World Wide Modeling Made Easy - A Simple, Lightweight Model Server

Olivier Le Goaer, Eric Cariou, Franck Barbier

World Wide Modeling Made Easy
A Simple, Lightweight Model Server

Olivier Le Goaer, Eric Cariou and Franck Barbier
Computer Science Laboratory, University of Pau, Pau, France

Keywords: Model-driven Engineering, Client-server, Knowledge Sharing, Reuse, API, JavaScript.

Abstract: Sharing Models across organizations is a good idea but the lack of a tailored and lightweight tool hinders its adoption. In this paper, we explain how to turn any computer into a Model server, which is a server specialized in Models' location and retrieval. Such a server relies exclusively on specific URIs and commands thereof. The result, called "WWM", is an out-of-the-box module built upon Node.js. WWM targets the EMF ecosystem and takes the form of a JavaScript API for both server-side and client-side programming.

1 INTRODUCTION

"World Wide Modeling" (WWM) is a quite recent idea that Models have to be distributed and shared in as vast and immediate a way was the Web (Desfray, 2015). Indeed, the situation where everyone produces Models individually (in a manual or automated way, as well) has certainly been a brake on the adoption of the Model-Driven Engineering (MDE), yet recognized as providing powerful techniques. In particular, Models are no longer throw-away artefacts and reusing them off-the-shelf, assuming the fact they have been well designed and tested by experts of a domain (often enough to be promoted as « reference models »), is a key factor of success to reach fast development of software applications in that domain. Besides, what we may observe is that these sets of reference models are more and more frequently used in reproducible experiments, to compare different approaches or tools. So, only from this condition, that of large-scale Model sharing (and knowledge thereof), MDE is able to keep his promises in classroom, research, and industrial practice.

There exist some modeling portal initiatives and central repositories ((Ulrich et al., 2007), (France et al., 2006), (Zaytsev, 2015), (Basciani et al., 2014)) but they clearly failed to fulfil this role. This hard fact promotes the emergence of another class of hosting service: a Model server. Such a server let's Models to be reused among teams of a given company and, ultimately, crosses the frontiers of the enterprise. In both cases, a Model server aims at freely storing and publishing a predefined set of models, while working in a completely autonomous and decentralized way.

When looking at the overall 3-layer modeling stack promoted by OMG (see Figure 1), it becomes clear that a huge number of Models can be potentially shared. In addition, it must be taken into account the crucial conformance relationship (a.k.a metaness (Kühne, 2006)) that holds between levels when sharing models. Indeed, the basic but cogent principle behind the modeling stack is that a given layer M_n has been instantiated from the M_{n+1} layer and hence, conforms to the latter. From a reuse perspective, it means that retrieving a Model without the possibility of knowing and/or retrieving the language in which it has been written, is totally unsound.

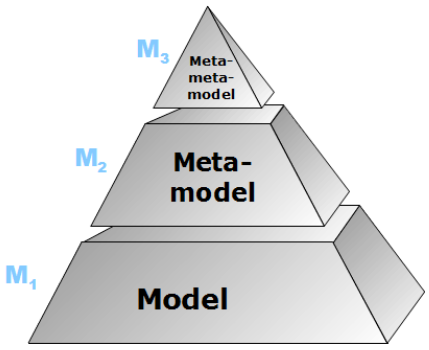

Figure 1: Standard modeling stack according to the OMG.

Le Goaer O., Cariou E. and Barbier F.
World Wide Modeling Made Easy - A Simple, Lightweight Model Server.
DOI: 10.5220/0006110802690276
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 269-276
ISBN: 978-989-758-210-3

So, the key idea at this point is that any MOF-compliant modeling activities are eligible for a world-wide dissemination. But paradoxically, even the technical assets of well-known OMG' standards like UML, SysML or BPMN are still a headache to obtain in a convenient and steady way. This is even truer for domain-specific languages (DSL), which have a more restricted audience.

Meanwhile, Eclipse Modeling Framework (EMF) is a very popular modeling workbench that implements the 3-layer modeling stack, and in which (Essential) MOF is embodied by the Ecore meta-language. The prevalent serialization mechanism of EMF is XMI. As a direct consequence, the solution proposed in this article is designed for, but is not limited to, the EMF ecosystem, provided that there is a XML-based storage under the hood. This means that remote Models are purposely retrieved so that they can be integrated into EMF projects or as inputs of EMF-based tools.

To illustrate the aforesaid hurdles, most of EMF tools are based on Models locally registered through platform-independent identifiers ("nsURIs" in the jargon) that are actually neither linked to anything tangible nor commonly shared. One of our ambitions is that these URIs become effective, delivered by clearly identified Model servers, starting with those of the OMG itself.

Besides, it is worthwhile recalling that a Model server is useless without a Model client thereof. That is why these two programs, which are two sides of the same corner, have all together been merged into a single package dubbed "wwm", and built using Node.js (https://nodejs.org). In doing so, we want to encourage developers to build various applications on top of this JavaScript API.

Notice that in this paper the term "Model" (notice the uppercase first letter) is used in its broadest sense, referring indifferently to M3, M2 or M1, whereas "model" strictly refers to an instance of a metamodel. Once this assumption made, the remainder of this paper is the following: Section 2 gives rationales for building a Model server. Section 3 describes the features of such a server while Section 4 elaborates on some aspects of its implementation. Section 5 can be viewed as a user manual. Section 6 provides a rigorous evaluation of performances of the server when reached by the default client. Conclusions and some perspectives are given in Section 7.

2 MOTIVATION

The original motivation of this proposal comes from another work on executable modeling, experienced on the Android platform in (Le Goaer et al., 2016), where a UML statechart could be "run" directly on a mobile device through *PauWare Engine* (www.pauware.com).

2.1 On the Need for a Model Server

Among the benefits claimed in the aforementioned publication, the issue of mobile applications updating was tackled thanks to an architecture in which models are delivered on-demand by a dedicated server: a so-called "Model server". This lets envisioning a panel of interesting capabilities like on-the-fly replacement of a model deployed on a device (sketched on Figure 2). This also enables Models exchange procedures among a set of connected objects provided that these objects are acting as Model servers in a peer-to-peer mode.

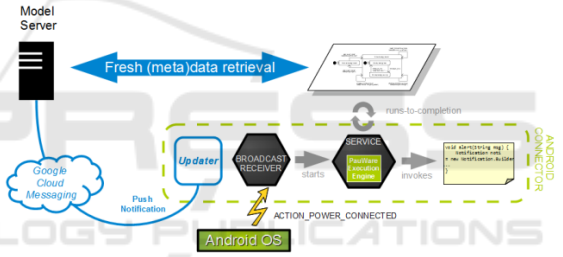

Figure 2: Model-based updatable Android architecture coming from (Le Goaer et al., 2016).

As far as we know, there is no really operational model server available. At best, model repositories technologies are part of an integrated modeling tool suite, and hence cannot be exported as a standalone feature. Moreover, such vendor-dependant solutions are cumbersome and are rarely free.

2.2 A Domain-Specific Server

The first question that may spring to mind is: Why not merely hosting the models on a regular server (e.g. HTTP, FTP, Version Control System …)? One may observe the same skepticism about domain-specific languages whilst general-purpose languages can do everything. Yet, they are legitimate since they are special for a narrow area of interest. The same applies for servers.

Firstly, it must be pinpointed that a Web server (saying, Apache HTTP server) is a heavy, general-

purpose container that can deliver evenly any kind of resources like pages, images, videos…. Only the MIME type differs and brings a few semantics when comes the time to handle resources. Yet, Models are specific abstractions on their own. Also, the gateway program layer provided by any modern Web Server is useless for inert resources, as are the cases of Models. Secondly, resources are arbitrarily organized so that URLs cannot exhibit any recurring pattern and hence no automated processing in a given scope or field. Thirdly, a Model is an example of linked data: to be useful, it has often to come along with other Models (ascending/descending conformance or siblings). Again, Models cannot be view just as raw, meaningless files and this is where regular servers fall short.

For all these reasons, it appears highly desirable to build a flyweight server able to run on top of devices with limited capacities. It has to be MDE-compliant and to offer just enough features: "the right tool for the job", in short.

3 PROPOSED SOLUTION

From a client point of view, any Model becomes identified by an URI that has roughly the following pattern:

```
model://host[:port]/M3/M2/M1#fragment
```

Any other URI format will be considered ill-formed by the server.

3.1 Scheme, Host and Port

The scheme part of the URI is model (without the trailing colon). The host is the IP of the computer running the model server or a name (resolved as an IP). Port sets the entry-point on which the server is listening clients' requests. Default port for a model server is 6464.

In the rest of the paper, for the sake of clarity, we exemplifies with a local Model server. All works exactly the same for a production-ready server of course.

3.2 Path and Segments

The path is hierarchical, as the strict reflect of the conformance relationship between the modeling levels M3, M2 and M1. Hence, it is decomposed as three depth levels of segments: the first segment exclusively refers to a metametamodel, the second exclusively refers to a metamodel and the last one exclusively refers to a model.

Example of metametamodel:

```
model://localhost:6464/ECORE/
```

Examples of metamodels:

```
model://localhost:6464/ECORE/UML/
model://localhost:6464/ECORE/ATL/
```

Examples of models:

```
model://localhost:6464/ECORE/UML/thermostat
model://localhost:6464/ECORE/ATL/class2table
```

Playing with this segmented URI, the end-user can seamlessly navigate through the OMG's modeling stack and has the insurance to get a Model serialized in the XMI format. At each level within the URI, the real file extension is omitted because it is inferred from the segment's name preceding it (all works case-insensitively). As an example, `/UML/foo` means that we want to get the model file named `foo.uml` (or `foo.xmi` if not found) stored on server-side. Because the M3 level is self-defined, we get directly `Ecore.ecore` from the root segment `/ECORE/`.

3.3 Fragments

Fragments aim at corresponding to a given piece of the entire Model, provided that pieces have been properly identified and labelled beforehand.

As a first striking example, it could be wise to consider the sequence diagram language definition as a fragment of the entire – thick – UML specification superstructure, as follows:

```
model://localhost:6464/ECORE/UML#SequenceDiagram
```

As a second example, we may consider the compound state "Operate" as a fragment of the complete behavior of a programmable thermostat defined with the UML statechart formalism (example given on the Franck Barbier's website):

```
model://localhost:6464/ECORE/UML/thermostat#Operate
```

3.4 Commands

Additionally to the aforesaid elements, two query commands are available: `?info` and `?list`.

The `?list` command allows us to see all the available Models at a given segment level. Naturally, this command does not work for the last segment. Below its usage to know all the models hosted on the server that are written in UML:

```
model://localhost:6464/ECORE/UML?list
```

The `?info` command returns information about a Model in a format that is simple to parse and easy

to read (See Section 4.3). Namely, who produced it? When? How? And so on. It works at any segment level. Here are examples:

```
model://localhost:6464/ECORE/UML?info
model://localhost:6464/ECORE/UML/
thermostat?info
```

4 IMPLEMENTATION

Writing a server from scratch is a tedious task. Instead, we choose Node.js, a JavaScript library that had become increasingly popular over the last few years, and fast for server-side programming. In addition, Node.js provides a powerful package manager that eases its distribution and installation. This Section shows important technical choices that are behind the scene of a Model server.

4.1 Communication Protocol

The Model server was built upon the TCP/IP layer. It is basically a running service that accepts connections through a given socket. It supports a request-response communication protocol style, which relies on a custom JSON-based encapsulation of data. The server closes the connection once a response is sent to the client in order to improve scalability. Last advantage: a simple DNS lookup mechanism is already available. Henceforth, the model protocol holds at the very same level than the http protocol and challenges the latter.

4.2 Server File System

A specific arrangement of directories and files on server-side stands behind the proposed URI mechanism and looks like that:

```
wwm
   |-- metametamodel
    |-- Ecore.ecore
    |-- Ecore.nfo
   |-- metamodel
    |-- UML.ecore
    |-- UML.nfo
    |-- ATL.ecore
   |-- model
    |-- thermostat.uml
    |-- thermostat.nfo
    |-- myshop.uml
    |-- class2table.atl
```

The root directory is `wwm`. It contains three mandatory directories: `metametamodel`, `metamodel` and `model`. Subdirectories are not allowed. The owner of the server ought to place her/his Model files into the suitable directories.

4.3 .nfo Descriptor

It is an ASCII text file which is optional but it is required in order for the `?info` command to work. This simple solution is an answer to the lack of metadata about a Model in general.

An .nfo file is a bulk, free, string format as a collection of text lines. Lines can contain a section name (starting with a # character) or a property. All the properties are in the form of `field:value` terminated by `\r\n`. Here is an example:

```
# General
creation: 2008/10/03
title: definition of a programmable
thermostat with the statechart
formalism

# Producer
author: John Doe
tool: Poseidon for UML
contact: john.doe@example.org

# Miscellaneous
metrics: 19 states
licence: Creative Commons
```

4.4 Fragment Markups

Unfortunately, there is no native solution to divide a Model into sub-model regions. This technical limitation remains even true at the XMI level. So, as a last resort, we can succeed to implement this feature with a low-level trick. Indeed, the idea is to leverage from standards XML comments markups `<!-- -->`. They have the advantage of being non-intrusive in the original file, but should not interfere with actual comments. They are much more annotations, and as such, must meet some conventions to be processed by our tool.

A Model fragment is then an enclosed chunk of XMI, and is labelled with a unique identifier. The simplest way to do that is to use a pair of markups, as follows:

```
...
<!-- wwm-begin(foo) -->
<eClassifiersxsi:type="ecore:EClass"
name="PackageableElement"
abstract="true"
eSuperTypes="#//NamedElement
#//ParameterableElement">
<eAnnotations
source="http://www.eclipse.org/emf/2002
/GenModel">
```

```
...
</eAnnotations>
</eClassifiers>
<!-- wwm-end(foo) -->
...
<!-- wwm-begin(bar) -->
...
<!-- wwm-end(bar) -->
...
```

In the current state, this lowbrow mechanism works. However, defining fragments that do not break existing dependencies (i.e. XMI attributes references) so that they remain consistent, is a non-trivial task, and even sometimes impossible. We think that more advanced techniques like Model slicing (Blouin et al., 2011) or theory of fragmentation (Amálio et al., 2015) should solve this issue.

4.5 JSON Listing

The preferred way to provide a listing, as a result of the `?list` command, is a JSON format. Indeed, this lightweight format can be natively handled with the JavaScript language and is easy to understand. Reconsidering a previous example, listing all the UML models should returns:

```
{
 "path":"model://localhost:6464/ECORE/U
 ML/"
  "models":["thermostat", "myshop"]
  "count":2
}
```

The following structure has been thought to ease a recursive usage because any concatenation of both path and models values rebuilds valid URIs. These can in turn be requested (See 5.2.2 for a sample), in the spirit of the HATEOAS principles.

5 GETTING STARTED

The wwm package is currently hosted on the official package registry (www.npmjs.com) and weighs only 11KB once minified. The command line to install our node.js package is the following:

```
$> npm install wwm
```

5.1 Server Setup

The specific directories are created when the module is first launched. This structure has to be perennial for the Model server run in a correct way. To that purpose, an integrity checking subroutine is performed every time the server boots.

As attested by the code below, launching the server is a breeze. Optionally, a callback allows listening which URIs are requested (line 3) by a client. Argument passed through the callback is a custom object modeling the URI and providing a set of useful methods to know its pattern.

```
1.  var wwm = require('wwm');
2.  var server = wwm.createServer('loc
    alhost', 6464);
3.  server.on('request',
    function (uri) {
4.    if (uri.isFragment()) {
5.      console.log('Someone asked for
     a fragment at' + Date.now());
6.      }
7.  });
```

5.2 Client Setup

The asynchronous nature of the server built with Node.js implies that the end-user defines her/his own callback functions to freely process the various responses of the Model server. The snippets given in the following are minimal for the sake of clarity. However, of course, much more sophisticated code can be written, the native language being JavaScript.

5.2.1 Callbacks

As explained in the previous section, what is received from the Model server depends on the URI pattern used. Consequently, there exist five specific event-based callbacks:

- on model: triggered once a plain Model is received. Argument passed through the callback is a custom object (name & content fields).
- on fragment: triggered once a model fragment is received. Argument passed through the callback is a custom object (name & content fields).
- on info: triggered once an info descriptor is received. Argument passed through the callback is an ASCII text.
- on list: triggered once a listing result is received. Argument passed through the callback is a JSON object (see fields in 4.5).
- on error: triggered once something went

wrong on server-side. Argument passed through the callback is a simple string describing the problem.

5.2.2 Samples

We first illustrate an exhaustive assignment of callbacks in order to handle all cases (ranging from line 3 to 14). Notice that method chaining is a convenient way to do that.

```
1.  var wwm = require('wwm');
2.  var client = wwm.createClient();
3.  client.on('model', function (m) {
        wwm.util.save(m);
4.      console.log(m.name + 'downloaded');
5.  }).on('info', function (i) {
6.      console.log(i);
7.  }).on('error', function (e) {
8.      console.error(e);
9.  }).on('list', function (l) {
10.     console.log (l.count + ' found');
11. }).on('fragment', function (f) {
12.     wwm.util.save(f);
13. });
14.
15. client.connect('model://localhost:6464/ECORE/UML'); //ask for a meta model
16.
17. client.connect('model://localhost:6464/ECORE/UML/myshop'); //ask for a model
```

Once the chosen callbacks functions are assigned, the client is ready for requesting the server with `connect()`. Here, the “UML” metamodel then “myshop” business model are requested, that will both trigger the same callback. Thus, the elected code (lines 4-5) calls `save()`, a helper bundled with wwm that creates a physical file with the proper XMI content on the client-side, and finally displays a message to the console.

As a second illustration, we give a programming idiom that solves nicely a frequent client-side intent, which consists in sending a list command having the final purpose to retrieve all the available Models.

```
1.  client.on('model', function(m) {
        wwm.util.save(m);
2.  }).on('list', function(l) {
3.      for (m in l.models) {
4.          //loop of requests
5.          this.connect(l.path + m);
6.      }
7.  });
8.
9.  //ask for listing
10. client.connect('model://localhost:6464/ECORE/UML?list');
```

The idea is to send a `?list` command (line 11) while having planned within the corresponding callback (lines 3-7) an iteration about the results so that Models are requested in the wake with `connect()`. When received, one after the other, the other callback (line 1-2) does the job thanks to, again, the `save()` helper.

6 STRESS TEST

Whilst there exists http load testing and benchmarking utilities (Siege, ab, Gatling ...), benchmarking a custom model protocol is quite new. We decided to write our very-own command line tool that spits its results out in csv format: wwm-bench. The example below requests a server for a Model 1000 times with a max of 50 concurrent clients:

```
wwm-bench -n 1000 -c 50 <ModelURI>
```

Again, the Model server is locally installed, thereby ignoring the unpredictable networking aspects during the test.

6.1 Preparation

Response times for the `info?` and `list?` commands are not primal concerns. Our measures have been therefore conducted on both Model and Model fragment access throughout the default Model client. For both tests, we experimented with a fixed pool of 5000 requests, ensuring an acceptable accuracy for average response time.

To that purpose, we started with a homemade DSL previously used in (Pierre et al., 2014) and then we automatically generated a set of dummy models containing an exponential number of model elements. In each one, fragment markups have been

inserted in such a way they represent 1/3 of the entire content. We used an unrefined generator, where meta-elements are arbitrarily instantiated until a given limit is reached. For a more controlled generation of instances, note that some tools exist, like (Ferdjoukh et al., 2015).

6.2 Results

We conducted experiments within 8 hours, using 2 vCores CPU 2.4 GHz with 8GB RAM. Figure 3 and Figure 4 show the results obtained for Model and Fragment respectively. Notice that we used a logarithmic scale (log-10) for the horizontal axis.

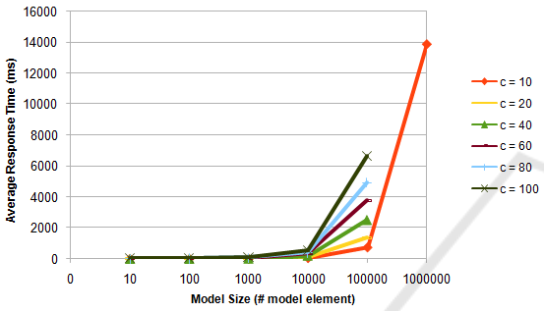

Figure 3: Model benchmark.

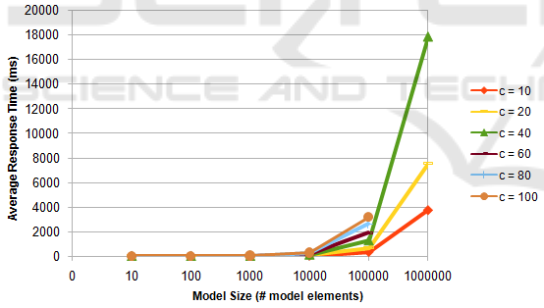

Figure 4: Fragment benchmark.

6.3 Observations

We can see that the average response time for models is consistently higher than for fragments, varying from one to triple, as the number of model elements increases. This result is logical since Models have a high transfer cost due to the verbosity of XMI. Fragments require also time-consuming operations for their real-time extraction, but this is counterbalanced by a lower transfer cost. Moreover, the performance loss is limited because we used string buffer operations rather than using xml parsing.

As we increase the number of concurrent requests, the average response time decreases. For instance, the response time for the smallest model size was on average 4.50 ms with 10 concurrent requests, and 31.67 ms with 100 concurrent requests. The average response time has a linear correlation to the number of concurrent requests, keeping the requests that can be served per second pretty constant. This good result is due to the single-threaded concurrency model of Node.js, relying on event-driven, non-blocking I/O. That is already the case with Web servers, where Node.js applications are noticeably faster than their equivalents (Lei et al., 2014).

We observed failures (timeout errors) for 10^6 model elements when concurrency exceeded 10 in the case of a model request, and 40 in the case of a fragment request. These threshold values represent a significant stress level for a non-optimized server, thereby demonstrating it perfectly fits to a normal use in MDE.

7 CONCLUSIONS

Model-based engineering never became a mainstream industrial practice partly due to a poor reuse level. In some respects, the tremendous success of Web is truly inspiring when looking at how models are nowadays unsatisfactory shared by engineers working in the MDE field.

In this paper, we clearly advocated in favor of a pure Model server so that any computer is amenable to host models reachable in read-only mode from all over the world. Every Model is turned into a URI outright, which exhibits a logical organization mapped with a physical organization on server-side. Its specific pattern enables simple but powerful features and, above all, ensures a global consistency for reuse. Owing much to the Node.js architecture, benchmarks showed it is enough scalable to face realistic usages.

We are expecting that its tiny size and its quick install procedure should ease its adoption, at least within the MDE community. From a practical point of view, while the server part of our solution can run effortlessly on any computer once the Node.js interpreter is installed, the default client currently written in JavaScript is not really intended to mobile OS, albeit this is technically possible on Android. So, the straightforward way to carry out the Android architecture introduced in Figure 2 is to write a whole new Android Java client. Nevertheless, Microsoft for example is hoping to power the IoT revolution and has announced new native support for

bringing Node.js to its Windows Phone OS. It's a safe bet that other platforms will do the same in a near future. Beyond mobile computing concerns, a Model client directly integrated into EMF as a plugin is probably a good idea for language engineers.

To broaden the discussion, assuming that the idea of putting open source modeling material at the disposal of the community thanks to Model server is well established, another challenge arises: a global index is missing. Indeed, the highly decentralized and autonomous nature of our solution requires a discovery mechanism, undoubtedly under the form of a Model search engine, like the “Moogle” of (Lucrédio et al., 2010). But crawling, indexing and complex querying on such Model servers are open research perspectives.

REFERENCES

Philippe Desfray, 2015. World wide modeling: The agility of the web applied to model repositories. *In Model-Driven Engineering and Software Development*, volume 506 of Communications in Computer and Information Science, pages 3-11. Springer International Publishing.

Frank Ulrich, Strecker Stefan, and Stefan Koch, 2007. Open Model, *ein Vorschlag für ein Forschungsprogramm der Wirtschaftsinformatik Wirtschaftinformatik*. Paper 69.

Olivier Le Goaer, Franck Barbier and Eric Cariou, 2016. Android Executable Modeling: Beyond Android Programming. *Modern Software Engineering Methodologies for Mobile and Cloud Environments*. IGI Global. Pages 269-283.

Robert France, Jim Bieman, and Betty H. C. Cheng, 2006. Repository for Model Driven Development (ReMoDD). *In Proceedings of the 2006 international conference on Models in software engineering*, pages 311-317. Springer Berlin Heidelberg, Berlin, Heidelberg.

Vadim Zaytsev. Grammar zoo: A corpus of experimental grammarware, 2015. *Science of Computer Programming, 98, Part 1*. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

Thomas Kühne. Matters of (meta-) modeling, 2006. *Software & Systems Modeling, 5(4)*: pages 369-385.

Daniel Lucrédio, Renata P. M. Fortes, and Jon Whittle, 2010. Moogle: a metamodel-based model search engine. *Software & Systems Modeling, 11(2)*: pages 183-208.

Arnaud Blouin, Benoit Combemale, Benoit Baudry, and Olivier Beaudoux, 2011. Modeling Model Slicers. *In ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, volume 6981, pages 62-76, Springer Berlin/Heidelberg.

Adel Ferdjoukh, Anne-Elisabeth Baert, Eric Bourreau, Annie Chateau, Rémi Coletta, and Clémentine Nebut, 2015. Instantiation of Meta-models Constrained with OCL: a CSP Approach. *In MODELSWARD, International Conference on Model-Driven Engineering and Software Development*, pages 213-222.

Nuno Amálio, Juan de Lara, and Esther Guerra, 2015. Fragmenta: A theory of fragmentation for MDE. *In ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, pages 106-115. IEEE.

K. Lei, Y. Ma, and Z. Tan, 2014. Performance comparison and evaluation of web development technologies in php, python, and node.js. *In IEEE 17th International Conference on Computational Science and Engineering* (CSE), pages 661-668.

Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, Alfonso Pierantonio, 2014. MDEForge: an Extensible Web-Based Modeling Platform. *CloudMDE@MoDELS*, pages 66-75.

Samson Pierre, Eric Cariou, Olivier Le Goaer, and Franck Barbier, 2014. A Family-based Framework for i-DSML Adaptation, *in European Conference on Modelling Foundations and Applications (ECMFA 2014)*, volume 8569 of LNCS, Springer, pages 164-179.

Franck Barbier’s thermostat example. http://web.univ-pau.fr/~barbier/PauWare/Programmable_thermostat/Programmable_thermostat.png.