Towards Distributed Ecore Models

Jesús M. Perera Aracil and Diego Sevilla Ruiz
DITEC, University of Murcia, Campus Universitario de Espinardo, 30100, Espinardo, Murcia, Spain

Keywords: MDE, REST, Cloud, Ecore.

Abstract: Models are the cornerstone of Model-Driven Engineering (MDE). Their size is constantly growing, becoming one of the main problems when it comes to manipulating them, via model-to-model transformations, model-to-text transformations or simply parsing them. In this paper we propose a way of distributing Ecore models representing them as JSON and URLs as identifiers, since HTTP is one of the most successful distributed protocols ever created. An implementation of distributed Ecore models using a RESTful-like service is also presented and and is publicly available.

1 INTRODUCTION

Models are a digital representation of reality, and as such they are used as a software artifact and manipulated by transformations to generate other artifacts. Models are becoming more complicated and larger in size, making it difficult for a single computer to parse, transform or handle them.

Some new tools and mechanisms have been designed and implemented, such as paralellization of transformations (Tisi et al., 2013), which enables the execution of rules in parallel by distributing the load on different cores. The use of continuations (Cuadrado and Aracil, 2014) has also been implemented. This continuations enable the automatic scheduling of transformation rules by suspending transformation rules whenever a needed element is not yet transformed or available.

There has been advances in data storage facilities (Benelallam et al., 2014) (Espinazo-Pagán et al., 2015), using databases in a way that queries for models can be performed in a faster way. Even applying big data techniques on models (Scheidgen and Zubow, 2012) (Barmpis and Kolovos, 2014) have been studied, such as using MapReduce algorithms.

Even though these have alliviated the problem, it is certain that an implementation which supports natively a distribution framework must be designed. This implementation would allow transformations to scale transparently for both the user and programmer.

One of the main challenges comes from the representation of references between model elements. In XML Metadata Interchange (XMI)(OMG, 2015), the standar serialization format for Ecore models, these are implemented fragment-like depending on the position of the target element in the graph (e.g., `#//EClass` refers to metaclass `EClass` from Ecore). Cross-references between elements from different models are prepended by the path to the XMI file of the other model. Thus, this strong connection hinders the ability to split a model into smaller pieces for distribution or even changes in the location of files might break them.

This paper is structured as follows. Section 2 introduces the concepts of distributed models which are used in our proposal. Section 3 introduces and describes our approach of distributing Ecore models using URLs and JSON representation. Section 4 presents our implementation of distributed Ecore models. Section 5 discusses related work and Section 6 summarizes conclusions and future work.

2 BACKGROUND

Software industry, and more concretely, Model-Driven Engineering (MDE), is constantly growing. In the case of models, these are becoming more and more complicated as well as bigger in size, up to a point in which using a single computer for handling models is not viable.

It is clear that a step forward must be taken in order to cope with this increase to still be able to use MDE technologies as a valid option. Distributing Ecore models would allow to better parallelize accesses to a model, since querying different pieces of

Aracil, J. and Ruiz, D.
Towards Distributed Ecore Models.
DOI: 10.5220/0005685002090216
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 209-216
ISBN: 978-989-758-168-7
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

the model would be independent from others. In addition, it would enable collaboration on models, since different groups could work on different pieces of the same model.

We are now going to explain the concepts on which this paper and our proposal of distributed Ecore models heavily rely on.

2.1 REST

Representational State Transfer (REST) (Fielding, 2000) is an architecture for building web applications. It is based on the idea of offering collections of resources through, mostly, HTTP.

RESTful web applications use HTTP methods as a means of accessing, modifying or deleting resources uniquely identified by URLs. These methods operate in a different way if the URL to which it is applied is a resource or a collection.

- GET method: Retreives the resource or collection as a response.
- POST method: Creates a new element in a collection.
- PUT method: Replaces a collection or resource with other.
- DELETE method: Deletes the resource or collection.

RESTful applications rely heavily on hypermedia, in which resources are named and referenced by their respective URL. A resource is identified by its URL and can link (and be linked) to other resources by these URLs.

2.2 JSON

JavaScript Object Notation (JSON)(JSON, 2015) is a lightweight data-interchange format based on JavaScript. It is a human readable format which makes it easy for humans to understand and write and for computers to parse and serialize.

In recent years, with the blooming interest in web applications, JSON has seen a rise in popularity at the expense of the other big serialization format, XML. JSON has a small but rich data values:

- Strings: a sequence of characters.
- Number: integers, floats, scientific notation.
- Booleans.
- Null: no value.
- Array: a collection of values.
- Object: a collection of key-value pairs, in which the keys are Strings naming the value it binds to.

JSON, on the contrary of XML, does not have a mechanism to extend the language with more complex values or a description of its structure, such as XMLSchema (W3C, 2001). This is overcome by the fact that the language is simple and structured without ambiguity.

3 DISTRIBUTED ECORE MODELS

Ecore models are traditionally stored by means of XMI(OMG, 2015), which is a file-based storage. References (and cross-references between different models) are also file-based, which makes splitting a model difficult, since it would now need to handle more files and keep them synchronized.

A roadmap (Clasen et al., 2012) has been proposed for distributing models and performing transformations in the cloud, but still no implementation has been designed to fulfill this challenge.

We propose to store Ecore models on the cloud as JSON objects, which would allow for a distributed access using REST web applications. Model elements are transformed into a JSON representation of themselves by a process which will be defined later depending on their metaclass.

Unique URLs are generated for all model elements based on containment of instances. These URLs are created by the URL of its parent prepended to the name of the containment `EReference` which holds the elements and, if it is the case, its numeric position (i.e., if an element `B` is contained in a `EReference` named `ref` of `A`, with URL `http://www.example.com/A`, its URL would be `http://www.example.com/A/ref`). Root elements of models are given a “base URL” from which the rest of the URLs of that model are derived.

For illustration purposes, we are going to explain this approach with 3 models:

- Subset of Ecore meta-meta-model (shown in Figure 1)
- Bag meta-model (shown in Figure 2).
- Bag model (shown in Figure 3).

The base URL for Ecore `EPackage` will be `http://www.example.com/repo/0`, that for the Bag metamodel will be `http://www.other.com/repo/1` and finally, that for the Bag model will be `http://www.example.com/repo/2`.

We are now going to explain in detail the process of generating both JSON and URLs for some metaclasses of Ecore, `EClassifier`, `EPackage` and `EStructuralFeature`.

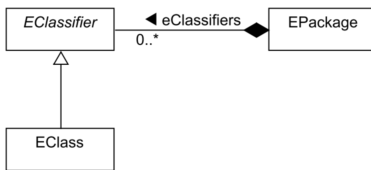

Figure 1: Ecore subset.

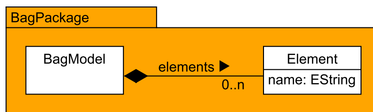

Figure 2: Bag metamodel.

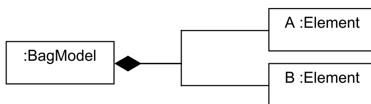

Figure 3: A simple Bag model.

3.1 EClassifiers

Instances of `EClassifiers` will be transformed by collecting all their `EStructuralFeatures` and creating a JSON object which binds the names of these `EStructuralFeatures` to their corresponding URLs. These new URLs will be the concatenation of the URL for the `EClassifier` being transformed concatenating the name of their `EStructuralFeatures`.

Code Example 1 shows a snippet of how both `EClassifier` and `EClass` are transformed into JSON. We assume that their relative positions in their containment reference are 0 and 1, respectively. It should be noted that an `EClassifier` is an `EClass` instance due to Ecore being its own metamodel. `EStructuralFeatures` conversion to JSON will be explained in Subsection 3.3.

Code example 2 shows how `EClasses` our Bag metamodel example are transformed into JSON. The base URL for the `Bag EPackage` is `http://www.other.com/repo/1`.

Code example 3 shows our Bag model transformed into JSON. The root element of this model is a `BagModel` instance, whose base URL is `http://www.example.com/repo/2`.

3.2 EPackages

`EPackages` are transformed as if they were `EClassifiers`: all their `EStructuralReferences` are given a name and an URL (based on the URL of the instance), then a JSON Object is created pairing these names to the URLs. Code example 4 shows how the `EPackage` for Ecore is transformed. The base URL for it is `http://www.example.com/repo/0`.

```
// EClassifier
// URL: http://www.example.com/repo/0/
//      eClassifiers/0
{
"name":"http://www.example.com/repo/0/
        eClassifiers/0/name",
"abstract":"http://www.example.com/repo/0/
            eClassifiers/0/abstract",
"eClass":"http://www.example.com/repo/0/
          eClassifiers/0/eClass",
...
}

// EClass
// URL: http://www.example.com/repo/0/
//      eClassifiers/1
{
"eClass":"http://www.example.com/repo/0/
          eClassifiers/1/eClass",
...
}
```

Code Example 1: Ecore subset as JSON.

```
// BagModel
// URL: http://www.other.com/repo/1/
//      eClassifiers/0
{
"name":"http://www.other.com/repo/1/
      eClassifiers/0/name",
"abstract":"http://www.other.com/repo/1/
          eClassifiers/0/abstract",
"eClass":"http://www.other.com/repo/1/
        eClassifiers/0/eClass",
...
}

// Element
// URL: http://www.other.com/repo/1/
//      eClassifiers/1
{
"name":"http://www.other.com/repo/1/
      eClassifiers/1/name",
"abstract":"http://www.other.com/repo/1/
          eClassifiers/1/abstract",
"eClass":"http://www.other.com/repo/1/
        eClassifiers/1/eClass",
...
}
```

Code Example 2: Bag metamodel.

3.3 EStructuralFeatures

The main challenge of distributing Ecore models is how to represent `EReferences` of model instances. In

```
// instance of BagModel
// URL: http://www.example.com/repo/2
{
"elements":"http://www.example.com/repo/2/
            elements",
"eClass":"http://www.example.com/repo/2/
          eClass",
...
}

// Element A (instance of Element)
// URL: http://www.example.com/repo/2/
//      elements/0
{
"name":"http://www.example.com/repo/2/
        elements/0/name",
"eClass":"http://www.example.com/repo/2/
          elements/0/eClass",
...
}

// Element B (instance of Element)
// URL: http://www.example.com/repo/2/
//      elements/1
{
"name":"http://www.example.com/repo/2/
        elements/1/name",
"eClass":"http://www.example.com/repo/2/
          elements/1/eClass",
...
}
```

Code Example 3: A Bag model.

```
// EPackage for Ecore
// URL: http://www.example.com/repo/0
{
"name":"http://www.example.com/repo/0/
        name",
"nsURI":"http://www.example.com/repo/0/
         nsUri",
"eClassifiers":"http://www.example.com/repo/0/
                eClassifiers",
"eClass":"http://www.example.com/repo/0/
          eClass",
...
}
```

Code Example 4: Ecore subset.

an XMI file, these are based on the target metaclass position inside the model graph. Cross-references between instances of different models are file-based, which hinders the ability to split a model into several parts without breaking these references.

Thus, a mechanism that could enable `EReferences` to be uniquely, globally identifiable and not be dependent on its physical position would allow for splitting and distributing a model easier.

We propose that `EStructuralFeature`s are given a full and valid URL constructed based on the URL of the metaclass it belongs and its name. For example, the `EAttribute` `nsURI` of the Ecore `EPackage` would be `http://www.example.com/repo/0/nsUri`. Then, if we `GET` that URL, we would receive the JSON value which represents it. This JSON value depends on the subclass it is (i.e., `EAttribute` or `EReference`) and on whether it is multi-valued or not.

Multi-valued features will be transformed into a JSON Array of values, while single-valued features will be the JSON representation of the feature. `EAttributes` will map directly into JSON values (i.e., a `EInt` will be a JSON Double, `EBoolean` will be a JSON Boolean...). On the other hand, `EReferences` will contain the URL of the target element.

Recalling our subset of Ecore, `EPackage` has a `eClassifiers` `EReference` of type `EClassifier`, which is multi-valued. As it was seen previosly on code example 1, the URL placeholder for this is `http://www.example.com/repo/0/eClassifiers`. If we query this URL, we would get a JSON Array of URLs, one for each `EClassifier` that the Ecore `EPackage` contains. On the other hand, the single-valued `name` `EAttribute` would point to a URL which would give its name. Code example 5 illustrates these features. First, the Ecore `EPackage` is shown to remember the URLs of its features. Then, the result of an HTTP GET request to its `name` URL is shown, followed by that of its `eClassifiers` feature.

Finally, `EStructuralFeatures` from metamodel instances are also generated in the same way, this can be seen in code example 6. In this example, the `Elements` instances contained by the `BagModel` instance can be queried through the URL `http://www.example.com/repo/2/elements`. It is worth noting that in this case, a `BagModel` is not an `EPackage` but it is the root of our example model, so it is given a base URL.

4 PROPOSED IMPLEMENTATION

Our implementation for distributed Ecore models is based on a RESTful-like web application and JSON, but no restrictions apply to the storage database or facility to be used. Thus, any implementation conforming to the generation of JSON for metaclasses or instances and creating URLs for the `EStructuralFeatures` of all model elements could

```
// EPackage for Ecore
// URL: http://www.example.com/repo/0
{
"name":"http://www.example.com/repo/0/
        name",
"eClassifiers":"http://www.example.com/repo/0/
                eClassifiers",
...
}

---

// EPackage name EAttribute
// URL: http://www.example.com/repo/0/
        name
"EPackage"

---

// EPackage eClassifiers EReference
// URL: http://www.example.com/repo/0/
        eClassifiers
["http://www.example.com/repo/0/
         eClassifiers/0",
 "http://www.example.com/repo/0/
         eClassifiers/1",
 ...
]
```

Code Example 5: An example of EStructuralFeatures.

```
// BagModel instance
// URL: http://www.example.com/repo/2
{
"elements":"http://www.example.com/repo/2/
            elements",
"eClass":"http://www.example.com/repo/2/
          eClass",
...
}

---

// BagModel elements EReference
// URL: http://www.example.com/repo/2/
//      elements
["http://www.example.com/repo/2/
         elements/0",
 "http://www.example.com/repo/2/
         elements/1"]
```

Code Example 6: An example of EStructuralFeatures.

be designed and would be interoperable.

We have already implemented a complete Ecore meta-meta-model following this proposal, which is available at http://www.cloudecore.com:8080/repo/0. Our implementation is written in Scala (Typesafe, 2015d) using the Cake Pattern (Hunt, 2013), Play! Framework (Typesafe, 2015c) and Akka (Typesafe, 2015b) library. As our data storage facility, we have used MongoDB (MongoDB, 2015). A simplified overview of the design can be seen in Figure 4. This simple architecture enables an easy development of new plugin-like drivers for other databases or ways of storaging and querying elements. This way, any new plugin for a database or storage facility (even using the filesystem) would be trivial to add, allowing for a faster growing list of supported ways of storaging models.

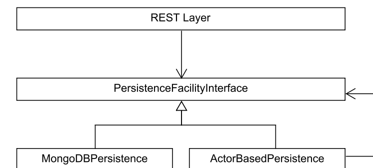

Figure 4: Implementation overview.

Akka allows to deploy on remote machines and sending messages in a transparent way, while also managing failures on actors, which can be automatically restarted. This framework also handles replication of actors as well as being able to deploy actors remotely. These features enable our implementation to be itself distributed in case it is needed and be flexible and elastic transparently. These actors implement a message passing mechanism to execute code. Messages received are first enqueued and then processed in order to guarantee that no race condition exist.

Model elements are stored and queried by their id, which is the URL on which they are located, overriding the `_id` field of MongoDB. This NoSQL database takes as value a JSON document, so we define a `_value` field to store the model element in its JSON format directly.

We have a simple API to access the storage facilities. Using the Cake pattern, we can simulate dependency injection (Fowler, 2004) and configure the concrete storage facility to be used. Code example 7 shows the methods from the API that need to be implemented for a new storage facility to be recognized. This is a simple design allows implementations to be developed rapidly, allowing for increasing the number of storage facilities that can be used; the Scala `Option` monad is used for null-safety.

```
class PersistentFacilityInterface{
  save(id: String, value: JSONValue):
      Option[JSONValue]
  load(id: String): Option[JSONValue]
}
```

Code Example 7: Methods to be implemented for a new database.

The REST layer provides HTTP verbs for actions such as creating new model elements (POST) or obtaining them (GET). A persistence layer has been designed to abstract the details of the actual database or filesystem used to store the models.

Figure 5 illustrates the process of retreiving an element. A client would make an HTTP GET request to the URL of the model element needed. The REST application would then, based on this URL, ask the storage facility to retreive its JSON representation; finally, this JSON would be sent to the client.

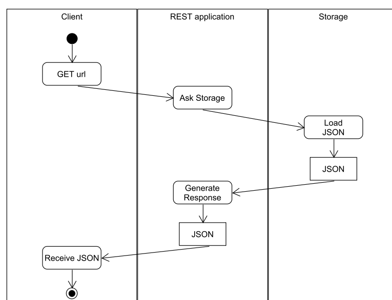

Figure 5: Activity diagram for GETing an element.

Figure 6 shows the process of uploading an element. As it can be seen, the client would generate the corresponding JSON of the element to be uploaded, together with the JSON corresponding to its `EStructuralFeatures`, and upload them to the server by an HTTP POST request on their URLs. The server would then obtain the data and save it in the storage linking the JSON to its URL. This process requires the base URL to have been asked for beforehand, which would require an additional HTTP GET request to obtain it to a special URL: `http://www.cloudecore.com:8080/nextModelURL`; any GET request will obtain a unique one.

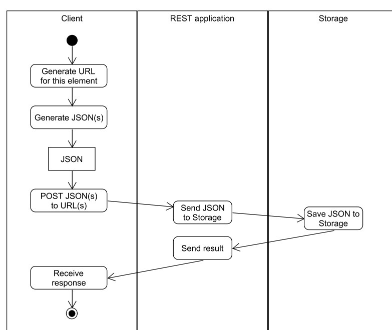

Figure 6: Activity diagram for POSTing an element.

These processes described are executed concurrently but no race condition is present, thanks to the message passing mechanism offered by actors. This means that to requests of `nextModelURL` will be enqueued and delivered in a first-in first-out order. This also implies that two successive HTTP POST requests will result in the last one overwriting the first one.

Finally, `EPackages` need an additional POST request to link their URI to its URL (i.e., Ecore in our implementation is located at `http://www.cloudecore.com:8080/repo/0`, so we link its URI `http://www.eclipse.org/emf/2002/Ecore` to its URL). This is done by a POST request to `http://www.cloudecore.com:8080/metamodels` with both the URI and URL as a JSON object as shown in code example 8.

```
{
 "uri":"http://www.eclipse.org/emf/2002/
  Ecore",
 "url":"http://www.cloudecore.com:8080/
  repo/0"
}
```

Code Example 8: JSON to be posted to add a new EPackage.

4.1 Benchmarks

We have tested our implementation by taking the Ecore meta-meta-model and uploading it as a distributed Ecore model, following the proposal described in this paper. We conduct our benchmarks as indicated in (IBM, 2008a) and (IBM, 2008b).

The following computer setups have been used:

Our client PC setup is as follows:

- Intel i7 3770K 3.90GHz.
- 16 GB RAM.
- 10 Mbps downstream internet connection.
- 600 Kbps upstream internet connection.
- Windows 10 Pro 64 bit.
- Eclipse 4.4.2 Luna.
- JDK 1.8.0_45 (with parameters -Xms512m -Xmx2048m).

Our server PC setup is as follows:

- Intel Atom N2800 1.86GHz.
- 2 GB RAM.
- 100 Mbps symmetric internet connection.
- ArchLinux Kernel 3.14.32 64 bit.
- JDK 1.8.0_51 (with parameters -Xmx1024m).
- Play! Framework 2.

We have performed tests in three different configurations:

- Client-Server in localhost.
- Client-Server in same network.
- Client-Server in different networks over the internet.

Table 1 shows the results of these benchmarks in milliseconds. From the data we can observe that even though our implementation performs well on localhost, the overhead of connecting over the internet is really high, as the average ping from the client to the server is 72 milliseconds, meanwhile the average ping in the same network is 3 milliseconds. These delays are the clear explanation of the decrease in performance, since all the requests are now being slowed down by the net. The theorical best average of elements per second for 72 milliseconds is 13.8 (i.e., 1000 milliseconds / 72 milliseconds, which are the most connections that can be achieved per second with that network lag). On the other hand, a ping of 3 milliseconds implies that 333 elements per second is the theorical maximum that can be achieved.

Table 1: Average elements per second for each benchmark.

	POST	**GET**
Localhost	5500	9000
Same Network	250	252
Internet	11	12

As previously mentioned, these benchmarks clearly indicate that the most important factor in performance for distributed Ecore models is network latency. It would also imply that it can handle multiple different connections, since each petition is not overloading the server. Any improvement on network connectivity would imply a better performance with the same hardware configuration.

5 RELATED WORK

The interest in being able to cope with huge models is an important and difficult challenge which has not been yet solved. Many propositions and tools have been developed to help on this task.

Emfjson (Hillairet, 2011) replaces XMI as the serialization standard for models with JSON. It can store models on .json files or use document databases such as MongoDB or CouchDB as storage. It does not split a model into different pieces or elements, saving it as a single JSON document. References are implemented using URIs as in the XMI format, and fragment-like (e.g., `http://www.eclipse.org/emf/2002/Ecore#//EClass` refers to Ecore EClass)

EMF-REST (Ed-Douibi et al., 2013) generates a REST interface to access and modify Ecore models. This generation must be done per metamodel, so two different metamodels generate two different applications. Interestingly, URLs are used to name elements in the model, but these elements need to have an `id` or `name` attribute or any attribute with `unique` activated.

A roadmap (Clasen et al., 2012) has been proposed for the transformation of Very Large Models (VLM), in which they discuss the importance of distributing models and strategies for partitioning them. They discuss some benefits of bringing Ecore models to the cloud such as being able to support VLM, offer better scalability than file-based models and enabling collaboration for teams by sharing models easily.

Finally, the Mondo Project (Kolovos et al., 2015) aims to tackle the increasingly important challenge of scalability in MDE in a comprehensive manner.

6 CONCLUSION AND FUTURE WORK

In this paper we have presented a proposal and implementation for distributing Ecore models based on JSON and REST applications. We have shown that using URLs as references between models grants the ability to distribute and partition with ease any model.

We have presented a implementation of our proposed representation, in which he have successfully uploaded the complete Ecore meta-meta-model, demonstrating that it can manage non-trivial models and metamodels.

We plan to add a configurable pagination support by default to `EStructuralFeatures` which are `EList`s, so that the the client can ask for a number of elements at a time.

We also want to study the use of Akka clusters(Typesafe, 2015a) so that we can transparently handle redundancy and fail-over recuperations, as well as being able to make a distributed implementation using this technology.

We are currently studying adding a batch-mode for adding elements, so network overhead can be reduced by reducing the necessary HTTP connections. Finally, we plan to pretty-print JSON answers son they can be more user-friendly and human readables if the web client is not capable of doing it. We would also like to study the use of using Big Data tools as storage facility, such as Hadoop.

Using automatic HTTP redirect rules might prove useful when dealing with complicated structures and relations between model elements. For instance, it would be easier for the user that

a redirect is performed whenever they access `http://www.cloudecore.com:8080/repo/0/eClass/name` (i.e., the name of the `EClass` for the Ecore `EPackage`). This would redirect to `http://www.cloudecore.com:8080/repo/0/eClassifiers/12/name` instead of giving a `Not Found` error for the original URL. This would imply that a processing be done before querying the databases with the given URL to look for cases in which a redirect must be performed.

A Java implementation for EMF Resource would be helpful, so it would allow to transparently access distributed Ecore models in a familiar way. This would allow other tools to natively be able to use distributed Ecore models.

A versioning system would be interesting to be included as a transparent feature in the implementation, so that it could allow different versions of the same model or metamodel to coexist with newer ones, as well as allow comparing changes made from one version to other.

Finally, we are also developing a model-to-model transformation language that is aware of distributed Ecore models and will exploit the inherent parallelism that they offer. A preliminary metamodel which implements this transformation language can be found at http://cloudecore.com:8080/repo/1.

The code for our distributed Ecore model implementation will be available soon, as well as the code for the transformation model whenever it is completed. Also, any new feature added will be also published and its code made available.

REFERENCES

Barmpis, K. and Kolovos, D. (2014). Towards scalable querying of large-scale models. In Cabot, J. and Rubin, J., editors, *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 35–50. Springer International Publishing.

Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., and Launay, D. (2014). Neo4EMF, a Scalable Persistence Layer for EMF Models. In Cabot, J. and Rubin, J., editors, *ECMFA- European conference on Modeling Foundations and applications*, volume 8569, pages 230–241, York, UK, United Kingdom. University of York, Springer International Publishing.

Clasen, C., Didonet Del Fabro, M., and Tisi, M. (2012). Transforming Very Large Models in the Cloud: a Research Roadmap. In *First International Workshop on Model-Driven Engineering on and for the Cloud*, Copenhagen, Denmark. Springer.

Cuadrado, J. S. and Aracil, J. P. (2014). Scheduling model-to-model transformations with continuations. *Softw., Pract. Exper.*, 44(11):1351–1378.

Ed-Douibi, H., Alvarez, C., Cánovas, J., and Cabot, J. (2013). Emf-rest. http://som-research.uoc.edu/tools/emf-rest/.

Espinazo-Pagán, J., Cuadrado, J. S., and Molina, J. G. (2015). A repository for scalable model management. *Software and System Modeling*, 14(1):219–239.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern.

Hillairet, G. (2011). emfjson. http://emfjson.org/.

Hunt, J. (2013). Cake pattern. pages 115–119.

IBM (2008a). Robust java benchmarking, part 1: Issues. http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html.

IBM (2008b). Robust java benchmarking, part 2: Statistics and solutions. https://www.ibm.com/developerworks/java/library/j-benchmark2/.

JSON (2015). Json. http://json.org/.

Kolovos, D. S., Rose, L. M., Paige, R. F., Guerra, E., Cuadrado, J. S., de Lara, J., Ráth, I., Varró, D., Sunyé, G., and Tisi, M. (2015). MONDO: scalable modelling and model management on the cloud. In *Proceedings of the Projects Showcase, part of the Software Technologies: Applications and Foundations 2015 federation of conferences (STAF 2015), L'Aquila, Italy, July 22, 2015.*, pages 44–53.

MongoDB (2015). Mongodb website. https://www.mongodb.org/.

OMG (2015). XML metadata interchange (xmi). http://www.omg.org/spec/XMI/.

Scheidgen, M. and Zubow, A. (2012). Map/reduce on emf models. In *Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud Computing*, MDHPCL '12, pages 7:1–7:5, New York, NY, USA. ACM.

Tisi, M., Martinez, S., and Choura, H. (2013). Parallel Execution of ATL Transformation Rules. In *MoDELS*, pages 656–672, Miami, United States.

Typesafe (2015a). Akka clusters. http://doc.akka.io/docs/akka/2.3.12/common/cluster.html/.

Typesafe (2015b). Akka toolkit. https://www.akka.io/.

Typesafe (2015c). Play! framework. https://www.playframework.com/.

Typesafe (2015d). Scala language. http://www.scala-lang.org/.

W3C (2001). Xmlschema. http://www.w3.org/XML/Schema.