V^1 : V \longrightarrow$ {attribute,class} is a vertex labelling function that indicates if a vertex represents an attribute value or the class name of an object.
- $\mu_V^2 : V \longrightarrow L_V$ where $L_V = L_A \cup L_C$. This is a vertex labelling function that maps vertices to a finite vocabulary which has two components:
  - L_A corresponds to the set of attribute values. If $v \in V$ is an attribute then $\mu_V^2(v) \in L_A$.
  - L_C corresponds to the set of names of the different classes. If $v \in V$ is a class then $\mu_V^2(v) \in L_C$.
- $\mu_E^2 : E \longrightarrow L_E$ where $L_E = L_R \cup L_{AR}$. This is an edge labelling function which has two components.
  - L_R corresponds to the set of names of the different references. An edge whose label belongs to this vocabulary connects two classes.
  - L_{AR} corresponds to the set of names of the different attributes. An edge whose label belongs to this vocabulary connects a class vertex and an attribute vertex.

The procedure followed to transform a model into a multigraph is shown in Algorithm 1. This procedure receives a model as input, its meta-model and the set of all meta-classes ($\mathcal{C}$), references ($\mathcal{R}$) and attributes ($\mathcal{A}$) that will be taken into account (i.e. those model elements whose meta-types are not in these sets are filtered out). The output is a directed multigraph whose label languages L_R, L_{AR} and L_C are determined by $\mathcal{R}$, $\mathcal{A}$ and $\mathcal{C}$, respectively. In the output multigraph we can distinguish between two type of nodes: nodes labelled with *class* in μ_1 (we call them *object class* nodes) and nodes labelled with *attribute* in μ_1 (we call them *object attribute* nodes). For instance, the model in Fig. 3 is transformed into the graph shown in Fig. 8 using Algorithm 1. In this example, we consider as input for the algorithm:
$\mathcal{R}$ ={all references}
$\mathcal{A}$ = {ENamedElement.name, EClass.abstract, EReference.containment}
$\mathcal{C}$ = { ENamedElement }

```
Data: input model, meta-model, $\mathcal{C}$, $\mathcal{R}$ and $\mathcal{A}$
Result: labelled multigraph
while not all model objects are visited do
    read an object $o$ from the input model;
    if $o$ does not have a associated class node and the meta-class of $o$ belongs to $\mathcal{C}$ or it inherits from a meta-class of $\mathcal{C}$ then
        add node $v$ (associated with $o$) labelled with class and the name of the meta-class;
    forall the attribute of $o$ do
        if attribute belongs to $\mathcal{A}$ then
            add node $w$ (associated with the attribute value);
            label $w$ with attribute and the value of the attribute;
            add edge $(v, w)$ labelled with the name of the attribute;
            add edge $(w, v)$ labelled with the name of the attribute;

    end
    forall the references of $o$ to another object $o'$ do
        if if the reference belongs to $\mathcal{R}$ then
            if if $o'$ has not been visited then
                add node $u$ (associated with object $o'$), labelled it with class and its meta-class;
            add $(v, u)$ labelled with the reference name ($u$ is the node associated with $o'$);

    end
    mark $o$ as visited;
end
```
Algorithm 1: Procedure that transforms a model into a labelled multigraph.

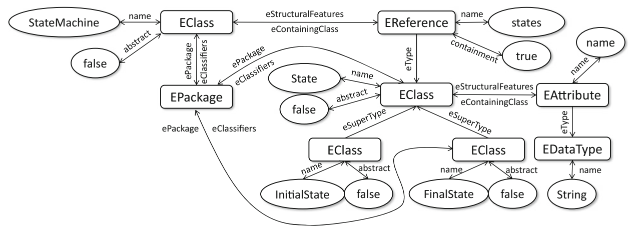


Fig. 8 Multigraph obtained from the model in Fig. 3. Object class nodes are depicted as rounded rectangles, while attribute nodes are depicted as ovals

This graph is the interface of MAR with the different modelling technologies, that is, MAR can process any model that can be transformed to this graph. Up to now, we have implemented the algorithm for EMF, but we foresee other implementations for other types of models.

5.2 Path extraction

Once the graph associated with a model has been created, we perform a path extraction process. The underlying idea is to use paths and subpaths found in the graph as the means to compare the example query against the models in the repository. In this context, the term **bag of paths** (BoP) refers to the multiset of paths that have been extracted from a multigraph. A path that belongs to a BoP can have one of the following forms:

– Path of null length: $(\mu_V^2(v))$ or
– Path of length greater than zero:

$$(\mu_V^2(v_1), \mu_E^2(e_1), \ldots, \mu_V^2(v_{n-1}), \mu_E^2(e_{n-1}), \mu_V^2(n)),$$

with $n \geq 2$.

This definition of BoP establishes the form of the elements that should belong to this multiset and does not enforce which paths must be computed. Therefore, extraction criteria must be defined depending on the application domain. We follow the same criteria as our previous work [40] in which only paths of these types are considered:

– All paths of zero length for objects with no attributes. In Fig. 8, the only object without attributes is the *EPackage* object (because we did not establish the package name). Therefore, the only example of this type of path is $\left(\boxed{\text{EPackage}}\right)$.
– All paths of unit length (paths of length one) between attribute values and its associated object class. In Fig. 8, examples of this type of paths are $\left(\boxed{\text{states}}, \overrightarrow{\text{name}}, \boxed{\text{EReference}}\right)$, $\left(\boxed{\text{State}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}\right)$, $\left(\boxed{\text{false}}, \overrightarrow{\text{abstract}}, \boxed{\text{EClass}}\right)$, etc.
– All simple paths with length less or equal than a threshold (paths of length 3 or 4 are typical enough to encode the model structure, as determined empirically in Sect. 7) between attributes and between attributes and objects without attributes. In Fig. 8, examples of this type of paths are $\left(\boxed{\text{states}}, \overrightarrow{\text{name}}, \boxed{\text{EReference}}, \overrightarrow{\text{containment}}, \boxed{\text{true}}\right)$, $\left(\boxed{\text{StateMachine}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{eStructuralFeatures}}, \boxed{\text{EReference}}, \overrightarrow{\text{name}}, \boxed{\text{states}}\right)$, $\left(\boxed{\text{EPackage}}, \overrightarrow{\text{eClassifiers}}, \boxed{\text{EClass}}, \overrightarrow{\text{name}}, \boxed{\text{InitialState}}\right)$, $\left(\boxed{\text{StateMachine}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{eStructuralFeatures}}, \boxed{\text{EReference}}, \overrightarrow{\text{eType}}, \boxed{\text{EClass}}, \overrightarrow{\text{name}}, \boxed{\text{State}}\right)$, etc.

The underlying idea is that paths of zero and unit length encode the data of an object (its internal state). For instance, the path $\left(\boxed{\text{State}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}\right)$ means that the model has an EClass whose name is State. On the other hand, paths of length greater than one encode the model structure. For instance, the path $\left(\boxed{\text{StateMachine}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{eStructuralFeatures}}, \boxed{\text{EReference}}, \overrightarrow{\text{name}}, \boxed{\text{states}}\right)$ means that the query expects the existence of a class named StateMachine with a reference named states.

5.3 Scoring

Given a query q (a model) and a repository of models $\mathcal{M}$, our goal is to compare each model of the repository with the query. To achieve this, all the models in $\mathcal{M}$ are transformed into a set of multigraphs and then paths are extracted obtaining a set of bags of paths, $BoPs = \{BoP_1, \ldots, BoP_n\}$. The query q follows the same pipeline and it is transformed into BoP_q. To perform the comparison between BoP_q and each bag BoP_i of $BoPs$, that is, to compute the score $r\left(BoP_q, BoP_i\right)$, we will use the adapted version of the scoring function Okapi BM25 [51,59] used in [40]:

$$\sum_{\text{❶}\boxed{\omega \in BoP_q \cap BoP_i}} \frac{c_\omega(q)(z+1)c_\omega(i)}{c_\omega(i)+z\,\text{❸}\boxed{\left(1-b+b\dfrac{|BoP_i|}{avdl}\right)}}\,\text{❷}\boxed{\log\left(\frac{n+1}{df(\omega)}\right)}, \tag{1}$$

where $1 \leq i \leq n$, $c_\omega(q)$ is the number of times that a path ω appears in a bag BoP_q, $c_\omega(i)$ is the number of times that a path ω appears in a bag BoP_i, $df(\omega)$ is the number of bags in $BoPs$ which have that path, $avdl$ is the average of the number of paths in all the $BoPs$ in the repository and $z \in [0, +\infty)$, $b \in [0, 1]$ are hyperparameters. This function takes $BoP_i \in BoPs$ and BoP_q, and returns a similarity score (the higher, the better). We have chosen this scoring function because it has three useful properties for our scenario:

– It takes into account the paths that are in both bag of paths (❶ box of Eq. (1)).
– It takes into account the paths that are very common in the bags of the entire repository (❷ box of Eq. (1)).
– It penalizes models which have a large size (❸ box of Eq. (1)) to prevent the effect caused by such models having significantly more matches. This is controlled by the hyperparameter b. If $b \to 1$, larger models have more penalization. We take $b = 0.75$ as default value.

It is important to remark that the hyperparameter z controls how quickly an increase in the path occurrence frequency results in path frequency saturation. The bigger z is, the slower saturation we will have. We take $z = 0.1$ as default value.

Therefore, to perform a query, for each element in $BoPs$ its score with respect to BoP_q is computed. Then, the repository is sorted according to the computed scores and the top k models are retrieved (the k models with highest score). While this is a straightforward approach, it is not efficient. Instead, an index structure is needed in order to make the scoring step scalable and efficient. Next section presents the implementation of this structure.

6 Indexing models for query-by-example

In a typical text retrieval scenario, an indexer organizes the documents in an appropriate data structure for providing a fast response to the queries. In our case, which is a model retrieval scenario, the indexer organizes models instead of documents. As in every search engine, the indexing module is crucial when we look for scalability and the management of large repositories.

6.1 Inverted index

The main data structure used by the indexer is the inverted index [59]. In a text retrieval context, the inverted index consists of a large array in which each row is associated with a word in the global vocabulary and points to a list of documents that contain this word.

In our context, instead of words we have paths and instead of documents we have models. This is a challenge that we have to deal with since it causes two problems:

- *Large amount of paths per query*. In contrast to a text retrieval scenario in which a query is composed by a small number of keywords, in query-by-example, a small model/query can be composed by an order of hundreds of different paths.
- *Huge inverted index*. In a text retrieval scenario, the number of words in the vocabulary has an order of hundred of thousands, whereas, in model retrieval, we have an order of millions different paths.

These two issues translate in practice into a lot of accesses per query to a huge inverted index. Therefore, the design of an appropriate and more complex structure beyond the basic inverted index is needed. The main ingredients that we use to tackle this are:

- The inverted index is built over HBase [1]. This database has a distributed nature. Therefore, the table that will be associated with the inverted index is split into chunks and they may be distributed over the nodes of a cluster. This design decision makes the system scalable with respect to the size of the index.
- Special HBase schema. We have designed an HBase schema to accommodate an inverted index particularly oriented to resolve path-based queries. The underlying idea is to split the paths in two parts. Doing this, the number of accesses per query to the inverted index is reduced. This is explained next.

6.2 Inverted index over HBase

We use Apache HBase [1,30] to implement the inverted index. HBase is a sparse, distributed, persistent, multidimensional, sorted, map database inspired by Google BigTable [19]. The HBase data model consists of tables, and each table has rows identified by row keys, each row has some columns (qualifiers) and each column belongs to a column family and has an associated value. A value can have one or multiple versions identified by a timestamp. Therefore, given a table, a particular value has 4 dimensions:

$$\text{(Row, Column Family, Column Qualifier, Timestamp)} \\ \longrightarrow \text{Value}$$

Due to the design of HBase, the Column Family and Timestamp dimensions should not have a large number of distinct values (i.e. three or four distinct values). On the other hand, there is no restriction in the Row and Column Qualifier dimensions.

Regarding reading operations, HBase provides two operations: *scan* (iterate over all rows that satisfy a filter) and *get* (random access by key to a particular row). With respect to writing, the main operation is *put*, to insert a new value given its dimensions (row, column family and column qualifier). This operation does not overwrite if multiple versions are allowed.

Designing an HBase schema for an inverted index can be relatively straightforward: each path is a row key and the models that contain this path are columns whose qualifier is the model identifier. However, this approach suffers from one of the problems that we have explained previously: lots of accesses (*get*) per query. Therefore, in order to avoid this problem, we use the schema presented in [40]. Each path is split into two parts: the prefix and the rest. The prefix is used as row key whereas the rest is used as column qualifier. The split point of the path depends on whether it starts with an attribute node (the prefix is the first sub-path of unit length) or an object class node (the prefix is the first node). The value associated with the column is a serialized map which associates the identifier of the models which have this path with a tuple composed by the number of times the path appears in the models and the total number of paths of the models. This tuple is used by the system to compute the scores in an efficient manner. Therefore, this schema has the following form:

$$\text{(Prefix, Column Family, Rest, Timestamp)} \longrightarrow \\ \text{Map ModelID} \rightarrow \text{(number of times, model size)}$$

Thus, now a path is the concatenation of the prefix (row id) and the rest (column qualifier). The column family is

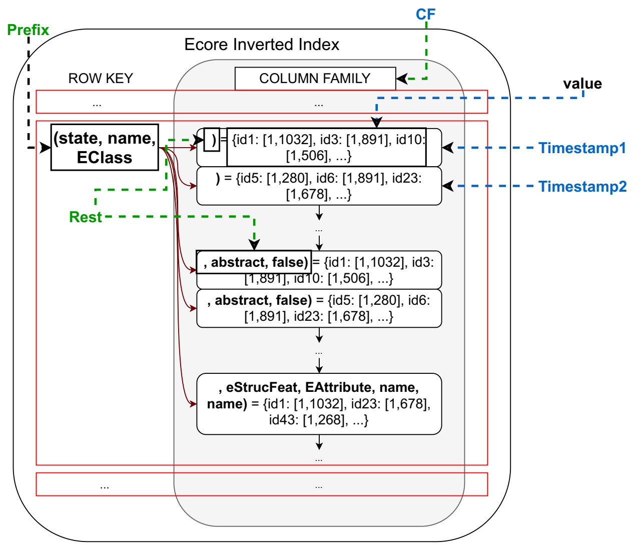

Prefix
CF
Ecore Inverted Index
ROW KEY
COLUMN FAMILY
value
...
(state, name, EClass
) = {id1: [1,1032], id3: [1,891], id10: [1,506], ...}
Timestamp1
) = {id5: [1,280], id6: [1,891], id23: [1,678], ...}
Timestamp2
Rest
, abstract, false) = {id1: [1,1032], id3: [1,891], id10: [1,506], ...}
, abstract, false) = {id5: [1,280], id6: [1,891], id23: [1,678], ...}
, eStrucFeat, EAttribute, name, name) = {id1: [1,1032], id23: [1,678], id43: [1,268], ...}

Fig. 9 HBase schema of the inverted index

constant for all the paths in a table, and the number of different timestamps is limited by the number of versions allowed in the column family.

This schema is illustrated in Fig. 9. As an example we use three paths:

1. $\left(\textcircled{\text{state}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}} \right)$
2. $\left(\textcircled{\text{state}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{abstract}}, \textcircled{\text{false}} \right)$
3. $\left(\textcircled{\text{state}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{eStructuralFeatures}}, \boxed{\text{EAttribute}}, \overrightarrow{\text{name}}, \textcircled{\text{name}} \right)$

For instance, the second path $\left(\textcircled{\text{state}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{abstract}}, \textcircled{\text{false}} \right)$ is split into prefix (state, name, EClass (row key) and rest , abstract, false) (column qualifier). The first and third paths have the same common prefix and thus share the same row key. The column qualifiers associated with the row key are the rest parts, that is,), , abstract, false) and eStructuralFeature, EAttribute, name, name). Each column qualifier has an associated value, which is a map whose keys are the model identifiers (id$x$ where x is a number) and whose values are the pairs $[a, b]$ where a is the number of times the path appears in the id$x$ model and b is total number of paths of the id$x$ model. Moreover, in this example, the paths have two versions associated.

Altogether, this approach reduces the number of accesses to the index because in a given model there are many paths with the same prefix, and thus with one access it is possible to retrieve all of them.

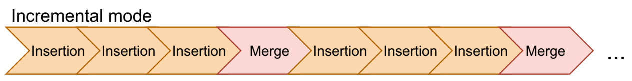


Fig. 10 Possible sequence of insertion and merge operations in the incremental mode. In this case, the number of distinct versions that the column family allows is three

6.3 Indexing process

Given a large repository of models of the same type (i.e. conforms to the same meta-model), indexing is the process of building the HBase table that represents the inverted index and follows the schema previously explained. Our search engine has two indexing modes:

– Batch: once the index is constructed, adding more models to the index is not allowed. Instead, if new models are discovered, the whole index needs to be deleted and all the models are indexed together. To create an index in batch mode, we process each model in the repository in turn, transform each model into a graph, compute paths, group paths by prefix and invoke HBase put operations to fill the index table.
– Incremental: adding more models to the index is allowed. We focus on this mode in the rest of the section.

In order to implement the incremental mode the indexing process is composed by two subprocesses:

– Insertion: it adds new models to the inverted index. In particular, this procedure receives as input a set of models, obtains its bags of paths and stores them in a HBase table. A detailed explanation is given in Sect 6.3.1.
– Merge: After a sequence of insertion operation has been performed, the cells in the HBase table associated with the inverted index may have a set of versions. The merge procedure is in charge of merging all these versions of a cell into a single one. A detailed explanation is given in Sect 6.3.2.

6.3.1 Insertion

This subprocess is in charge of adding new models to the inverted index. It is implemented by a Spark [58] script which does the following (see Fig. 11):

1. Read and distribute all the models (this corresponds to label ❶ in Fig. 11). Given a repository of n models, they are read and loaded in memory.
2. Obtain graphs, extract and process the paths (this corresponds to label ❷ in Fig. 11). The set of loaded models $\{m_1, \ldots, m_n\}$ is transformed into the set $\{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$

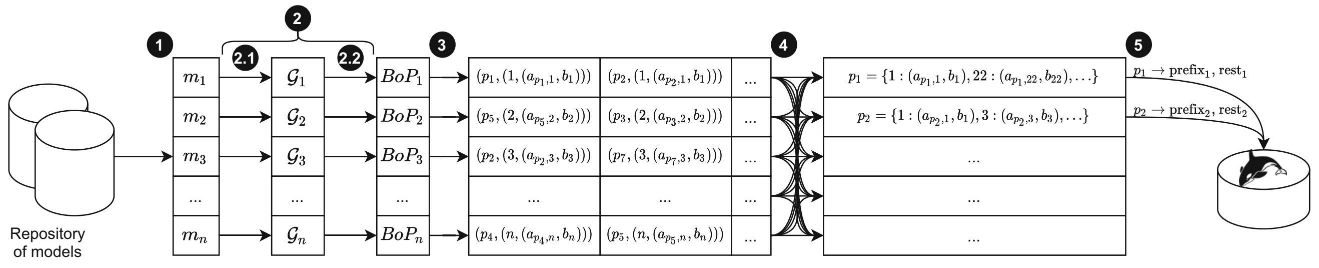


Fig. 11 Insertion subprocess

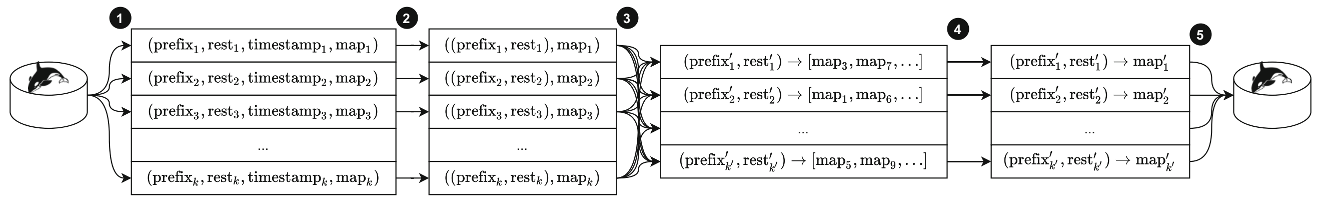

(prefix$_1$, rest$_1$, timestamp$_1$, map$_1$)
((prefix$_1$, rest$_1$), map$_1$)
(prefix$'_1$, rest$'_1$) $\rightarrow$ [map$_3$, map$_7$, ...]
(prefix$'_1$, rest$'_1$) $\rightarrow$ map$'_1$

Fig. 12 Merge subprocess

using Algorithm 1 (label **2.1** in Fig. 11). After that, the path extraction process is performed getting the set of bags of paths $\{BoP_1, \ldots, BoP_n\}$ (label **2.2** in Fig. 11).

3. Emit key-value pairs. The key is the processed path and the value is a tuple (model id,statistics) (label **3**). For each BoP and for each distinct path that belongs to this BoP we emit a pair key-value with the form $(p, (\text{id}, (a, b)))$ where p is the path, id is the model identifier that the BoP is extracted from, a is the number of times the path appears in the BoP and b is total number of paths of the BoP. Here, p is the key and $(\text{id}, (a, b))$ is the value.
4. Group by key and merge the list of values (from the key-value pairs) into a serialized map (label **4**). We group the pairs emitted in the previous step according to the path (key) and the list of values associated with this key is transformed into a map of the form $\{id_1 : (a_1, b_1), id_2 : (a_2, b_2), \ldots\}$. For instance, as we can observe in Fig. 11, the map associated with the p_2 path has the elements $1 : (a_{p_2,1}, b_1)$ and $3 : (a_{p_2,3}, b_3)$ since BoP_1 and BoP_3 contain the path p_2.
5. Split the paths and store them in the HBase table (label **5**). Each path is split into a prefix and the rest according to the HBase schema presented in Sect. 6.2 and it is stored in the inverted index table with its associated map using the HBase *put* operation.

This step is similar in both the batch and the incremental modes. The only difference is that in incremental mode a timestamp is attached to each inserted value to allow multiple versions. This is possible because HBase allows multiple versions in column families. Therefore, when a *put* operation is executed and the key already exists the last value is not removed, but a new version is added.

6.3.2 Merge

HBase is able to resolve read operations in the presence of versions. However, the number of versions of a column family is limited by an upper bound in order to control the size of the HFile (the distributed file that stores the column family). Therefore, it is necessary to perform a merge operation before the number of versions grows too large (i.e. in our implementation we set the maximum number of versions to four).

According to our HBase schema, a value associated with a version in a column will be a serialized map of models that contains a path (the path is given by the concatenation of row key and the column qualifier). Therefore, these versions can be merged into a single one (i.e. these maps can be merged into a single one). The merge subprocess is a Spark [58] script which does the following (Fig. 12):

1. Read the table associated with the inverted index and extract all tuples (prefix, rest, timestamp, map) (label **1** in Fig. 12).
2. Ignore timestamp dimension and emit pairs key-value (label **2** in Fig. 12). Here, the key is (prefix, rest) and the value is the serialized map.
3. Group by the key, that is, coordinates (prefix, rest). This corresponds to step **3** in Fig. 12 in which (prefix$'_i$, rest$'_i$) are the distinct keys and each [map] is the list of maps associated with a key.
4. Merge the list of maps associated with a key (prefix$'$, rest$'$) and generate a new map. This step corresponds to the label **4** in Fig. 12. Here, (prefix$'_i$, rest$'_i$) are the distinct keys and map$'_i$ are the merged map.

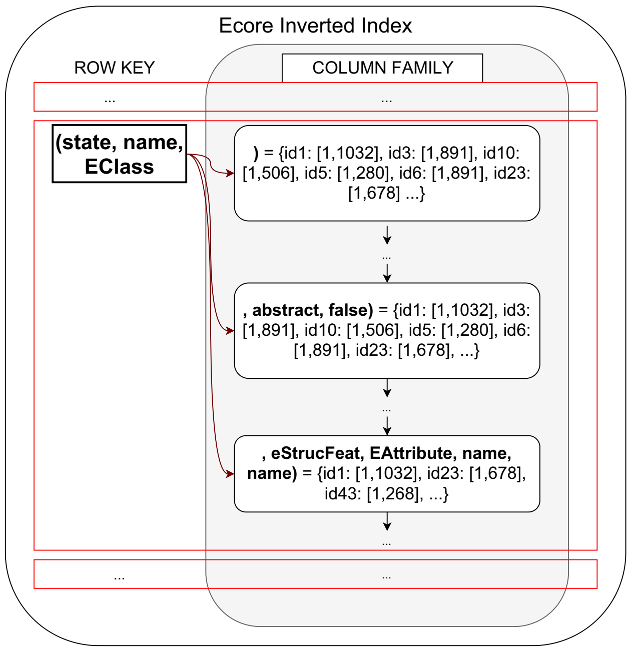

Ecore Inverted Index
ROW KEY
COLUMN FAMILY
...
(state, name, EClass
) = {id1: [1,1032], id3: [1,891], id10: [1,506], id5: [1,280], id6: [1,891], id23: [1,678] ...}
, abstract, false) = {id1: [1,1032], id3: [1,891], id10: [1,506], id5: [1,280], id6: [1,891], id23: [1,678], ...}
, eStrucFeat, EAttribute, name, name) = {id1: [1,1032], id23: [1,678], id43: [1,268], ...}

Fig. 13 HBase table obtained after the application of the merge operation in the table represented by Fig. 9

5. Store the result in a new HBase table (*put* operation) and delete the old one. This corresponds to the label ❺ in Fig. 12.

For example, if the merge operation is applied to the HBase table represented in Fig. 9, the table represented by Fig. 13 will be obtained. Looking at these two figures, we can appreciate that maps associated with the same path but with different timestamps/versions have been merged into a single map.

To summarize, the batch mode is adequate when the managed repository is immutable (or very unlikely to change) or when it is small and it is very fast to re-index everything after a change. Instead, the incremental one is interesting when the repository may change (this is the case of GitHub or GenMyModel) or when it is very large and it is better to index in chunks of models to avoid exhausting the available heap memory. We evaluate the indexing time in both models in the following section.

7 Evaluation

In this section, we evaluate MAR from three perspectives. First, we evaluate the indexing procedure in terms of its performance, comparing batch mode and incremental mode. Second, we evaluate the search precision and finally we evaluate the query response time. Moreover, the source code of MAR is available at http://github.com/mar-platform/mar so that others can install it locally and perform experiments with it.

7.1 Indexing performance

As the number of crawled models grows, the performance of the indexer module becomes increasingly important. We have measured the time that the incremental and batch modes take, and compared them. The experiments have been run in an AMD Ryzen 7 3700X, 4.4Ghz, 8 core, 16 threads with 32 GB of RAM. The Spark process is run locally with a driver memory of 8GB.

To evaluate the effect of the repository size in the indexing time, we use the non-incremental mode and split a repository of 17.000 Ecore models in 17 "incremental" batches, that is, a batch with the first 1000 models, a second batch with the first 2000 models and a last batch with the complete 17.000 models. We have first shuffled the models to have a uniform distribution of the different sizes. We index each batch, cleaning the database index between each run, and record the indexing time. Fig. 14 shows the results. As the number of models increases, the indexing times grow *linearly*. The time taken to index the full repository is about 12 minutes.

To evaluate the incremental mode we use the same repository of 17.000 Ecore models, but this time, we create batches of 1000 models each. The experiment consists of indexing each batch in turn, and after 4 insertions (5 when the index is empty) we invoke the merge operation. We record the time of both the insertion and merge operations. Figure 15 shows the results. The insertion times are in a range of about 40–75 s per batch (the variations due to some models being much larger than others). The merge operation is more costly and it grows linearly with the size of the index.

Comparing the total indexing time, the batch mode takes 724 s ($\sim$ 12 min), whereas the incremental mode takes 1410 s ($\sim$ 23 min) when adding up all the insertion and merge times. Thus, there is a penalty when using the incremental mode due to the merge operation. However, given that merge is only done from time to time, it pays off to use the incremental mode for large collections of models that require updates.

7.2 Search precision

We are interested in analysing the ability of MAR to match and rank in the first positions relevant models, as well as the effect of the different parameters on this task (i.e. the path length and the kind of meta-model elements considered in the graph construction). The evaluation of a search engine is known to be difficult since the notion of result relevance is subjective [59]. To address this issue, we follow the same idea proposed in [40], that is, we simulate a user who has in mind a concrete model by taking a model from the repository and deriving its associated query using mutation operators.

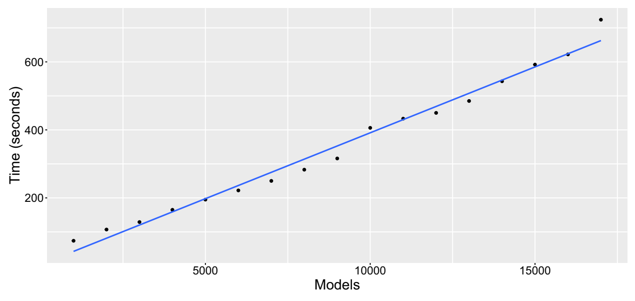


Fig. 14 Non-incremental mode performance. The x-axis indicates the number of models that the process has to index and the y-axis the time in seconds. MAR scales smoothly as the size of the index increases

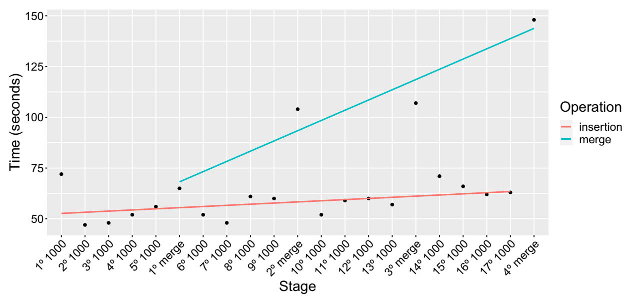


Fig. 15 Incremental mode performance. The x-axis indicates the stages and the y-axis the time in seconds. In case of merge operations, MAR scales smoothly as the size of the index increases. In case of insertions, the time used is constant since it only depends on the number of models considered and their size

This query is a shrunk version of the original model which includes some structural and naming changes, which is called *query mutant*. The underlying idea is that, for each generated query mutant, we already know that the mutated model is precisely the model that we expect the search to return in the first position. This type of search in which there is only one relevant model for each query is called *known item search* [59].

We use the repository of 17.690 Ecore meta-models used in [40] and we reuse the mutation operators already implemented for this previous work. We describe the operators here for the sake of completeness. For each meta-model, we apply in turn the mutation operators summarized in Table 2. First, we identify a potential root class for the meta-model and we keep all classes within a given "radius" (i.e. the number of reference or supertype/subtype relationships which needs to be traversed to reach a class from the root), and the rest are removed. All EPackage elements are renamed to avoid any naming bias. From this, we apply mutants to remove some elements (mutations 2–6). Next, mutation #7 is intended to make the query more general by removing elements whose name is very specific of this meta-model and it is almost never used in other meta-models (in text retrieval terms the element name has a low document frequency). Finally, to implement mutation #8 we have applied clustering (k-means), based on

Table 2 Mutations operators used to simulate queries

	Mutant	Description
1	Extract connected subset	Select a root element and pick classes reachable via references or subtype/supertype relationships, up to a given length.
2	Remove inheritance	Remove a random inheritance link (up to 20%).
3	Remove leaf classes	Remove classes, but prioritize those which are farther from the root element (up to 30%)
4	Remove references	Remove a random reference (up to 30%).
5	Remove enumeration	Remove random enumerations or literals (50%).
6	Remove attributes	Remove a random attribute (up to 30%).
7	Remove low-df classes	Remove elements whose name is "rare".
8	Rename from cluster	Replace a name by another name corresponding to element belonging to the same meta-model cluster (up to 30% of names).

the names of the elements of the meta-models, in order to group meta-models that belong to the same domain. In this way, this mutant attempts to apply meaningful renamings by picking up names from other meta-models within the same cluster. We apply this process with different radius configurations (5, 6 and 7) and we discard mutants with less than 3 classes or with less references than $|classes|/2$. Using this strategy we generate 1595 queries.

As a baseline to compare with, we use a pure text search engine. The full repository is translated to text documents which contain the names of each meta-model (i.e. property ENamedElement.name). These documents are indexed and we associate the original meta-model to the document. Similarly, we generate the text counterparts of the mutant queries. On the other hand, we run MAR with 6 different configurations. In particular, two dimensions will be used: the maximum path length and the attributes considered (that is, the set $\mathcal{A}$ in Algorithm 1). We will consider the path lengths 2, 3 and 4 and these two versions of $\mathcal{A}$:

- $\mathcal{A}_{\text{all}}$ = {ENamedElement.name, EClass.abstract, EReference.containment, ETypedElement.upperBound, ETypedElement.lowerBound}
- $\mathcal{A}_{\text{names}}$ = {ENamedElement.name}

The sets $\mathcal{R}$ and $\mathcal{C}$ are fixed to {all references} and {ENamedElement}, respectively. Therefore, a total of six configurations will be taken into account in the evaluation (Table 3). These six settings are chosen in order to study the

Table 3 The six configurations considered

	2	3	4
$\mathcal{A}_{\text{all}}$	All-2	All-3	All-4
$\mathcal{A}_{\text{names}}$	Names-2	Names-3	Names-4

Table 4 Precision evaluation results

	MRR	Differences in MRR
Text search	0.668	–
Names-2	0.734	$<$ .001 / +0.066
Names-3	0.742	$<$ .001 / +0.074
Names-4	0.757	$<$ .001 / +0.089
All-2	0.742	$<$ .001 / +0.089
All-3	0.752	$<$ .001 / +0.084
All-4	0.702	$<$ .001 / +0.034

effect of the path length (model structure considered) and the set $\mathcal{A}$ (information of the model considered) in the precision.

The evaluation procedure is as follows. Each query mutant has associated the original meta-model which it comes from. Therefore, for each query, we perform the search and retrieve the ranked list of results. We look up the original meta-model in the ranked list and annotate its position r. Then, the reciprocal rank $(1/r)$ is computed. Ideally, the best precision is achieved when the original meta-model is in the first position. Once all queries have been resolved, we summarize the set of reciprocal ranks using the average obtaining the mean reciprocal rank (MRR). To see if the differences between MAR and the baseline are statistically significant we use the t test. The results are shown in Table 4. The first column contains the MRR of each one of the configurations of MAR and the text search engine (baseline). The second one contains the differences in the mean and the p value when comparing with the baseline (for example, the cell $<$ .001 / +0.066 associated with Names-2 means that the difference in MRR when comparing with Text search is $0.734 - 0.668 = +0.066$ and it is statistically significant since the p value is $<$ .001).

When only names are considered (Names-x where $x = 2, 3, 4$), the greater the length the better the precision. This is caused by the fact that increasing the path length produces an increase in the amount of model structure considered. In particular, in Names-2 the majority of paths are of the form of $\left(\boxed{\text{n}}, \overrightarrow{\text{name}}, \boxed{\text{named element}}\right)$ (that is, the bags of paths consider information of the form *this model has a named element whose name is n*). Names-3 considers all information that Names-2 takes and adds interesting paths that encode facts of the type *this EClass of name n1 has this reference of name n2* (e.g. $\left(\boxed{\text{StateMachine}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{eStruct.Feat.}}, \boxed{\text{EReference}}, \overrightarrow{\text{name}}, \boxed{\text{states}}\right)$ in Fig. 8) or *this EClass of name n1 has this attribute of name n2* ($\left(\boxed{\text{State}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{eStruct.Feat.}}, \boxed{\text{EAttribute}}, \overrightarrow{\text{name}}, \boxed{\text{name}}\right)$ in Fig. 8). Finally, Names-4 adds paths of the type *this EClass of name n1 has a reference connected with an EClass of name n2* ($\left(\boxed{\text{StateMachine}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{eStruct.Feat.}}, \boxed{\text{EReference}}, \overrightarrow{\text{eType}}, \boxed{\text{EClass}}, \overrightarrow{\text{name}}, \boxed{\text{state}}\right)$ in Fig. 8).

On the other hand, when more attributes are considered (All-x where $x = 2, 3, 4$), increasing the path length does not always improve the precision, since the configuration with length 4 is the one that achieves the worst results. In this case, an increase in the path length provokes an increase in the number of noise and useless paths. In Fig. 8, an example of this type of noisy paths is $\left(\boxed{\text{StateMachine}}, \overrightarrow{\text{name}}, \boxed{\text{EClass}}, \overrightarrow{\text{ePackage}}, \boxed{\text{EPackage}}, \overrightarrow{\text{eClassifier}}, \boxed{\text{EClass}}, \overrightarrow{\text{abstract}}, \boxed{\text{false}}\right)$. This path encodes the fact that the user is interested in models with an EClass whose name is *StateMachine* and belongs to an EPackage that contains a non-abstract EClass. The information that this path encodes does not help in the search and produces noise which may confuse the search engine.

From this evaluation we can conclude that using some model structure in the search is beneficial in terms of precision. As we have shown MAR, outperforms the text search in all configurations (all differences are positive and are statistically significant with p value < 0.001). In order to choose a proper configuration we need to find trade-off between precision and efficiency since increasing the size of $\mathcal{A}$ and the maximum length causes an increase in the size of the index and, consequently, the time used to resolve a particular query. The effect on the response time is studied in the next section.

The main threat to validity in this experiment is that the derived mutants might not represent the queries that an actual user would create. We have manually checked that they are reasonably adequate, but an experiment with queries from real users is required to confirm the results. Regarding the selection of the configuration parameters, they are only valid for Ecore models. For other type of models, a similar analysis is required in order to select a proper configuration. Nevertheless, for models in which names play an important role (e.g. UML, BPMN) we expect configurations which rely on name attributes to perform similarly well.

7.3 Query response time

To evaluate the performance of MAR we have used all mutant queries used in the search precision evaluation. We want to measure the query response time using each one of the six configurations introduced in the previous section. Thus, for each configuration, we have indexed the 17690 Ecore models and for each query mutant, we perform the search and measure the response time. Moreover, to evaluate the effect of the query size we count the number of packages, classes, structural features and enumerations and we classify the queries in

Table 5 Statistics of the mutants set used in the evaluation

	EPackage				EClass				EAttribute				EReference				Total
	Min	Median	Mean	Max	Min	Median	Mean	Max	Min	Median	Mean	Max	Min	Median	Mean	Max	Queries
Small	1	1	1.2	8	3	5	5.3	11	0	2	2.4	11	2	4	3.8	11	337
Medium	1	1	1.6	18	3	13	13.4	35	0	8	9.8	45	2	10	10.8	29	848
Large	1	1	3.6	46	3	27	37.25	193	1	29	35.7	168	2	24	30.9	116	410

Table 6 Mean and max statistics of the query response times (in seconds)

	Small		Medium		Large	
	Mean	Max	Mean	Max	Mean	Max
All-2	0.13	0.25	0.24	0.51	0.35	1.82
All-3	0.29	1.02	0.66	2.07	0.97	3.63
All-4	0.98	2.75	2.36	6.60	3.47	9.18
Names-2	0.03	0.09	0.06	0.16	0.09	1.08
Names-3	0.05	0.21	0.11	0.41	0.19	1.64
Names-4	0.08	0.41	0.28	1.41	0.66	3.85

Table 7 Mean and max statistics of the number of paths

	Small		Medium		Large	
	Mean	Max	Mean	Max	Mean	Max
All-2	66.0	195	178.7	892	437.1	1578
All-3	238.2	916	789.1	4337	2001.0	8600
All-4	619.7	4997	2849.0	21282	10340.0	48803
Names-2	22.6	94	65.5	284	172.0	552
Names-3	91.6	544	366.1	2030	1059.4	5069
Names-4	284.5	3687	1679.0	13108	7724.0	37474

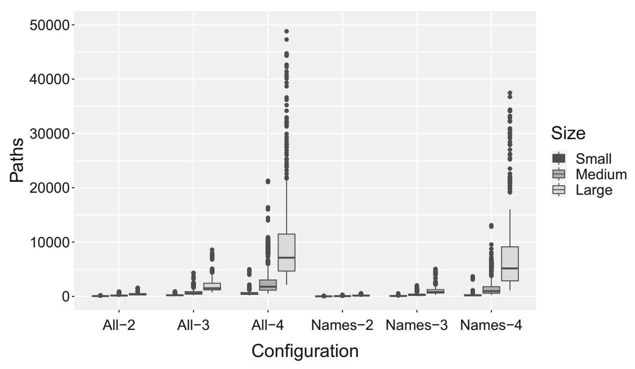


Fig. 16 Boxplot of the number of paths per query. The x-axis represents the configurations considered, the y-axis the number of paths of a query

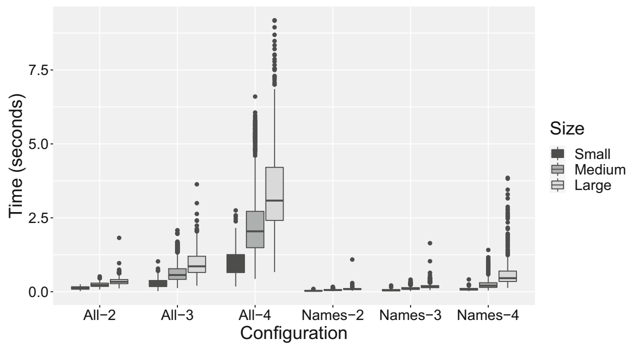


Fig. 17 Boxplot of the query response time. The x-axis represents the configurations considered, the y-axis the time in seconds and the colour the size of the queries

three types: small (less than 20 elements), medium (between 20 and 70) and large (more than 70). Table 5 shows some details about the contents of the queries.

Figure 17 shows for each configuration and for each query size the query response time as a set of boxplots. Table 6 shows the mean and max. statistics of the query response times per configuration and query size.

Increasing the path length causes an increase in the number of distinct paths. On the other hand, adding elements to the set $\mathcal{A}$ implies generating bigger multigraphs (since more nodes are considered). Consequently, increasing the set $\mathcal{A}$ also causes an increase in the number of distinct paths. This fact has an impact on the size of the index, the number of accesses to the inverted index and the path extraction process. In Fig. 16, the distribution of paths per query size and MAR configuration is shown and Table 7 contains the mean and max statistics of the number of paths of the queries per configuration and query type. It can be observed that a change in the MAR configuration of the form $conf\text{-}x \rightarrow conf\text{-}(x+1)$ (where *conf* is All or Names) and Names-x → All-x (with $x = 1, 2$ or 3) causes an increase in the number of paths for each one of the three sizes.

Therefore, taking account the results in Tables 4 and 6 a reasonable choice to consider part of the model structure, have a good precision and still have a fast response time is *Names-4*.

Finally, it is worth mentioning that category *small* corresponds to queries that users would normally perform. In this case, the average response time is less than a second in all configurations (Table 6). On the other hand, for medium and large mutants the average query response time is near 2 and 3 s, respectively, in the most demanding configuration (All-4), which can be considered a good result since there are queries with up to 193 classes. This suggests that MAR could be applicable in scenarios like model clone detection.

8 Related work

In this section, we review works related to our proposal. We organize these works in five categories: model encoding techniques, model repositories, mining and analysis of model repositories, search engines and model clone detection.

8.1 Model encoding techniques

The concept of BoP, which is a key element of our query-by-example approach, is inspired by the notion of *bag of path-context* defined in Code2vec [6]. In this paper, the authors study the transformation of code snippets into continuous distributed vectors by using a neural network with an attention mechanism whose input is a multiset of paths extracted from the AST of a code snippet.

SAMOS is a platform for model analytics. It is used in [8] for clustering and in [10] for meta-model clone detection. SAMOS's approach is very close to ours since both transform models into graphs and extract some paths (the authors call them n-grams). However, there are some differences between MAR and SAMOS. First, our graph is different as we consider single attributes as independent nodes, and therefore the extracted paths are different. And second, SAMOS is not designed to perform a search since it uses similarity metrics which are not thought to perform a fast search.

AURORA [45] extracts tokens from Ecore models and uses a neural network to classify them in domains (e.g. state machines, database, bibliography, etc.). These tokens are treated as words and the input of the neural network is a TF-IDF vector. In AURORA, paths are encoded by concatenating tokens. MEMOCNN [46] can be considered the evolution of AURORA. It uses an encoding schema similar to AURORA but MEMOCNN transforms meta-models into images and they are used to feed a convolutional neural network that performs the classification.

Clarisó and Cabot propose the use of graph kernels for clustering models [21]. The general idea is to transform models into graphs and perform machine learning methods by using graph kernels [47] as similarity measure between graph models.

8.2 Model repositories

The study performed by Di Rocco [26] shows that the search facilities provided by existing model repositories are typically keyword based, tag based or there is no search facility at all. Our aim with MAR is to offer a search platform able to provide a search layer on top of third-party repositories from which models can be extracted.

MDEForge is a collaborative modelling platform [13] intended to foster "modelling as a service" by providing facilities like model storage, search, clustering, workspaces, etc. It is based on a megamodel to keep track of the relationships between the stored modelling artefacts. One goal of MDEForge is to remove the need of installing desktop-based tooling and to enable collaboration between users. As such, the services offered by MDEForge are focused on its registered users and their resources. In contrast, MAR is intended to offer a simpler and more open search service.

Img2UML [35] is a repository of UML models obtained by collecting images of UML diagrams by searching for Google images. These images are converted to XMI files, and there is a search system based on querying names of specific properties (e.g. class names). The use of this repository in a teaching environment has been studied in [36], showing its usefulness for improving students' designs. MAR could be used a similar setting, using its example-based facilities to improve the search process.

GenMyModel is a cloud service which provides online editors to create different types of models and stores thousands of public models [3]. Our crawlers target GenMyModel, using the REST API it offers in order to provide search capabilities for its models.

ReMoDD is a repository of MDE artefacts of different nature, which is available through a web-based interface [29]. It is intended to host case studies, rather than individual files. In addition, it does not include specific search facilities.

Hawk provides an architecture to index models [11], typically for private projects and it can be queried with OCL. This means that the query result will only contain models that satisfy the OCL expression. In contrast, the main use case of MAR is inexact matching, which means that the resulting models do not necessarily match the complete query.

8.3 Mining and analysis of model repositories

There are some works devoted to crawl and analyse models. ModelMine [50] is a tool to mine models from GitHub, along with additional information such as versioning. It is similar to our approach for crawling GitHub, except that our system uses file sizes instead of dates for filtering results. Another difference, is that it does not provide specific capabilities for validating the retrieved models. Similarly, Restmule is a tool to help in the creation of REST APIs to mine data from public repositories [52]. We plan to use it and compare its performance and reliability against our system.

The work by Barriga et al. proposes a tool-chain to analyse repositories of Ecore models [12]. To prove its usefulness, they analyse a repository of 2420 ecore models. Their approach is similar to the analysis module of MAR. However,

Table 8 Summary of search engine for models approaches adapted and extended from [14]

	Query format	Search type	Models	Index	Crawler	Megamodel aware
[42] MOOGLE	Text	Text search	Any	Yes	No	No
[11] HAWK	OCL-like language	Exact results	Any	Yes	No	Partial
[15] Text version	Text search	Text search	WebML	Yes	No	No
[15] Structure-based version	Query-by-example	Structure based	WebML	No	No	No
[31]	Query-by-example	Structure based	UML	Yes	No	No
[38] MoScript	OCL-like language	Exact results	Any	No	No	Yes
[14]	Text search	Text search	Any	Yes	No	Yes
[28]	Query-by-example	Structure based	BPMN	No	No	No
[57]	Query-by-example	Structure based	BPMN	Yes	No	No
[18]	Query-by-example	Structure based	Petri nets	No	No	No
MAR	Query-by-example	Structure based	Any	Yes	Yes	No

our validators can handle models of different types (not only Ecore meta-models), they are fault-tolerant and we have used them to analyse an order of magnitude more models.

A statistical analysis is carried out in repositories of ~ 450 ecore models by Di Rocco et al. [25]. In particular, the authors studied relations between structural features of the meta-models (for instance, correlation between featureless meta-classes and number of meta-classes with a supertype) and their distributions (for instance, the distribution of isolated classes). On the other hand, a similar analysis is carried out by the same authors in [27] in the context of ATL transformations together with the source and target meta-models.

SAMOS is used to analyse and cluster the S.P.L.O.T. repository [9]. First, they try to get a high-level overview of the repository by considering large clusters. Then, they carry out a more fine-grained clustering to discover model clones or duplicates.

8.4 Search engines

A number of search engines for models have been proposed. However, as far as we know, there is no one widely used. In Table 8, the proposed search engines are shown along with an analysis of their features.

A search engine of WebML models is presented by Bislimovska in [15]. It supports two types of queries: text queries and query-by-example. This search engine uses an inverted index so the response is fast. On the other hand, when an example is given as a query, the similarity between models is computed by using a variation of the $A*$ algorithm. Therefore, the query's response is slow since an iteration of the full repository is needed to obtain the final ranked list. On the other hand, the work by Gomes [31] focuses on the retrieval of UML models using the combination of WordNet and Case-Based Reasoning.

MOOGLE [42] is a generic search engine that performs text search and the user can specify the type of the desired model element to be returned. MOOGLE uses the Apache Lucene query syntax and Apache SOLR as the backend search engine. In [14], a similar approach to MOOGLE is proposed with some new features like megamodel awareness (consideration of relations among different kinds of artefacts).

MoScript is proposed in [38], a DSL for querying and manipulating model repositories. This approach is model-independent, but the user has to type complex OCL-like queries that only retrieve exact models, not a ranking list of models.

The work by Dijkman [28] investigates the application of graph matching algorithms to ranking BPMN given an example as the query. After that, in [57], the *greedy* algorithm (studied in [28]) is improved obtaining a procedure of computing similarity between BPMN graphs ten times faster. On the other hand, Cao [18] proposes to use the Hungarian algorithm to query similar Petri nets.

In the work by Kalnina [34], a tool for finding model fragments in a DSL tool is proposed. The fragments are expressed using concrete syntax and a graph matching procedure is used as the search mechanism.

Genetic algorithms with structural and syntactic metrics were used to compare two meta-models in the work by Kessentini [37]. However, this approach is not intended to be fast since it uses genetic algorithms.

8.5 Model clone detection

It is worth including this topic in the related work section because it can be seen as a type of approximate matching. Model clone detection is a relatively new topic [10] and much research has been devoted to deal with challenges such as scalability (models can be large graphs [22]), tool-specific representations, internal identifiers, and abstract versus concrete syntaxes [10,54]. However, model clone detection approaches have not been applied to search in large-scale repositories, since these techniques are more focussed on pairwise model matching. In our case, we have a scoring algorithm that is global in the sense that model paths are stored together in the index. This makes our approach more scalable in terms of the size of the repository.

In [10], SAMOS is applied to meta-model clone detection. A study of the UML clones is carried out in [54] and [55] where a tool to detect clones (called MQ_{lone}) is presented and a description of clone types in UML is provided.

Exas encodes a graph model into a vector to perform approximate matching [44,48]. This technique has also been applied to detect model transformation clones [56].

On the other hand, clone detectors in Simulink models have been studied in the literature. For instance, an approach of clone detection in data flow languages is presented in [23] which is based on graph theory and it is applied on model clone detection in Simulink. SIMONE (an extension of text-based tool NICAD) is presented in [5] to detect clones in Simulink models.

9 Conclusions

In this paper, we have presented MAR, a search engine specifically designed for models. We have explained its crawler and analysis architecture, with which we have processed 600,000 models so far. We have implemented indexers for both keyword-based and example-based queries. For the latter, we report on the indexer design which includes both batch and incremental modes. We have evaluated the performance of MAR in terms of indexing and search time, obtaining good results. Moreover, we have evaluated the search precision using mutation operators to simulate user queries. At the user level, MAR provides a web interface to perform queries and inspect the results.

Regarding its practical usage, MAR is able to crawl models in several sources and it has currently indexed more than 500,000 unique models of different kinds, including Ecore meta-models, BPMN diagrams and UML models. This makes it a unique system in the MDE ecosystem, that we hope it is useful for the community. MAR is available at http://mar-search.org.

As future work, we plan to keep discovering new model sources and indexing more models. We want to include feedback mechanisms for users to rate the results of the queries and use this feedback to improve the system. Moreover, we would like to address some limitations on the expressiveness of the queries. For instance, it would be interesting to allow encoding incomplete models (e.g. a UML class whose name is a wildcard) and to consider synonyms (e.g. a search for *employee* also returns models related to *workers*). Another line of work is the application of machine learning techniques that may help us automate some tasks like generating descriptive tags, automatically classifying models in order to improve faceted search and to consider quality features in the ranking algorithm by using a learning to rank approach [17,32].

Acknowledgements This work was funded by a Ramón y Cajal 2017 Grant (RYC-2017-237), funded by MINECO (Spain) and co-funded by the European Social Fund. José Antonio Hernández López enjoys a FPU Grant funded by the Universidad de Murcia.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

References

1. Apache HBase. https://hbase.apache.org/
2. Apache Lucene. https://lucene.apache.org/
3. GenMyModel. https://www.genmymodel.com/
4. Massif: Matlab simulink integration framework for eclipse. https://github.com/viatra/massif
5. Alalfi, M.H., Cordy, J.R., Dean, T.R., Stephan, M., Stevenson, A.: Models are code too: Near-miss clone detection for simulink models. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 295–304. IEEE (2012)
6. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL), 1–29 (2019)
7. Arasu, A., Cho, J., Garcia-Molina, H., Paepcke, A., Raghavan, S.: Searching the web. ACM Trans. Internet Technol. **1**(1), 2–43 (2001)
8. Babur, Ö., Cleophas, L.: Using n-grams for the automated clustering of structural models. In: International Conference on Current Trends in Theory and Practice of Informatics, pp. 510–524. Springer (2017)
9. Babur, Ö., Cleophas, L., van den Brand, M.: Model analytics for feature models: case studies for splot repository. In: MODELS Workshops, pp. 787–792 (2018)
10. Babur, Ö., Cleophas, L., van den Brand, M.: Metamodel clone detection with samos. J. Comput. Lang. (2019)
11. Barmpis, K., Kolovos, D.: Hawk: towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, pp. 1–9 (2013)
12. Barriga, A., Di Ruscio, D., Iovino, L., Nguyen, P.T., Pierantonio, A.: An extensible tool-chain for analyzing datasets of metamodels. In: Proceedings of the 23rd ACM/IEEE International Conference

on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–8 (2020)
13. Basciani, F., Di Rocco, J., Di Ruscio, D., Di Salle, A., Iovino, L., Pierantonio, A.: Mdeforge: an extensible web-based modeling platform. In: CloudMDE@ MoDELS, pp. 66–75 (2014)
14. Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Exploring model repositories by means of megamodel-aware search operators. In: MODELS Workshops, pp. 793–798 (2018)
15. Bislimovska, B., Bozzon, A., Brambilla, M., Fraternali, P.: Textual and content-based search in repositories of web application models. ACM Trans. Web (TWEB) **8**(2), 1–47 (2014)
16. Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Softw. Syst. Model. 1–9 (2020)
17. Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G.: Learning to rank using gradient descent. In: Proceedings of the 22nd international conference on Machine learning, pp. 89–96 (2005)
18. Cao, B., Wang, J., Fan, J., Yin, J., Dong, T.: Querying similar process models based on the Hungarian algorithm. IEEE Trans. Serv. Comput. **10**(1), 121–135 (2016)
19. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 1–26 (2008)
20. Chowdhury, S.A., Varghese, L.S., Mohian, S., Johnson, T.T., Csallner, C.: A curated corpus of simulink models for model-based empirical studies. In: 2018 IEEE/ACM 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), pp. 45–48. IEEE (2018)
21. Clarisó, R., Cabot, J.: Applying graph kernels to model-driven engineering problems. In: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, pp. 1–5 (2018)
22. Deissenboeck, F., Hummel, B., Juergens, E., Pfaehler, M., Schaetz, B.: Model clone detection in practice. In: Proceedings of the 4th International Workshop on Software Clones, pp. 57–64 (2010)
23. Deissenboeck, F., Hummel, B., Jürgens, E., Schätz, B., Wagner, S., Girard, J.F., Teuchert, S.: Clone detection in automotive model-based development. In: 2008 ACM/IEEE 30th International Conference on Software Engineering, pp. 603–612. IEEE (2008)
24. Di Rocco, J., Di Ruscio, D., Härtel, J., Iovino, L., Lämmel, R., Pierantonio, A.: Understanding mde projects: megamodels to the rescue for architecture recovery. Softw. Syst. Model. **19**(2), 401–423 (2020)
25. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining metrics for understanding metamodel characteristics. In: Proceedings of the 6th International Workshop on Modeling in Software Engineering, pp. 55–60 (2014)
26. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Collaborative repositories in model-driven engineering [software technology]. IEEE Softw. **32**(3), 28–34 (2015)
27. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining correlations of atl model transformation and metamodel metrics. In: 2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering, pp. 54–59. IEEE (2015)
28. Dijkman, R., Dumas, M., García-Bañuelos, L.: Graph matching algorithms for business process model similarity search. In: International Conference on Business Process Management, pp. 48–63. Springer (2009)
29. France, R., Bieman, J., Cheng, B.H.: Repository for model driven development (remodd). In: International Conference on Model Driven Engineering Languages and Systems, pp. 311–317. Springer (2006)
30. George, L.: HBase: the definitive guide: random access to your planet-size data. O'Reilly Media, Inc. (2011)
31. Gomes, P., Pereira, F.C., Paiva, P., Seco, N., Carreiro, P., Ferreira, J.L., Bento, C.: Using wordnet for case-based retrieval of UML models. AI Commun. **17**(1), 13–23 (2004)
32. He, C., Wang, C., Zhong, Y.X., Li, R.F.: A survey on learning to rank. In: 2008 International Conference on Machine Learning and Cybernetics, vol. 3, pp. 1734–1739. IEEE (2008)
33. Holmes, R., Walker, R.J.: Systematizing pragmatic software reuse. ACM Trans. Softw. Eng. Methodol. (TOSEM) **21**(4), 1–44 (2013)
34. Kalnina, E., Sostaks, A.: Towards concrete syntax based find for graphical domain specific languages. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 236–242. IEEE (2019)
35. Karasneh, B., Chaudron, M.R.: Online Img2UML repository: an online repository for UML models. In: EESSMOD@ MoDELS, pp. 61–66. Citeseer (2013)
36. Karasneh, B., Jolak, R., Chaudron, M.R.: Using examples for teaching software design: an experiment using a repository of uml class diagrams. In: 2015 Asia-Pacific Software Engineering Conference (APSEC), pp. 261–268. IEEE (2015)
37. Kessentini, M., Ouni, A., Langer, P., Wimmer, M., Bechikh, S.: Search-based metamodel matching with structural and syntactic measures. J. Syst. Softw. **97**, 1–14 (2014)
38. Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., Cabot, J.: Moscript: A dsl for querying and manipulating model repositories. In: International Conference on Software Language Engineering, pp. 180–200. Springer (2011)
39. Kolovos, D., De La Vega, A., Cooper, J.: Efficient generation of graphical model views via lazy model-to-text transformation. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 12–23 (2020)
40. López, J.A.H., Cuadrado, J.S.: Mar: A structure-based search engine for models. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 57–67 (2020)
41. López-Fernández, J.J., Guerra, E., De Lara, J.: Assessing the quality of meta-models. In: MoDeVVa@ MoDELS, pp. 3–12. Citeseer (2014)
42. Lucrédio, D., Fortes, R.P., Whittle, J.: Moogle: A model search engine. In: International Conference on Model Driven Engineering Languages and Systems, pp. 296–310. Springer (2008)
43. Lucrédio, D., Fortes, R.P., Whittle, J.: MOOGLE: a metamodel-based model search engine. Softw. Syst. Model. **11**(2), 183–208 (2012)
44. Nguyen, H.A., Nguyen, T.T., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Accurate and efficient structural characteristic feature extraction for clone detection. In: International Conference on Fundamental Approaches to Software Engineering, pp. 440–455. Springer (2009)
45. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Automated classification of metamodel repositories: a machine learning approach. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 272–282. IEEE (2019)
46. Nguyen, P.T., Di Ruscio, D., Pierantonio, A., Di Rocco, J., Iovino, L.: Convolutional neural networks for enhanced classification mechanisms of metamodels. J. Syst. Softw. **172**, 110860 (2021)
47. Nikolentzos, G., Siglidis, G., Vazirgiannis, M.: Graph kernels: a survey. arXiv preprint arXiv:1904.12218 (2019)
48. Pham, N.H., Nguyen, H.A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N.: Complete and accurate clone detection in graph-based models. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 276–286. IEEE (2009)
49. Porter, M.F.: An algorithm for suffix stripping. Program (1980)
50. Reza, S.M., Badreddin, O., Rahad, K.: Modelmine: a tool to facilitate mining models from open source repositories. In: Proceedings

of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–5 (2020)

51. Robertson, S., Zaragoza, H.: The Probabilistic Relevance Framework: BM25 and Beyond. Now Publishers Inc, London (2009)
52. Sanchez, B.A., Barmpis, K., Neubauer, P., Paige, R.F., Kolovos, D.S.: Restmule: enabling resilient clients for remote apis. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 537–541 (2018)
53. Stephan, M.: Towards a cognizant virtual software modeling assistant using model clones. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), pp. 21–24. IEEE (2019)
54. Störrle, H.: Towards clone detection in UML domain models. Softw. Syst. Model. **12**(2), 307–329 (2013)
55. Störrle, H.: Effective and efficient model clone detection. In: Software, Services, and Systems, pp. 440–457. Springer (2015)
56. Strüber, D., Acreţoaie, V., Plöger, J.: Model clone detection for rule-based model transformation languages. Softw. Syst. Model. **18**(2), 995–1016 (2019)
57. Yan, Z., Dijkman, R., Grefen, P.: Fast business process similarity search. Distrib. Parallel Databases **30**(2), 105–144 (2012)
58. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664
59. Zhai, C., Massung, S.: Text data management and analysis: a practical introduction to information retrieval and text mining (2016)

José Antonio Hernández López is a predoctoral researcher at the University of Murcia. He obtained a B.Sc. in Mathematics, a B.Sc. in Computer Science and a M.Sc. in Big Data from the University of Murcia. Currently, he is enrolled in a Ph.D. programme in Computer Science supervised by Jesús Sánchez Cuadrado. His research interests include model-driven engineering (MDE), recommender systems and machine learning for software engineering.

Jesús Sánchez Cuadrado is a Ramón y Cajal researcher at the Languages and Systems Department of the University of Murcia. His research is focused on model-driven engineering (MDE) topics, notably model transformation languages, meta-modelling and do main-specific languages. Lately, he has been involved in the construction of the MAR search engine (http://mar-search.org). On these topics, he has published several articles in journals and peer-revie wed conferences, and developed several open-source tools. His e-mail address is jesusc@um.es and his web-page is http://sanchezcuadrado.es.