Software Technologies: Applications and Foundations

STAF2014

July 24, 2014

York, United Kingdom

# BigMDE 2014

# 2nd Workshop on Scalable Model Driven Engineering

(http://www.big-mde.eu)

Dimitris Kolovos, Davide Di Ruscio, Nicholas Matragkas, Juan De Lara, Istvan Rath, Massimo Tisi (Eds.)

# Preface

As Model Driven Engineering (MDE) is increasingly applied to larger and more complex systems, the current generation of modelling and model management technologies are being pushed to their limits in terms of capacity and efficiency. As such, additional research and development is imperative in order to enable MDE to remain relevant with industrial practice and to continue delivering its widely-recognised productivity, quality, and maintainability benefits.

The second edition of the BigMDE workshop (http://www.big-mde.eu/) has been co-located with the Software Technologies: Applications and Foundations (STAF 2014) conference. BigMDE 2014 provided a forum for developers and users of modelling and model management languages and tools where to discuss different problems and solutions related to scalability aspects of MDE, including

- Working with large models
- Collaborative modelling (version control, collaborative editing)
- Transformation and validation of large models
- Model fragmentation and modularity mechanisms
- Efficient model persistence and retrieval
- Models and model transformations on the cloud
- Visualization techniques for large models

Many people contributed to the success of BigMDE 2014. We would like to truly acknowledge the work of all Program Committee members, and reviewers for the timely delivery of reviews and constructive discussions given the very tight review schedule. Finally, we would like to thank the authors, without them the workshop simply would not exist.

July 24, 2014
York (UK)

Davide Di Ruscio
Juan De Lara
Dimitris Kolovos
Nicholas Matragkas
Istvan Rath
Massimo Tisi

## Organizers

| | |
|---|---|
| Dimitris Kolovos | University of York (UK) |
| Davide Di Ruscio | University of L'Aquila (Italy) |
| Nicholas Matragkas | University of York (UK) |
| Juan De Lara | Universidad Autonoma de Madrid (Spain) |
| Istvan Rath | Budapest University of Technology and Economics (Hungary) |
| Massimo Tisi | Ecole des Mines de Nantes (France) |

## Program Committee

| | |
|---|---|
| Marko Boger | University of Konstanz (Germany) |
| Marco Brambilla | Politecnico di Milano (Italy) |
| Tony Clark | University of Middlesex (UK) |
| Juan De Lara | Universidad Autonoma de Madrid (Spain) |
| Marcos Didonet Del Fabro | Universitade Federal du Parana (Brazil) |
| Davide Di Ruscio | University of L'Aquila (Italy) |
| Jesús García Molina | Universidad de Murcia (Spain) |
| Esther Guerra | Universidad Autonoma de Madrid (Spain) |
| Dimitris Kolovos | University of York (UK) |
| Tihamer Levendovszky | Vanderbilt University (USA) |
| Nicholas Matragkas | University of York (Uk) |
| Alfonso Pierantonio | University of L'Aquila (Italy) |
| Istvan Rath | Budapest University of Technology and Economics (Hungary) |
| Markus Scheidgen | Humboldt-Universitat zu Berlin (Germany) |
| Seyyed Shah | University of York (Uk) |
| Gerson Sunyé | University of Nantes (France) |
| Jesús Sánchez Cuadrado | Universidad Autonoma de Madrid (Spain) |
| Massimo Tisi | INRIA and Ecole des Mines de Nantes (France) |
| Salvador Trujillo | IKERLAN (Spain) |
| Daniel Varro | Budapest University of Technology and Economics (Hungary) |
| Ed Willink | Willink Transformations (UK) |

# Table of Contents

# Hypersonic: Model Analysis and Checking in the Cloud

Vlad Acretoaie and Harald Störrle
Department of Applied Mathematics and Computer Science
Technical University of Denmark
rvac@dtu.dk, hsto@dtu.dk

## ABSTRACT

*Context:* Modeling tools are traditionally delivered as monolithic desktop applications, optionally extended by plug-ins or special purpose central servers. This delivery model suffers from several drawbacks, ranging from poor scalability to difficult maintenance and the proliferation of "shelfware".
*Objective:* In this paper we investigate the conceptual and technical feasibility of a new software architecture for modeling tools, where certain advanced features are factored out of the client and moved towards the Cloud. With this approach we plan to address the above mentioned drawbacks of existing modeling tools.
*Method:* We base our approach on RESTful Web services. Using features implemented in the existing Model Analysis and Checking (MACH) tool, we create a RESTful Web service API offering model analysis facilities. We refer to it as the Hypersonic API. We provide a proof of concept implementation for the Hypersonic API using model clone detection as our example case. We also implement a sample Web application as a client for these Web services.
*Results:* Our initial experiments with Hypersonic demonstrate the viability of our approach. By applying standards such as REST and JSON in combination with Prolog as an implementation language, we are able to transform MACH from a command line tool into the first Web-based model clone detection service with remarkably little effort.

## Keywords

Hypersonic, MACH, Models in the Cloud, clone detection, Web services, Prolog

## 1. INTRODUCTION

Until very recently, the only practically viable architecture for modeling tools was the traditional rich client architecture for desktop applications, sometimes complemented by specialized central servers, e. g., to provide model versioning and group collaboration capabilities. Such modeling tools are large, monolithical applications, even though some do offer scripting facilities, application programming interfaces (APIs), or a plug-in mechanism to allow for a certain degree of extensibility. A typical example is NoMagic's MagicDraw [21], which can be extended through an API, and complemented by the Teamwork Server plug-in [20] for centralized version control. With this type of modeling tools, the main revenue source for tool vendors is the sale of perpetual licenses for their product, possibly supplemented by ongoing technical support fees. Both the rich client architecture and the associated business model suffer from a series of disadvantages, some of a generic nature and some more specific to modeling tools.

In many other areas of computing, however, other, more flexible architectures are commonplace today. In particular, recent technological developments have brought about the widespread adoption of Cloud-based software architectures. Typically, such architectures involve the deployment of computationally intensive tasks to a centralized and fully transparent shared pool of configurable computing resources (i. e., "the Cloud") [18]. In this context, Web applications are nowadays a widely used method of delivering software functionality of many different kinds, including lightweight editors for parts of UML (e. g., GenMyModel [1], yUML [25]). Though already attracting users, most such Web-based offerings are currently not able to match traditional desktop tools in terms of features. Nevertheless, modeling in the Cloud does have the potential to address some important problems, such as achieving scalability in relation to increasingly large models and model repositories [16].

To realize this potential, we propose a solution where the features required in a fully-fledged modeling environment are hosted remotely and accessed in a transparent way. Playing the role of basic building blocks for a modeling workflow, these features should be accessible independently of each other and across a variety of devices. Fig. 1 visualizes the contrast between the widely used rich client architecture and the solution we propose. The crucial part of this proposal is the identification of features of a modeling environment that should be executed locally, and of those that should be executed on remote servers.

Arguably one of the most suitable application areas for Cloud-based approaches in modeling is model analysis. We select the requirements of this area as a background for constructing Hypersonic, a test vehicle for demonstrating our proposed approach for the delivery of modeling services - in this case, model analysis services. To implement Hypersonic, we use the features offered by our existing model analysis tool, MACH [27]. In its current form, MACH is a desktop application with a textual user interface. As most desktop applications, it requires installation prior to its usage, as well as explicit user actions/approval for installing updates. With Hypersonic, the features implemented in MACH be-

BigMDE'14 July 24, 2014. York, UK.

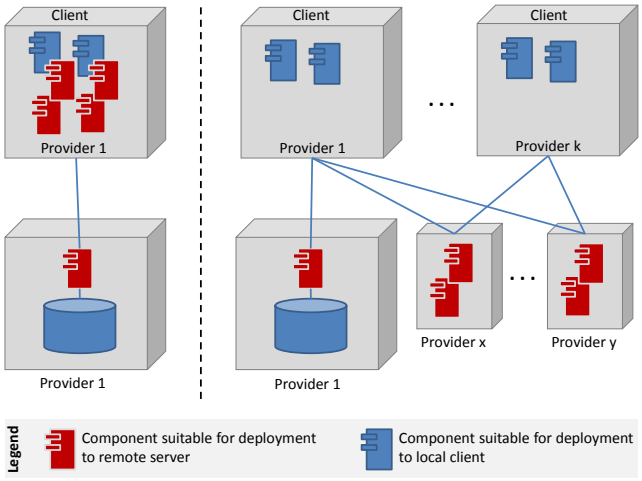



**Figure 1: Possible software architectures of modeling tools: most common solution today (left); modeling in the Cloud with Hypersonic (right)**

come RESTful Web services. They can be accessed remotely from a wide range of clients, without requiring installation or explicit user actions for installing updates. To demonstrate the usefulness of our Web service API we also create a sample client in the form of a responsive Web application.

In this paper we discuss the application scenarios and business cases for Cloud-based modeling services, derive requirements and constraints for the associated tools (Chapter 2), propose a distributed architecture to satisfy these requirements (Chapter 3), and report on a proof-of-concept implementation (Chapter 4). We also provide an overview of related work (Chapter 5) and summarize the conclusions of our study (Chapter 6).

# 2. DEFINING HYPERSONIC

## 2.1 Analysing Requirements by Stakeholder

Moving modeling tools to the Cloud is not primarily motivated by technological reasons, but by new application scenarios and business cases. In this section we analyze these application scenarios and business cases to highlight the advantages of our proposed architecture. We start by describing the status-quo, considering the stakeholders "tool provider", "modeler", "IT administrator", and "MBSD community", where MBSD stands for Model Based Software Development. Observe that all of these stakeholders exist in similar ways both in academic and commercial settings. We argue that the current state of the MBSD tool landscape is unsatisfactory for the stakeholders in several ways.

To a modeler, tools come as fixed packages: it is usually not possible (or not practical) to use one aspect, the editor, say, of one tool, and another aspect of a second tool. For instance, if the modeler appreciates the editing facilities of tool A, but that tool does not provide (adequate) code generation facilities, code generation may be difficult. It may not be economically viable due to the cost of purchasing two tools, or it may actually be impossible to use the editor of tool A combined with the code generator of another tool, unless both tools strictly adhere to model interchange standards. Also, from a modeler's perspective, resource intensive tasks may take an unreasonably long time to complete when executed locally.

To an IT administrator, repeatedly deploying tools to a large number of computers implies additional effort. Even if this effort is not incurred by the actual deployment (that might be expected to be taken care of by the modelers themselves), it becomes inevitable when distributing updates and fixes, help-desk services, and possibly ensuring that the tool's license usage is compliant with the agreement entered in with the tool vendor.

To the tool provider, delivering a modeling solution as a self-contained product with all the business logic on the client side makes it difficult to support the wide array of emerging computing devices. Migrating a large modeling tool to a tablet, smartphone, or Web interface would likely require extensive re-implementation, not to mention that the hardware requirements of many modeling tools are still prohibitive for mobile devices. Furthermore, separating the highly critical (and sometimes highly innovative) and non-critical functionalities of the application is cumbersome, yet still achievable through plug-ins. The distribution of counterfeit copies of the tool is also difficult to mitigate.

To the MBSD community, all of these factors are limiting, in much the same way as superficial differences between pre-UML modeling languages created niche markets for many different tools that were more expensive and less powerful than the UML tools that emerged after the unification of mainstream modeling languages in the late 1990's.

In a nutshell, the existing situation is suboptimal. Thus, we propose a different architecture: compute-intensive features and features with a high degree of innovation should be deployed to and executed in the Cloud rather than on the client machine. The client and server in this architecture are coupled loosely, by Web services, so that providing a new feature amounts to providing a new Web service. Such features can include advanced model analysis tools, code and report generation, and model transformation. Conversely, features of smaller distinctive value and small change rates, as well as features that require a higher degree of interactivity, can continue to be implemented as part of the client application. This will likely be the case for pure editing features, which many commercial vendors already offer as free community editions of their tools.

We have hinted at a set of criteria for determining which features of a modeling tool should be executed locally, and which would benefit from being migrated to the Cloud. Table 1 summarizes these criteria and presents examples.

## 2.2 Benefits of Modeling in the Cloud

Several advantages can be achieved by breaking up today's monolithic modeling tools into one stable, remote part that provides little added value distinguishing products (e. g., the editor), and a second, centralized part that comprises more advanced features with a higher change rate and higher distinctive value.

First, there are the advantages associated with thin-client systems in general: maintaining one central server instead of many remote clients reduces the work effort for IT administrators and ensures that modelers always have access to the latest version of the modeling tool. It also becomes easier to monitor license usage, which benefits administrators and tool providers alike.

Second, there are advantages rooted in the specific properties of Cloud-based solutions. This includes the availability

| | Local Features | Cloud-based Features |
|---|---|---|
| Property | high interactivity & incidence<br>high degree of stability<br>exchangeable features | high resource consumption<br>high degree of innovation<br>unique features |
| Examples | editing<br>simple syntax checking<br>automatic layout<br>context specific help<br>difference visualization<br>querying and navigation | reporting<br>global consistency checking<br>advanced model analysis<br>check in/out, locking<br>difference computation<br>code generation |

**Table 1: Criteria for assigning features to a local client or remote Web service, together with examples**

of considerable computational resources to each individual user at reduced overall cost, as well as high scalability of the available resources with an increasing number of users.

Third, a scenario in which some modeling facilities are provided as services is conductive to model interchange standards conformance. In such a scenario, service providers and client tool providers must both conform to model interchange standards such as the XML Metadata Interchange (XMI, [22]) in order to meet the requirements of modelers.

Fourth, there are advantages brought about by changes in the business model and enabling market mechanisms. Today, the modeling tool market is dominated by companies providing feature-complete bundles. A new competitor can only enter the market by providing a feature-complete solution, which all but excludes small companies and academic tool providers. Innovative analysis methods, specialized code generators, and similar tools can only be provided as plug-ins to one of the existing platforms, sometimes depending on the approval of the platform owner. In contrast, with a service-based architecture, tool providers can enter the market at lower cost, since they only have to provide their contribution *per se*, not a feature complete-tool. They can also address a larger part of the market, as they provide a generic Web service rather than a platform specific plug-in that applies to only one modeling tool. Modelers, on the other hand, can mix and match services as they like within the limits of standardized exchange formats.

It is likely that the described change in dynamics of the modeling tools market will inspire an increase in competitiveness between existing tool providers, while at the same time providing an incentive for new providers to enter the market. Both these developments will likely lead to a higher degree of innovation. This could translate into new concepts from academia dissipating to industrial modeling at a faster pace. Users will thus have both a financial and a technical incentive to experiment with new features.

From a financial perspective, tool providers will be able to bill features separately as subscription Web services. This could reduce unit prices for customers, who only buy the features they need, and may subscribe to services as they need them. Spending on expensive "shelfware" can be reduced or avoided altogether. For the supplier, this opens the perspective of a new business model in which a steady stream of revenue is generated through subscription services, while features of high distinctive value are much better protected against counterfeiting.

It must be mentioned, though, that adopting a service-based approach to modeling entails some trade-offs. Most are caused by the distributed nature of the approach. For instance, the process of uploading large models to a Cloud-based service may constitute a performance bottleneck. Security and privacy are also new aspects that come into play, considering that a centralized warehouse will store models owned by different organizations. These organisations must be able to trust that not only will their models not be accessible to other users, but they will also be protected against unauthorised mining by the modeling service provider. And, perhaps most importantly, the usefulness of the solution is dependent on a working Internet connection. Nevertheless, these drawbacks are common to the majority of Software as a Service (SaaS) solutions, and have not undermined this architecture style's acceptance.

## 2.3 A Test Case for Hypersonic

In Section 2.1 we have discussed which types of features lend themselves to deployment as Web services. We now select one feature, clone detection, as a test case for exploring the proposed architecture (i. e., the Hypersonic API). Our selection is motivated by the following considerations.

- The feature is well researched, published, and implemented (see [26] and [27], respectively), and has been used by a large number of students in several courses in which it has demonstrated its usefulness. So, the feature is readily available and arguably valuable.
- Clone detection demands significant resources, as it is based on semantic and structural model matching. For large models, the latency implied by uploading a model to the Cloud may be offset by savings in run-time achieved by using a powerful machine in the Cloud.
- Detecting clones is an activity carried out as part of the model quality assurance process. It is normally not executed concurrently to other modeling activities. Therefore, in some scenarios, a clone detection feature is not required or even useful. For example, models reverse-engineered from code do not require clone detection if the code is known to be clone-free.
- It is a unique feature: no UML modeling tool currently offers a clone detection feature. This includes research prototypes other than our own MACH tool. It is therefore safe to assert that this feature provides a high degree of innovation.

# 3. ARCHITECTURE

A first step towards the realization of the ideas presented in Section 2 is the definition of a common Web service interface accepted by all stakeholders. The details of such an interface must be the result of a wide reaching discussion, which is beyond the scope of this paper. Instead, we take an exploratory approach and design a RESTful Web service API for the purpose of Cloud-based model analysis. We refer to this API as the Hypersonic API. By doing so, we study the requirements and potential setbacks of processing models via RESTful Web services.

In keeping with the REST architectural style [10], the Hypersonic API exposes resources for clients to interact with via HTTP requests. The two exposed resources are *models* and *model*. The *models* resource plays the role of an access point to Hypersonic's model warehouse, whereas the *model* resource represents a single model in the warehouse. These resources are manipulated via HTTP requests, where the HTTP method determines the operation to be performed. In addition, the Hypersonic URL scheme specifies explicit operations on the *model* resource as part of the request URL. The list of supported operations is presented in Table 2.

This architectural approach allows physically decoupling clients from the execution of the various analysis algorithms exposed by the Hypersonic API. This aspect is part of the motivation behind the creation of Hypersonic, since many of these algorithms are resource demanding on models of non-trivial size. By using such a Web service API, a large variety of clients can have access to model analysis facilities regardless of their hardware capabilities. Fig. 2 highlights this aspect, showing that different clients can access existing analysis algorithms provided by the MACH tool via the Hypersonic API wrapper. The only prerequisites for API clients are HTTP support, the ability to process documents returned by the API, and, optionally, a model viewer. Note that all of these prerequisites are entirely reasonable for modern mobile devices.

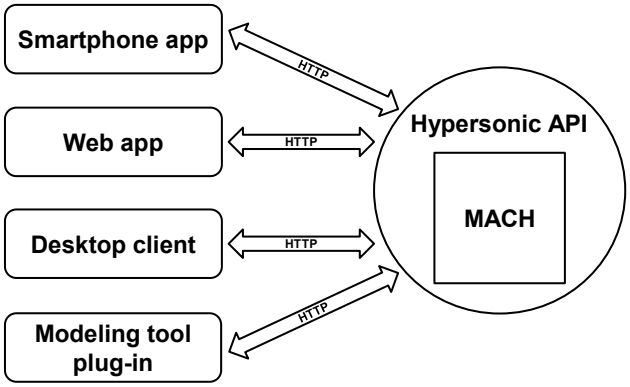



**Figure 2: High-level architecture of Hypersonic**

Additional non-functional considerations must be taken into account to ensure the practicality of the Web service API. Since using the API implies uploading entire models to a remote server, security becomes an important factor. With this in mind, the OAuth [11] authentication protocol is a widely used solution that can provide some important guarantees to Hypersonic users. The most important such guarantee is that a user cannot gain access to the models uploaded by other users. When combined with a role-based authentication policy, a sound authentication mechanism such as OAuth is an effective way to manage model access rights. At a technical level, implementing OAuth will require all Hypersonic API clients to obtain an access token prior to using the API. This process can be carried out through a separate channel, such as a dedicated API management Web application.

From a file format standpoint, Hypersonic currently supports models stored in the MDXML format, the XMI-based native format of the MagicDraw modeling tool. That is to say, some API requests (e. g., POST requests to the *models* resource) are expected to have an MDXML document as payload. Most API response messages carry a JSON [12] document as payload, representing either the result of an analysis operation or a confirmation or error message.

The internal components involved in answering a call to the Hypersonic API are presented in Fig. 3. The *RESTful API* component handles HTTP communication with remote API clients and delegates all actual model processing to the *MACH* component. It also forwards all models sent by clients to the *XMI2PL* component, which performs a format translation from the MDXML format to the internal Prolog-based file format described in [26]. Once translated, models are stored in a dedicated model warehouse for future analysis upon the client's request. The *MACH* component exposes several supported model analysis and checking algorithms [27]. These algorithms can be applied on models stored in the warehouse. The *MACH* component functions as a self contained black-box, hiding all algorithm implementations from other components and returning the produced results encoded as Prolog lists. The *RESTful API* component handles the translation of these lists into JSON analysis reports ready for consumption by the API client. All processing components are executed inside a single instance of the SWI-Prolog runtime [29], thus allowing seamless inter-component communication.

The SWI-Prolog runtime should be deployed to either a public or private Cloud platform. Since all models are stored separately in a model warehouse, several instances of the SWI-Prolog runtime can be deployed, assuming that the model warehouse provides appropriate concurrent access policies. Persistent model storage can be provided by a separate Cloud storage service. Since models are stored as XML and Prolog files, the storage service should support a document-oriented database management system.

# 4. EVALUATION

To demonstrate the feasibility of the architecture described in Section 3, a subset of the proposed Web service API has been implemented and is publicly accessible[1]. Due to the reasons elaborated on in Section 2.3, we have focused on a Web service providing model clone detection as a minimum useful scenario. Though important for a final release, we have considered features such as user accounts and authentication beyond the scope of our proof of concept. Both the prototype API and the model warehouse are currently hosted on a dedicated server. They do not benefit from the scalability of a true Cloud deployment, although for the current proof of concept this is hardly a limiting factor.

[1]The Hypersonic API is available at the following base URL: `http://hypersonic.compute.dtu.dk`

| Resource | Method | Req. payload | Resp. payload | Description |
|---|---|---|---|---|
| /models | GET | — | JSON | List all uploaded models. |
| /models | POST | MDXML | JSON | Upload a model. |
| /model/<id> | GET | — | MDXML | Download a model. |
| /model/<id> | PUT | MDXML | JSON | Replace a model. |
| /model/<id> | DELETE | — | JSON | Delete a model. |
| /model/<id>/clones | GET | — | JSON | Detect clones in a model. |
| /model/<id1>/diff/<id2> | GET | JSON | JSON | List differences between two models. Options are specified in the request payload. |
| /model/<id>/dump | GET | — | JSON | List model elements included in a model. |
| /model/<id>/dump/<me_id> | GET | JSON | JSON | List the details of a model element. Options are specified in the request payload. |
| /model/<id>/find/<string> | GET | JSON | JSON | Find a string in a model. Options are specified in the request payload. |
| /model/<id>/frequency | GET | — | JSON | Compute the meta class frequency distribution in a model. |
| /model/<id1>/similarity/<id2> | GET | JSON | JSON | Compute the similarity between two models. Options are specified in the request payload. |
| /model/<id>/size | GET | JSON | JSON | Compute the size of a model. Options are specified in the request payload. |

**Table 2: List of operations supported by the Hypersonic API**

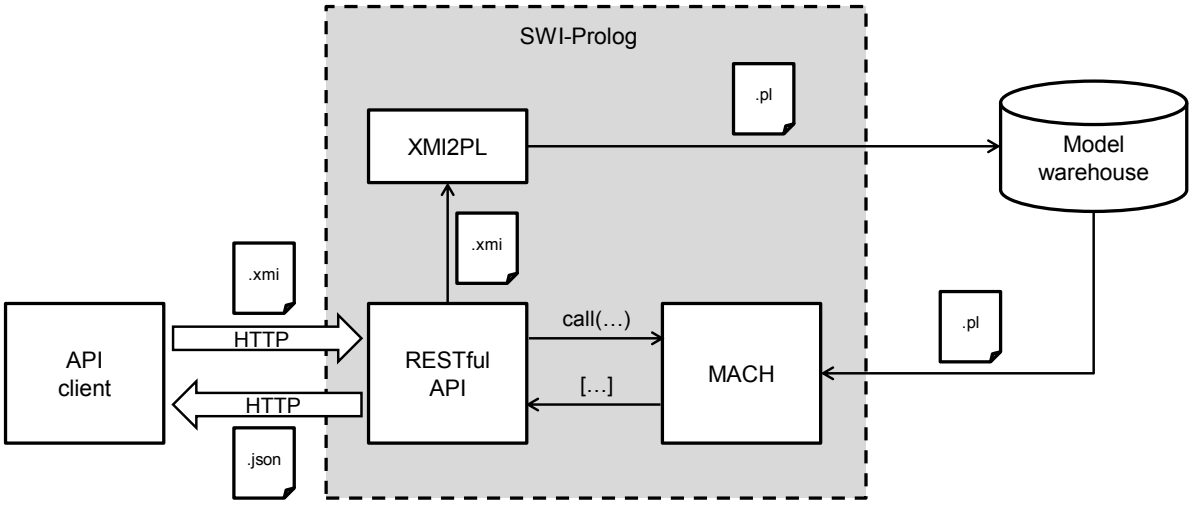



**Figure 3: Components which participate in responding to a Hypersonic API request**

Fig. 4 represents a message exchange between a client and the Hypersonic API. The purpose of this exchange is to perform clone detection on a model. First, the model is added to the Hypersonic model warehouse by a POST request to the *models* resource. Upon this request's successful handling, a new *model* resource representing the model is available to the client. The resource has a unique identifier returned in the JSON response document of the initial POST request. The client then parses this document and extracts the identifier. Thus, the client is subsequently able to use the identifier to construct the appropriate URL for a GET request to the *clones* operation of the identified *model* resource. The GET request returns a list of clone candidates, also in the form of a JSON document (see Listing 1).

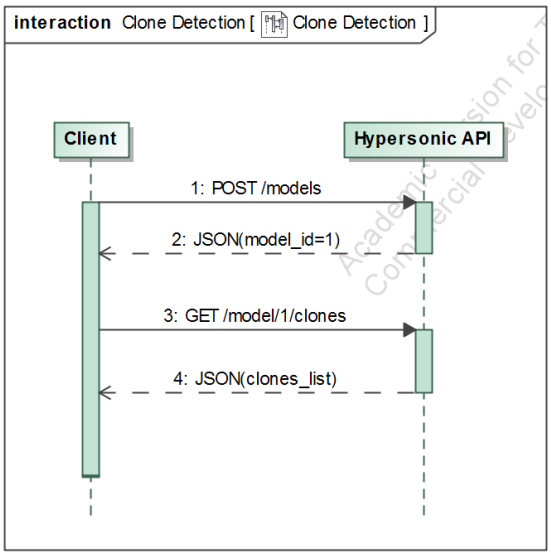



**Figure 4: HTTP message exchange for model clone detection**

The response document includes a model identifier, the number of detected clones, and a list of discovered clone candidates. Each clone candidate is described by two model elements, where one is a possible clone of the other, a numeric similarity metric computed for the two elements following the approach presented in [26], and a clone "kind" identifying the candidate as either a naturally occurring clone or a seeded clone. Candidates also Clone candidates are returned in the descending order of their similarity scores, i. e., the most likely clone is the first one in the list.

**Listing 1: JSON clone detection report**

```
{
  "model": "1",
  "candidates": 2,
  "clones": [
    {
      "type_1": "package",
      "id_1": 29,
      "name_1": "Reserve Medium",
      "type_2": "package",
      "id_2": 938,
      "name_2": "Reserve Medium",
      "similarity": 185.7143,
      "kind": "natural clone",
    }
    {
      "type_1": "package",
      "id_1": 189,
      "name_1": "Lend Medium",
      "type_2": "package",
      "id_2": 1194,
      "name_2": "Lend Medium",
      "similarity": 128.7287,
      "kind": "natural clone",
    }
  ]
}
```

By conforming to the architecture presented in Fig. 3, the effort required to implement the clone detection proof of concept has been minimal. The RESTful API functioning as a wrapper around MACH's existing clone detection implementation consists of around 100 lines of Prolog code, largely thanks to the comprehensive support offered by SWI-Prolog for the HTTP protocol. Work on implementing the remaining API calls described in Table 2 is ongoing, as is work on the API management application that must be in place in order to to enable user authentication in API calls.

As a preliminary validation of the API's fitness for purpose, we have created a simple, mobile device friendly Web application as an API client[2]. The application supports selecting a local model file, uploading it to the Hypersonic model warehouse, and requesting a clone report which it subsequently displays in tabular form. The application is written in JavaScript and is executed entirely in the browser (i. e., it does not rely on a server-side script for calling API operations). Though it is so far basic in terms of functionality, this sample client exemplifies our vision of Web service driven modeling tools: using Web 2.0 technologies (REST, JavaScript, JSON) to enable advanced model analysis outside the constraints of the desktop and of traditional modeling environments.

## 5. RELATED WORK

Model analysis is an activity typically performed in local, non-distributed environments. As an example, the model clone detection operation considered here as a proof of concept has scarcely been addressed in itself, but is closely related to the intensely studied model matching and differencing operations. To name just a few proposals in this area, SiDiff [14] presents a differencing algorithm targeting UML Class Diagrams, while the approach presented in [19] targets sequence charts, and [17] is a clone detection proposal aimed at UML Sequence Diagrams. EMF DiffMerge [8] and EMF Compare 3.0 [3] represent more generic approaches targeting the Eclipse Modeling Framework (EMF, [9]).

With the increase in size of industrially relevant models and the increase in complexity of the operations performed on these models, the need for distributed, Cloud-enabled modeling solutions has become apparent [5, 16]. So far, the main driver behind Cloud-based modeling research has been

[2]The client application is available at `http://www.compute.dtu.dk/~rvac/hypersonic`. It is currently under development, and will be updated to support all operations of the Hypersonic API as they are deployed.

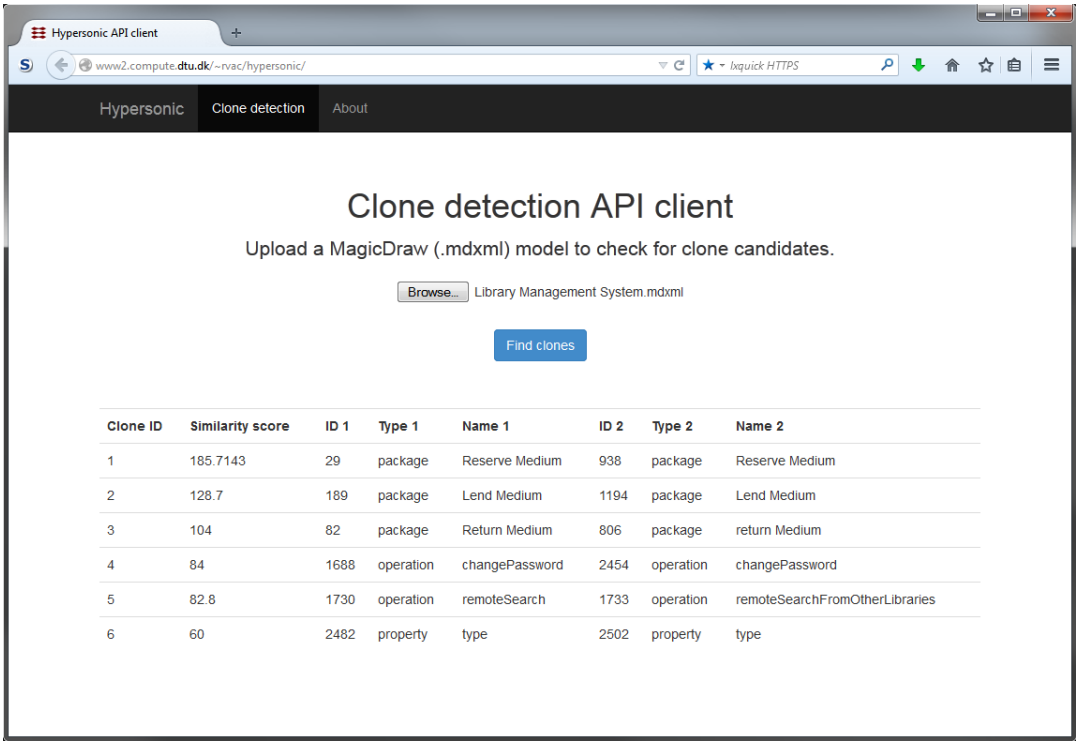



**Figure 5: Screenshot of the Hypersonic API client Web application**

the promise of important performance and scalability gains for all modeling activities. Perhaps the most fundamental of these activities, model storage, has attracted several proposals, including ModelBus [4], EMFStore [15], and Morsa [23]. These are all remote model warehousing solutions offering Web service access to the stored models.

More advanced activities such as model querying and transformation have also been addressed. IncQuery-D [13] is a tool which takes the established IncQuery tool and adapts it to a scenario where it can be deployed and accessed in the Cloud. The Morsa model repository also benefits from a dedicated query language, MorsaQL [24]. A roadmap for research on Cloud-based model transformations has been presented in [7].

However, performance gains due to Cloud deployment are only a part of the overall vision of Hypersonic. Rather than focusing on the benefits to the application itself (i. e., model analysis), Hypersonic emphasises the benefits brought by a Cloud-based approach to the interface and availability of this application. The idea of performing model analysis via a RESTful Web service API has yet to receive significant attention in the literature. The closest related proposal is the EMF-REST project [6], aimed at automatically generating RESTful Web service interfaces for EMF models, much like existing EMF tools generate Java APIs for such models. Like Hypersonic, EMF-REST uses JSON documents to transport information about remotely stored models. Nevertheless, while it does provide basic model manipulation operations, EMF-REST is not designed as a model analysis tool. Similarly to Hypersonic, EMF-REST is a tool under ongoing development, one of its current limitations being the lack of full support for HTTP methods other than GET.

# 6. CONCLUSIONS

## 6.1 Summary

In this paper we have discussed the application conditions, benefits, and general business case for Cloud-based modeling tools. In particular, we have presented a scenario in which modeling capabilities are delivered as Web services to a wide array of clients, ranging from desktop applications to Web and mobile applications. We have contrasted this scenario with the current status-quo of rich client desktop modeling tools, reaching the conclusion that, in many respects, the Cloud-based approach is superior.

To explore our proposal, we have introduced Hypersonic, a RESTful Web service API aimed at offering high-throughput processing for Cloud-based model analysis. We have implemented this architecture and made it available online. Currently, the only service it offers is the detection of model clones, a feature that was previously only available in the MACH command line tool. Today, MACH is a stand-alone desktop tool providing only a textual user interface. Through the Hypersonic API, the features of MACH can be made available over the Internet to any API client. As a proof of concept for the utility of the API, we have developed a Web application acting as a client to the Hypersonic API and providing Web-based model analysis capabilities.

## 6.2 Future Work

The concepts presented in this paper offer us ample opportunities for future work. As a first step, we will continue the development of the Hypersonic API with the aim of reaching functional parity with the MACH model analysis tool. Once this has been achieved, the API will be deployed to

a Cloud platform. In parallel, we will update the sample Web-based API client to both validate and showcase the model analysis features of Hypersonic. Second, in order to become a practical tool, the API client must offer several critical features such as user authentication and model security mechanisms. Third, we will carry out a systematic performance evaluation of MACH in order to substantiate the claim that Cloud-based model analysis can bring significant performance benefits. Finally, we intend to develop a second client for the Hypersonic API in the shape of a plug-in for MagicDraw. This will permit seamless integration of our Web services approach with a commercial modeling tool and complement our existing model querying MagicDraw plug-in, MQ-2 [2]. As a parallel development, we envision a Web service API similar to Hypersonic for RED, our requirements engineering tool [28].

# 7. REFERENCES


[1] GenMyModel. `http://www.genmymodel.com`, retrieved 16.05.2014.

[2] V. Acretoaie and H. Störrle. MQ-2: A Tool for Prolog-based Model Querying. In *Proc. co-located Events 8th Eur. Conf. on Modelling Foundations and Applications (ECMFA'12)*, pages 328–331.

[3] M. Barbero. EMF Compare 3.0. `http://www.eclipse.org/emf/compare`.

[4] X. Blanc, M.-P. Gervais, and P. Sriplakich. Model Bus: Towards the Interoperability of Modelling Tools. In *Proc. European MDA Workshops: Foundations and Applications (MDAFA'03/'04)*, volume 3599 of *LNCS*, pages 17–32. Springer Berlin Heidelberg, 2005.

[5] H. Bruneliere, J. Cabot, F. Jouault, et al. Combining Model-Driven Engineering and Cloud Computing. In *Proc. 4th Ws. on Modeling, Design, and Analysis for the Service Cloud (MDA4ServiceCloud'10)*, 2010.

[6] J. Cabot. EMF-REST. `http://emf-rest.com`, retrieved 16.05.2014.

[7] C. Clasen, M. D. Del Fabro, and M. Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. In *Proc. First Intl. Ws. Model-Driven Engineering on and for the Cloud (CloudMDE'12)*, pages 3–12, 2012.

[8] O. Constant. EMF Diff/Merge. `http://wiki.eclipse.org/EMF_DiffMerge`.

[9] Eclipse Foundation, Inc. Eclipse Modeling Framework (EMF). `http://eclipse.org/modeling/emf`.

[10] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] Internet Engineering Task Force (IETF). IETF RFC 6749: The OAuth 2.0 Authorization Framework. `http://tools.ietf.org/html/rfc6749`, 2012.

[12] Internet Engineering Task Force (IETF). IETF RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format. `http://tools.ietf.org/html/rfc7159`, 2014.

[13] B. Izsó, G. Szárnyas, I. Ráth, and D. Varró. IncQuery-D: Incremental Graph Search in the Cloud. In *Proc. Ws. Scalability in Model Driven Engineering (BigMDE'13)*, pages 4:1–4:4, New York, NY, USA, 2013. ACM.

[14] U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. In K. Pohl, editor, *Proc. Natl. Germ. Conf. Software-Engineering (SE'05)*, number P-64 in Lecture Notes in Informatics, pages 105–116. Gesellschaft für Informatik e.V. 2005.

[15] M. Koegel and J. Helming. EMFStore: a Model Repository for EMF models. In *Proc. 32nd ACM/IEEE Intl. Conf. on Software Engineering (ICSE'10)*, volume 2, pages 307–308. ACM, 2010.

[16] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proc. Ws. Scalability in Model Driven Engineering (BigMDE'13)*, pages 2:1–2:10, New York, NY, USA, 2013. ACM.

[17] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *13th Asia Pacific Software Engineering Conf. (APSEC)*, pages 269–276. IEEE CS, 2006.

[18] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology, 2011.

[19] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *Proc. 29th Intl. Conf. Software Engineering (ICSE)*, pages 54–64. IEEE Computer Society, IEEE Computer Society, 2007.

[20] NoMagic INC. MagicDraw Teamwork Server. `http://www.nomagic.com/products/teamwork-server`, retrieved 16.05.2014.

[21] NoMagic INC. MagicDraw UML 17.0.3. `http://www.nomagic.com/products/magicdraw`, retrieved 16.05.2014.

[22] Object Management Group (OMG). OMG MOF 2 XMI Mapping Specification, Version 2.4.1. `http://www.omg.org/spec/XMI/2.4.1`, 2013.

[23] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Proc. 14th Intl. Conf. Model Driven Engineering Languages and Systems (MODELS'11)*, volume 6981 of *LNCS*, pages 77–92. Springer Berlin Heidelberg, 2011.

[24] J. E. Pagán and J. G. Molina. Querying Large Models Efficiently. *Inf. Softw. Tech.*, pages 586–622, 2014.

[25] Pocketworks. yUML. `http://yuml.me`, retrieved 16.05.2014.

[26] H. Störrle. Towards Clone Detection in UML Domain Models. *J. Softw. Syst. Model.*, 12(2), 2013.

[27] H. Störrle. UML Model Analysis and Checking with MACH. In *4th Intl. Ws. Academic Software Development Tools and Techniques (WASDETT'13)*, 2013.

[28] H. Störrle and M. Kucharek. The Requirements Editor RED. In *ECOOP, ECSA and ECMFA 2013: Joint Proceedings of Tools, Demos and Posters*, pages 32–34, 2013. DTU Technical Report 2014-01.

[29] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

# Towards an Open Set of Real-World Benchmarks for Model Queries and Transformations

Amine Benelellam
AtlanMod team
Inria, Mines-Nantes, Lina
Nantes, France
amine.benelallam@inria.fr

Massimo Tisi
AtlanMod team
Inria, Mines-Nantes, Lina
Nantes, France
massimo.tisi@inria.fr

István Ráth
Budapest University of
Technology and Economics
Budapest, Hungary
rath@mit.bme.hu

Benedek Izsó
Budapest University of
Technology and Economics
Budapest, Hungary
izso@mit.bme.hu

Dimitrios S. Kolovos
Enterprise Systems Group
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

## ABSTRACT

With the growing size and complexity of systems under design, industry needs a generation of Model-Driven Engineering (MDE) tools, especially model query and transformation, with the proven capability to handle large-scale scenarios. While researchers are proposing several technical solutions in this sense, the community lacks a set of shared scalability benchmarks, that would simplify quantitative assessment of advancements and enable cross-evaluation of different proposals. Benchmarks in previous work have been synthesized to stress specific features of model management, lacking both generality and industrial validity. In this paper, we initiate an effort to define a set of shared benchmarks, gathering queries and transformations from real-world MDE case studies. We make these case available to community evaluation via a public MDE benchmark repository.

## Categories and Subject Descriptors

D.2.2 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.8 [**Software Engineering**]: Metrics—*performance measures, Complexity measures*; K.6.3 [**Computing Milieux**]: Software Management—*Software selection*

## General Terms

Performance, Experimentation, Measurement

## Keywords

Benchmarking, Very Large Models, Model transformations, Model queries

## 1. INTRODUCTION

Over the past decade, technologies around Model-Driven Engineering (MDE) have been offering a systematic approach for software development on top of models. This elegant approach conquered an interesting popularity and engaged both researchers and industrials. However MDE is not able yet to attract large-scale industrial systems considering the serious scalability issues that the current generation of MDE tools is facing, namely (i) the growing complexity of systems under design, and (ii) the huge amount of data they might represent. Therefore, there is a calling need to develop a new generation of tools capable of managing, querying, and transforming Very Large Models (VLMs). Existing empirical assessment [7, 36, 21] has accredited that necessity indeed.

In spite of the advances on Model Transformation (MT) tools, additional research studies are still needed in order to acquire the attention of large-scale industrial systems. Aiming at improving performance and computation cost, many works have been developed around elaborating new features for existing model transformation engines (e. g., *Incrementality* [18, 33, 9], and *Parallelization* [10, 38, 51]). Others [14] chose to develop new tools based on existing frameworks effectively handling concurrency and parallelization.

One of the most computationally expensive tasks in modeling applications is the evaluation of *model queries*. While there exists a number of benchmarks for queries over relational databases [52] or graph stores [12, 50], modeling tool workloads are significantly different. Specifically, modeling tools use more complex queries than typical transactional systems [29], and the real world performance is affected by response time (i.e. execution time for a specific operation such as validation or transformation) than throughput (i.e. the amount of parallel transactions).

In order to overcome these limitations and achieve scalability in MDE, several works [16, 37] drew research roadmaps advancing the use of the Cloud for distributed and collaborative processing of VLMs. Hence, in order to be able to compare this new generation of tools and measure their performance, it is required to provide a transformation benchmark that takes into consideration the well-known scalability issues in MTs [35]. Likewise, benchmarks might be a good reference to help engineers in choosing what fits the most to their solution while developing new tools.

Most of existing benchmarks [32, 8, 54, 55] were more focused

*BigMDE '14* July 24, 2014. York, UK.

on other properties than scalability (e. g., Transformation sequence length, Transformation strategy, Matching size etc.). Moreover these benchmarks do not provide neither any clue on how to measure transformation scalability from a theoretical point of view, nor real-world case studies. In contrast to the former benchmarks, Izsó et al. [29] are the first ones to provide a precise metrics to predict the performance of graph queries – based on instance models and query specification –, and therefore results in indicators that help selecting the suitable technology. In addition, these benchmarks are specific to either model transformation or model query. On the other side, there exist some reference benchmarks for databases with a higher level of maturity [15, 49]. In [15], Cattel et al. present a benchmark OO1 to measure the performance of specific characteristics of the most frequently used operations – according to feedbacks from industrials – on engineering objects. This benchmark come with a precise specification, measures, and evaluation indeed. In [49] Schmidt et al. introduce XMark, a benchmark for XML data management that copes with a large range of queries coming from real world case scenarios. Each query comes to stress a particular aspect of XML query engines.

In this paper we present a set of benchmarks for model transformation and query engines. Our proposal is to select a set of transformations/queries from real-world cases and to make them available to large public. Two of the four benchmarks included deal with model transformations, while the other two deal with model queries. Models that feed each one of the benchmarks are of increasing size, also different kinds/extensions. They are either concrete, coming from real projects (i. e., reverse engineered Java project models) or generated using deterministic model instantiators. These instantiators can be easily customized to be able to generate models that suit the benchmarking of a specific feature.

These benchmarks are part of the results of the Scalable Modeling and Model Management on the Cloud (MONDO)[1] research project, that aims at advancing MDE's state of the art to tackle very large models [37]. We plan to involve the community in order to build a larger set of case studies covering additional properties/domains (i. e., verification and simulation of critical systems).

The rest of the paper is organized as follows. In Section 2 we describe benchmarks from the state-of-the-art. In Section 3 we outline the four different sets of benchmarks provided in the scope of the paper. Section 4 describes usage and contribution modalities for the repository. Finally, Section 5 provides a conclusion and recapitulates the future plans for the benchmarks suite.

## 2. RELATED WORK

A few works in literature [54, 55, 31, 8] proposed benchmarks to assist developers in selecting the most suitable query/transformation technology for their application scenarios. However only one of the existing case studies is explicitly dedicated to the manipulation of very large models ([31]) and none is based on benchmarks queries and transformations based on real-world applications. In this section we give a short overview of the related works, while in the next section we introduce the real-world benchmarks proposed within this paper.

One of the widely used benchmarks in MDE is Grabats'09 Reverse Engineering [31], where Jouault et al. proposed a case study that consists of the definition of program comprehension operators (i. e., queries over source code) using a graph or model transformation tool. One of the main challenges in the definition of program comprehension operators as transformations is scalability with respect to input size. This case study is divided into two independent tasks (i) a simple filtering query that selects a subgraph of its input according to a given condition, and (ii) a complex query that computes a control flow graph and a program dependence graph. These queries are performed over the JDTAST metamodel, the Java metamodel used in early versions of MoDisco [13] to discover Java projects. This benchmark comes with 5 different sets of increasing size ranging from $7 \times 10^5$ up to $5 \times 10^9$.

The experiment in [54] compares the performance of three model transformation engines: ATL, QVT-R, and QVT-O. This comparison is based on two different transformation examples, targeting meta-models with different structural representations: linear representation (Class2RDBMS) and tree-like representation (RSS2ATOM). The benchmarking set involves randomly generated input models of increasing numbers of elements (up to hundreds of thousands). Like the previous work [55], the benchmark sets are also tuned according to a particular feature such as the size of input models, their complexity (complex interconnection structure) and transformation strategies. In order to study the impact of different implementation strategies in ATL, the Class2RDBMS transformation was implemented in different programming styles. The first one promotes expressing input models navigation in the in the right-hand side of the bindings, the second use ATL attribute helpers, and third uses the imperative part of ATL.

The work [55] is considered one of the early systematic MDE benchmarks dedicated to Graph Transformations (GT). It proposes a methodology for quantitative benchmarking in order to assess the performance of GT tools. The overall aim is to uniformly measure the performance of a system under a deterministic, parametric, and especially reproducible environment. Four tools participated in the experimentation: AGG, PROGRES, FUJABA and DB. Every benchmarking set is tuned according to some features related on one side to the graph transformation paradigms, and on the other side to the surveyed tools. Thus, a benchmarking set is characterized by turning on/off these features. Bergmann et al. extended this benchmark with incrementality. Two kinds of benchmarks kind were carried out, simulation and synchronization, for which, a benchmark-specific generators has been provided. The benchmarks were run over different implementations of pattern matchers, VIATRA/LS (Local Search), VIATRA/RETE, and GEN.NET with the distributed mutual exclusion algorithm.

## 3. BENCHMARKS SET

The benchmarks set has the purpose to evaluate and validate a proposed query and transformation engine. This set of benchmarks is made public, with the intention to also help both research groups and companies to assess their solutions.

In this section we describe the source, context and properties of the benchmarks. The set of benchmarks is designed to cover the main use cases for queries and transformations in model-driven applications. Table 1 summarizes the characteristics of the four benchmarks in terms of type of computation (query/transformation) and computational complexity (high/low).

Each one of the benchmarks either includes concrete source models, or a model generator that can be used to produce models of different sizes in a deterministic manner. In the latter case, models

[1] http://www.mondo-project.org/

**Table 1: Summary of the MONDO WP3 benchmarks**

| Benchmark | Type | Computational complexity |
|---|---|---|
| Train benchmark | query | high |
| Open-BIM | query/transformation | low |
| ITM Factory | transformation | high |
| Transformation zoo | transformation | low |

of different sizes can be generated, but seeds are provided to drive the deterministic generators in producing the same models for each user.

Each benchmark in the set is given a reference implementation that has to be considered as a specification of the case semantics. Languages and technologies used for each reference implementation may vary, including MDE-specific and general-purpose technologies.

Finally, while each benchmark defines the source/target relation for each query or transformation, other aspects of the transformation runtime semantics are left open. For instance high-complexity benchmarks can be run in batch or incremental mode, to test different execution properties of the tool under study.

## 3.1 Train Benchmark

### *3.1.1 Context and objectives*

The Train Benchmark [53, 1] is a macro benchmark that aims to measure batch and incremental query evaluation performance, in a scenario that is specifically modeled after *model validation* in (domain-specific) modeling tools: at first, the entire model is validated, then after each model manipulation (e.g., the deletion of a reference) is followed by an immediate re-validation. The benchmark records execution times for four phases:

1. During the *read* phase, the instance model is loaded from hard drive to memory. This includes the parsing of the input as well as initializing data structures of the tool.
2. In the *check* phase, the instance model is queried to identify invalid elements. This can be as simple as reading the results from cache, or the model can be traversed based on some index. By the end of this phase, erroneous objects need to made available in a list.
3. In the *edit* phase, the model is modified to simulate effects of manual user edits. Here the size of the change set can be adjusted to correspond to small manual edits as well as large model transformations.
4. The re-validation of the model is carried out in the *re-check* phase similarly to the *check* phase.

The Train Benchmark computes two derived results based on the recorded data: (1) *batch validation time* (the sum of the *read* and check phases) represents the time that the user must wait to start to use the tool; (2) *incremental validation time* consists of the *edit* and *re-check* phases performed 100 times, representing the time that the user spent waiting for the tool validation.

### *3.1.2 Models and metamodels*

The Train Benchmark uses a domain-specific model of a railway system that originates from the MOGENTES EU FP7 project, where both the metamodel and the well-formedness rules were defined by railway domain experts. This domain enables the definition of both simple and more complex model queries while it is uncomplicated enough to incorporate solutions from other technological spaces (e.g. ontologies, relational databases and RDF). This allows the comparison of the performance aspects of wider range of query tools from a constraint validation viewpoint.

The instance models are systematically and reproducibly generated based on the metamodel and the defined complex model queries: small instance model fragments are generated based on the queries, and then they are placed, randomized and connected to each other. The methodology takes care of controlling the number of matches of all defined model queries. To break symmetry, the exact number of elements and cardinalities are randomized (with a fixed seed to ensure deterministic reproducibility). In the benchmark measurements, model sizes ranging from a few elements to 13 million elements (objects and references combined) are considered.

This brings artificially generated models *closer to real world instances*, and *prevents query tools from efficiently storing* or caching of instance models. This is important in order to reduce the sampling bias of the experiments. During the generation of the railway system model, errors are injected at random positions. These errors can be found in the check phase of the benchmark, which are reported, and can be corrected during the edit phase. The initial number of constraint violating elements is low (<1% of total elements).

### *3.1.3 Queries and transformations*

Queries are defined informally in plain text (in a tool independent way) and also formalized using the standard OCL language as a reference implementation (available on the benchmark website [1]). The queries range from simple attribute value checks to complex navigation operations consisting of several (4+) joins.

The functionally equivalent variants of these queries are formalized using the query language of different tools applying tool based optimizations. As a result, all query implementations must return (the same set of) invalid instance model elements.

In the *edit* phase, the model is modified to change the result set to be returned by the query in the re-check phase. For simulating manual modifications, the benchmark always performs a hundred random edits (fixed low constant) which increases the number of erroneous elements. An edit operation only modifies one model element at a time - more complex model manipulation is modeled as a series of edits.

The Train Benchmark defines a Java-based framework and application programming interface that enables the integration of additional metamodels, instance models, query implementations and even new benchmark scenarios (that may be different from the original 4-phase concept). The default implementation contains a benchmark suite for queries implemented in Java, Eclipse OCL and EMF-IncQuery.

Measurements are recorded automatically in a machine-processable format (CSV) that is automatically processed by R [2] scripts. An extended version of the Train Benchmark [29] features several (in-

stance model, query-specific and combined) *metrics* that can be used to characterize the "difficulty" of benchmark cases numerically, and – since they can be evaluated automatically for other domain/model/query combinations – allow to compare the benchmark cases with other real-world workloads.

## 3.2 Open-BIM

### *3.2.1 Context and objectives*

This benchmark includes a model validation and a model transformation in the context of construction models. The construction industry has traditionally communicated building construction information (sites, buildings, floors, spaces, and equipment and their attributes) through drawings with notes and specifications. BIM (Building Information Model), a CAD (Computer Aided Design) method, came to automate that process and enhance its operability according to different tools, actors, etc. within the AECO (Architecture, Engineering, Construction, and Operations) industry. A BIM model is a multi-disciplinary data specific model instance which describes all the information pertinent to a building and its components.

A BIM model is described using the IFC (Industry Foundation Classes) specification, a freely available format to describe, exchange, and share information typically used within the building and facility management industry sector. Intrinsically, IFC is expressed using the EXPRESS data definition language, defined as ISO10303-11 by the ISO TC184/SC4 committee. EXPRESS representations are known to be compact and well suited to include data validation rules within the data specification.

### *3.2.2 Models and metamodel*

The repository contains 8 real-world IFC data files with size ranging from 40MB to 1GB. All the files represent real construction projects and were used in production context. They contain a precise and detailed information about actors, approvals, buildings etc. The data files, in EXPRESS format, are translated into EMF models so that they can be used by EMF-based query and transformation tools.

### *3.2.3 Queries and transformations*

The Open-BIM use case includes a query benchmark and a transformation benchmark:

*IFC well-formedness rules.* The IFC format describes, using the EXPRESS language, the set of well-formed IFC models. The EXPRESS notation includes, in a single specification, 1) the set of element types and properties allowed in the data file, 2) the set of well-formedness constraints that have to be globally satisfied. When representing an IFC model in an EMF format these two parts of the specification translate to 1) an Ecore metamodel defining element and property types, 2) a set of constraints encoding the well-formedness rules.

This benchmark involves validating the set of well-formedness rules (2) over a given model, model that conforms to the IFC Ecore metamodel (1). An Ecore metamodel is provided, coming from the open-source BIMServer [2] project. The well-formedness rules are given in EXPRESS format and are meant to be translated to the query technology under evaluation.

[2]https://github.com/opensourceBIM/BIMserver

*IFC2BIMXML.* BIMXML [3] is an XML format describing building data in a simplified spatial building model. The BIMXML XML Schema was developed as an alternative to full scale IFC models to simplify data exchanges between various AEC applications and to connect Building Information Models through Web Services. It is currently used by several primary actors in the CAD construction domain, including Onuma System (Onuma, Inc.), DDS Viewer (Data Design System), vROC, Tokmo, BIM Connect, and various plugins for CAD Applications (Revit, SketchUp, ArchiCAD). The BIMXML specification includes an XML Schema[4] and documents the translation rules from the full IFC specification.

This benchmark involves performing the translation of a full IFC model into the BIMXML format. Ecore metamodels for the source and target models are provided.

## 3.3 ITM Factory

### *3.3.1 Context and objectives*

This benchmark contains two transformations and a set of queries, each addressing a different phase in a model-driven reverse engineering process. The use case for this benchmark is taken from the Eclipse MoDisco project.

MoDisco (Model Discovery) is the open-source Model Driven Reverse Engineering project lead by the company Soft-Maint. It uses a two steps approach with a **model discovery** followed by **model understanding**. The initial step consists in obtaining a model representation of a specific view on the legacy system, whereas, the second involves extracting useful information from the discovered model. MoDisco requires high efficiency in handling large models, especially these involved in reverse engineering of legacy systems.

### *3.3.2 Models and metamodel*

Thanks to the MoDisco Java discoverer, we are able to extract Java models up to 1.3GB, that conform to the Java metamodel [39] defined in MoDisco (refinement of the JDTAST metamodel). Those models are the input of the Java2KDM and Java code quality transformations, while, KDM output models are inputs for the KDM2UML transformation. It is also possible to retrieve directly KDM models using MoDisco. Because of confidentiality agreements, Soft-Maint is not able to publish instance models derived from their commercial projects. For this reason we choose to derive instance models from the source code of open-source projects, specifically from the Eclipse JDT plugins (org.eclipse.jdt.*). This does not affect the relevance of the benchmark, as these plugins are written by experienced developers with a quality standard that is comparable to commercial projects.

Table 2 depicts the different models recovered against the discovered plugins.

### *3.3.3 Queries and transformations*

*Java2KDM.* This transformation takes place at beginning of almost every modernization process of a Java legacy system, it comes just after the discovery of the Java model from Java projects (plugins) using the MoDisco Java Discoverer. This transformation generates a KDM [44] (Knowledge Discovery Metamodel) model that defines common metadata required for deep semantic integration of

[3]http://bimxml.org/

[4]http://www.bimxml.org/xsd/001/bimxml-001.xsd

| | |
|---|---|
| Set1 | org.eclipse.jdt.apt.pluggable.core |
| Set2 | Set1 + org.eclipse.jdt.apt.pluggable.core |
| Set3 | Set2 + org.eclipse.jdt.core+ org.eclipse.jdt.compiler + org.eclipse.jdt.apt.core |
| Set4 | Set3 + org.eclipse.jdt.core.manipulation + org.eclipse.jdt.launching + org.eclipse.jdt.ui + org.eclipse.jdt.debug |
| Set5 | org.eclipse.jdt.* (all jdt plugins) |

**Table 2: Discovered plugins per set**

application life-cycle management tools. Java2KDM transformation is useful when the only available information on a Java source code is contained in a Java model, even without the source code it is then possible to get a KDM representation. This intermediate model provides useful and precise information that can be used to produce additional types of models.

*KDM2UML.* Based on the previously generated model, this transformation generates a UML diagram in order to allow integrating KDM-compliant tools (i. e., discoverers) with UML-compliant tools (e.g. modelers, model transformation tools, code generators, etc.).

*Java code quality.* This set of code quality transformations identify well-known anti-patterns in Java source code and fix the corresponding issues by a model transformation. The input format of the transformations is a model conforming to the Java metamodel. For a specification for the transformations we refer the reader to the implementations of these fixes in well-known code-analysis tools like CheckStyle and PMD. Table 3 summarizes the references to the documentation for each code fix considered in this benchmark.

## 3.4 ATL Transformation Zoo

### *3.4.1 Context and objectives*

The ATL project maintains a repository of ATL transformations produced in industrial and academic contexts (ATL Transformation Zoo [28]). These transformations are representative of the use of model transformations for low-complexity tasks (i.e., low number of transformation rules, lack of recursion, etc. . . ).

In this benchmark we select a subset of the transformations in the ATL Transformation Zoo based on their quality level and usage in real-world applications. We specifically include only transformations that may be used in production environments. We automatize the sequential execution of this subset and the generation of performance analysis data.

### *3.4.2 Models and metamodels*

For the aforementioned transformations, we do not have large enough models that conform to the respective metamodels, and as such we make use of a probabilistic model instantiator. This instantiator takes as parameter a generation configuration specified by the user. A generation configuration holds information such as 1) metaclasses that should (not) be involved in the generation, 2) probability distributions to establish how many instances should be generated for each metaclass, and which values should be assigned to structural features. We provide a default generation configuration, using uniform probability distributions for each meta-class and structural feature. For some transformations we provide ad-hoc probability distributions, exploiting domain knowledge over the instances of the corresponding metamodel.

A generation configuration may come also with a seed that makes the generation deterministic and reproducible. For each one of the built-in generation configurations we provide a seed, producing the exact set of models we used during our experimentation.

### *3.4.3 Queries and transformations*

*Ant to Maven.* Ant [4] is an open source build tool (a tool dedicated to the assembly of the different pieces of a program) from the Apache Software Foundation. Ant is the most commonly used build tool for Java programs. Maven [5] is another build tool created by the Apache Software Foundation. It is an extension of Ant because ant Tasks can be used in Maven. The difference from Ant is that a project can be reusable. This transformation [22] generates a file for the build tool Maven starting from a file corresponding to the build tool Ant.

*CPL2SPL.* CPL (Call Processing Language) is a standard scripting language for the SIP (Session Initiation Protocol) protocol. It offers a limited set of language constructs. CPL is supposed to be simple enough so that it is safe to execute untrusted scripts on public servers [30]. SPL programs are used to control telephony agents (e.g. clients, proxies) implementing the SIP (Session Initiation Protocol) protocol. Whereas, the CPL has an XML-based syntax, the *CPL2SPL* transformation [24], provides an implementation of CPL semantics by translating CPL concepts into their SPL equivalent concepts.

*Graphcet2PetriNet.* This transformation[25] establishes a bridge between grafcet [17], and petri nets [43]. It provides an overview of the whole transformation sequence that enables to produce an XML petri net representation from a textual definition of a grafcet in a PNML format, and the other way around.

*IEEE1471 to MoDAF.* This transformation example [3] realizes the transformation between IEEE1471-2000 [34] and MoDAF-AV [11]. The IEEE1471 committee prescribes a recommended practice for the design and the analysis of Architecture of Software Intensive Systems. It fixes a terminology for System, Architecture, Architectural Description, Stakeholder, Concerns, View ,and Viewpoints concepts. MoDAF (Ministry of Defense Architecture Framework) is based on the DoDAF (Department of Defense Architecture Framework). DoDAF is a framework to design C4ISR systems. MoDAF-AV (Architecture View) used several concepts defined in the IEEE1471.

*Make2Ant.* Make (the most common build tool) is based on a particular shell or command interface and is therefore limited to the type of operating systems that use that shell. Ant uses Java classes rather than shell-based commands. Developers use XML to describe the modules in their program build. This benchmark [26] describes a transformation from a Makefile to an Ant file.

*MOF2UML.* The MOF (Meta Object Facility)[45] is an OMG standard enabling the definition of metamodels through common semantics. The UML (Unified Modeling Language) Core standard is the OMG common modeling language. Although, MOF is primarily designed for metamodel definitions and UML Core for the design of models, the two standards define very close notions. This example [27] describes a transformation enabling to pass from the MOF to the UML semantics. The transformation is based on the UML Profile for MOF OMG specification.

*OCL2R2ML.* The OCL to R2ML transformation scenario [41] describes a transformation from OCL (Object Constraint Language) [46] metamodel (with EMOF metamodel) into a R2ML (REWERSE I1 Rule Markup Language) metamodel. The Object Constraint Language (OCL) is a language that enables one to describe expressions and constraints on object-oriented (UML and MOF) models and other object modeling artifacts. An expression is an indication or specification of a value. A constraint is a restriction on one or more values of (part of) an object-oriented model or system. REWERSE I1 Rule Markup Language (R2ML) is a general web rule markup language, which can represent different rule types: integrity, reaction, derivation and production. It is used as pivotal metamodel to enable sharing rules between different rule languages, in this case with the OCL.

*UML2OWL.* This scenario [42] presents an implementation of the OMG's ODM specification. This transformation is used to produce an OWL ontology, and its *OWL Individuals* from an UML Model, and its *UML Instances*.

*BibTeXML to DocBook.* The *BibTeXML to DocBook* example [23] describes a transformation of a BibTeXML [47] model to a DocBook [56] composed document. BibTeXML is an XML-based format for the BibTeX bibliographic tool. DocBook, as for it, is an XML-based format for document composition.

*DSL to EMF.* This example [6] provides a complete overview of a transformation chain example between two technical spaces: Microsoft DSL Tools [40] and EMF. The aim of this example is to demonstrate the possibility to exchange models defined under different technologies. In particular, the described bridges demonstrate that it should be possible to define metamodels and models using both Microsoft DSL Tools and Eclipse EMF technologies. The bridge between MS/DSL and EMF spans two levels: the metamodel and model levels. At the level of metamodels, it allows to transform MS/DSL domain models to EMF metamodels. At the level of models, the bridge allows transforming MS/DSL models conforming to domain models to EMF models conforming to EMF metamodels. At both levels, the bridge operates in both directions. A chain of ATL-based transformations is used to implement the bridge at these two levels. The benefit of using such a bridge is the ability to transpose MS/DSL work in EMF platform, and inversely.

# 4. THE MDE BENCHMARK REPOSITORY

The *MDE Benchmark repository* is the central storage area where the artifacts of the benchmarks are archived for public access. These artifacts, mainly text files, comprise large models and metamodels – typically represented in their XMI serialization – and model transformations. To increase the visibility of these files we have chosen to make them publicly available through the OpenSourceProjects.eu (OSP) platform. The OSP platform is a software forge dedicated to hosting Open Source projects created within EU research projects.

The OSP platform provides, among other tools, a Git revision control system (RCS). Git repositories hosted in the OSP platform can be easily navigated by a Web interface.

## 4.1 Benchmark structure

The *MDE Benchmark repository* is located at [48]. Inside this repository every top level resource corresponds to a git submodule, each, representing a different case study held in a separate git repository.

Related resources for benchmarking a specific feature of a transformation engine are grouped in *projects*. A *project* is a self-contained entity, and can be considered as the basic benchmarking unit. *Projects* share a common internal structure that includes a short case description and a set of (optional) folders:

**Short case description** — A mandatory human-readable file describes the details of the test case, the file and directory structure, and any other important information (e. g., test cases can evolve and additional information not considered at the point of writing this document may be needed for executing the benchmark).

**Documentation** — This directory stores the documentation about the test case. The documentation of a test case may include, among other information, a detailed description of the test case, the foundations of the feature under testing, the building and execution instructions, etc.

**Queries and Transformations** — This directory stores the queries and transformations, in source code form, that stress the feature under testing.

**Models** — This directory contains the model and metamodel descriptions involved in the test transformation(s).

**Input data** — This directory contains the input data to be used by the test case(s).

**Expected data** — In this directory we store the files that contain the expected values that must be returned by the transformation. The expected data are compared with the actual output of the transformation to determine if the test execution has been successful or not.

**Source code** — In some situations, test cases may require additional code (such as Java code) to be executed. For example, test cases may be automatically launched with the help of third party libraries (such as JUnit), or test cases may execute external code following a black-box scheme. In this situations the additional code should be placed inside the `/src` directory.

**Libraries** — This directory is used to store any additional third party library (usually a binary file) required by the test case.

**Scripts** — Build and execution scripts should be placed under the `/build` directory. Examples of such scripts are Ant files [4], Maven files [5], Makefiles [20], bash shell scripts [19].

## 4.2 Submission guidelines

In order to increase the quality and soundness of the test cases available in the *MDE Benchmark repository*, we plan to keep it open to further submissions from the MDE community.

We have defined a simple set of guidelines that must be followed when contributing a new case study to guarantee that the quality of the repository is maintained. Specifically:

- New contributions must include a comprehensive description of the case study. A rationale for its inclusion must be provided, specially focusing on the differential aspects of the proposed case study, compared to the already included benchmarks.
- The sources, models, documentation and utility scripts must be organized as described in Section 4.1.
- Contributions must be sent to the address `mondo_team@opengroup.org` for their evaluation and approval.

# 5. CONCLUSION

This paper introduces the first open-set benchmark gathered from real-world cases to stress scalability issues in model transformation and query engines. These benchmark suite comes not only with the aim of providing a point of reference against which industrials and researchers might compare between different technologies to choose what could suit their needs, but also to motivate the MDE community to be part of its extension and contribute with additional cases not covered by this set.

In our future work we plan to furnish a feature-based organization of the benchmark in order to ease its use and enable efficient profit. We also intend to provide theoretical background on how to measure transformations scalability. Another point would be to optimize model instances generation to allow the generation of bigger models, also to contribute to the repository with a live/real-time instantiators for the consideration of infinite model transformations.

# 6. ACKNOWLEDGMENTS

This work is partially supported by the MONDO (EU ICT-611125) project. The authors would like to thank UNINOVA and Soft-Maint for their inputs, materials, and valuable discussions.

# 7. REFERENCES

[1] The train benchmark website. `https://incquery.net/publications/trainbenchmark/full-results`, 2013.

[2] The R project for statistical computing. `http://www.r-project.org/`, 2014.

[3] Albin Jossic. ATL Transformation Example: IEEE1471 to MoDAF, 2005. URL: `http://www.eclipse.org/atl/atlTransformations/IEEE1471_2_MoDAF/IEEE1471_2_MoDAF.doc`.

[4] Apache. Apache ant, 2014. URL: `http://ant.apache.org/`.

[5] Apache. Apache maven project, 2014. URL: `http://maven.apache.org/`.

[6] ATLAS group – LINA & INRIA. The Microsoft DSL to EMF ATL transformation , 2005. URL: `http://www.eclipse.org/atl/atlTransformations/DSL2EMF/ExampleDSL2EMF%5Bv00.01%5D.pdf`.

[7] P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial contextŮmotorola case study. In *Model Driven Engineering Languages and Systems*, pages 476–491. Springer, 2005.

[8] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In *Graph Transformations*, pages 396–410. Springer, 2008.

[9] G. Bergmann, D. Horváth, and Á. Horváth. Applying incremental graph transformation to existing models in relational databases. In *Graph Transformations*, pages 371–385. Springer, 2012.

[10] G. Bergmann, I. Ráth, and D. Varró. Parallelization of graph transformation based on incremental pattern matching. *Electronic Communications of the EASST*, 18, 2009.

[11] B. Biggs. Ministry of defence architectural framework (modaf). 2005.

[12] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *International Journal on Semantic Web & Information Systems*, 5(2):1–24, 2009.

[13] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014.

[14] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. On the concurrent execution of model transformations with linda. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 3. ACM, 2013.

[15] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Trans. Database Syst.*, 17(1):1–31, Mar. 1992.

[16] C. Clasen, M. D. Del Fabro, and M. Tisi. Transforming very large models in the cloud: a research roadmap. In *First International Workshop on Model-Driven Engineering on and for the Cloud*, 2012.

[17] R. David. Grafcet: A powerful tool for specification of logic controllers. *Control Systems Technology, IEEE Transactions on*, 3(3):253–268, 1995.

[18] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8(1):21–43, 2009.

[19] GNU. Bourne-Again SHell manual, 2014. URL: `http://www.gnu.org/software/bash/manual/`.

[20] GNU. GNU 'make', 2014. URL: `http://www.gnu.org/software/make/manual/`.

[21] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480. ACM, 2011.

[22] INRIA. ATL Transformation Example: Ant to Maven, 2005. URL: `http://www.eclipse.org/atl/atlTransformations/Ant2Maven/ExampleAnt2Maven%5Bv00.01%5D.pdf`.

[23] INRIA. ATL Transformation Example: BibTeXML to DocBook, 2005. URL: `http://www.eclipse.org/atl/atlTransformations/BibTeXML2DocBook/ExampleBibTeXML2DocBook%5Bv00.01%5D.pdf`.

[24] INRIA. ATL Transformation Example: CPL to SPL, 2005. URL: `http://www.eclipse.org/atl/atlTransformations/CPL2SPL/README.txt`.

[25] INRIA. ATL Transformation Example: Grafcet to Petri Net, 2005. URL: `http://www.eclipse.org/atl/atlTransformations/Grafcet2PetriNet/`

ExampleGrafcet2PetriNet[v00.01].pdf.

[26] INRIA. ATL Transformation Example: Make to Ant, 2005. URL: http://www.eclipse.org/atl/atlTransformations/Make2Ant/ExampleMake2Ant[v00.01].pdf.

[27] INRIA. ATL Transformation Example: MOF to UML, 2005. URL: http://www.eclipse.org/atl/atlTransformations/MOF2UML/ExampleMOF2UML[v00.01].pdf.

[28] Inria. Atl transformation zoo, 2014. URL: http://www.eclipse.org/atl/atlTransformations/.

[29] B. Izsó, Z. Szatmári, G. Bergmann, Á. Horváth, and I. Ráth. Towards precise metrics for predicting graph query performance. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 412–431, Silicon Valley, CA, USA, 11/2013 2013. IEEE.

[30] F. Jouault, J. Bézivin, C. Consel, I. Kurtev, F. Latry, et al. Building dsls with amma/atl, a case study on spl and cpl telephony languages. In *ECOOP Workshop on Domain-Specific Program Development*, 2006.

[31] F. Jouault, J. Sottet, et al. An amma/atl solution for the grabats 2009 reverse engineering case study. In *5th International Workshop on Graph-Based Tools, Zurich, Switzerland*, 2009.

[32] F. Jouault, J.-S. Sottet, et al. An amma/atl solution for the grabats 2009 reverse engineering case study. In *5th International Workshop on Graph-Based Tools, Grabats*, 2009.

[33] F. Jouault and M. Tisi. Towards incremental execution of atl transformations. In *Theory and Practice of Model Transformations*, pages 123–137. Springer, 2010.

[34] E. Jouenne and V. Normand. Tailoring ieee 1471 for mde support. In *UML Modeling Languages and Applications*, pages 163–174. Springer, 2005.

[35] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *Theory and practice of model transformations*, pages 46–60. Springer, 2008.

[36] D. S. Kolovos, R. F. Paige, and F. A. Polack. The grand challenge of scalability for model driven engineering. In *Models in Software Engineering*, pages 48–53. Springer, 2009.

[37] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 2. ACM, 2013.

[38] G. Mezei, T. Levendovszky, T. Mészáros, and I. Madari. Towards truly parallel model transformations: A distributed pattern matching approach. In *EUROCON 2009, EUROCON'09. IEEE*, pages 403–410. IEEE, 2009.

[39] MIA-Software. Modiscojava metamodel (knowledge discovery metamodel) version 1.3, 2012. URL: http://dev.eclipse.org/svnroot/modeling/org.eclipse.mdt.modisco/main/branches/0_11/org.eclipse.gmt.modisco.java/model/java.ecore.

[40] Microsoft Corp. The DSL tools, 2014. URL: http://msdn.microsoft.com/vstudio/DSLTools/.

[41] Milan Milanovic. ATL Transformation Example: OCL to R2ML, 2005. URL: http://www.eclipse.org/atl/atlTransformations/OCL2R2ML/README.txt.

[42] Milan Milanovic. ATL Transformation Example: UML to OWL, 2005. URL: http://www.eclipse.org/atl/atlTransformations/UML2OWL/README.txt.

[43] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[44] OMG (Object Management Group). Kdm (knowledge discovery metamodel) version 1.3, 2011. URL: http://www.omg.org/spec/KDM/1.3/.

[45] OMG (Object Management Group). Mof (meta object facility) version 2.4, 2011. URL: http://www.omg.org/spec/MOF/2.4.

[46] OMG (Object Management Group). Ocl (object constraint language) v2.0, 2011. URL: http://www.omg.org/spec/OCL/2.0/PDF.

[47] L. Previtali, B. Lurati, and E. Wilde. Bibtexml: An xml representation of bibtex. In V. Y. Shen, N. Saito, M. R. Lyu, and M. E. Zurko, editors, *WWW Posters*, 2001.

[48] M. Project. Transformation benchmarks, 2014. URL: http://opensourceprojects.eu/p/mondo/d31-transformation-benchmarks/.

[49] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 974–985. VLDB Endowment, 2002.

[50] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL performance benchmark. In *Proc. of the 25th International Conference on Data Engineering*, pages 222–233, Shanghai, China, 2009. IEEE.

[51] M. Tisi, S. Martinez, and H. Choura. Parallel execution of atl transformation rules. In *Model-Driven Engineering Languages and Systems*, pages 656–672. Springer, 2013.

[52] Transaction Processing Performance Council (TPC). TPC-C Benchmark, 2010.

[53] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 2014.

[54] M. Van Amstel, S. Bosems, I. Kurtev, and L. F. Pires. Performance in model transformations: experiments with atl and qvt. In *Theory and Practice of Model Transformations*, pages 198–212. Springer, 2011.

[55] G. Varro, A. Schurr, and D. Varro. Benchmarking for graph transformation. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 79–88. IEEE, 2005.

[56] N. Walsh and R. Hamilton. *DocBook 5: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2010.

**Table 3: List of Java code quality fixes**

| Rule | Documentation |
|---|---|
| ConstantName | http://checkstyle.sourceforge.net/config_naming.html#ConstantName |
| LocalFinalVariableName | http://checkstyle.sourceforge.net/config_naming.html#LocalFinalVariableName |
| LocalVariableName | http://checkstyle.sourceforge.net/config_naming.html#LocalVariableName |
| MemberName | http://checkstyle.sourceforge.net/config_naming.html#MemberName |
| MethodName | http://checkstyle.sourceforge.net/config_naming.html#MethodName |
| PackageName | http://checkstyle.sourceforge.net/config_naming.html#PackageName |
| ParameterName | http://checkstyle.sourceforge.net/config_naming.html#ParameterName |
| StaticVariableName | http://checkstyle.sourceforge.net/config_naming.html#StaticVariableName |
| TypeName | http://checkstyle.sourceforge.net/config_naming.html#TypeName |
| AvoidStarImport | http://checkstyle.sourceforge.net/config_imports.html#AvoidStarImport |
| UnusedImports | http://checkstyle.sourceforge.net/config_imports.html#UnusedImports |
| RedundantImport | http://checkstyle.sourceforge.net/config_imports.html#RedundantImport |
| ParameterNumber | http://checkstyle.sourceforge.net/config_sizes.html#ParameterNumber |
| ModifierOrder | http://checkstyle.sourceforge.net/config_modifier.html#ModifierOrder |
| RedundantModifier | http://checkstyle.sourceforge.net/config_modifier.html#RedundantModifier |
| AvoidInlineConditionals | http://checkstyle.sourceforge.net/config_coding.html#AvoidInlineConditionals |
| EqualsHashCode | http://checkstyle.sourceforge.net/config_coding.html#EqualsHashCode |
| HiddenField | http://checkstyle.sourceforge.net/config_coding.html#HiddenField |
| MissingSwitchDefault | http://checkstyle.sourceforge.net/config_coding.html#MissingSwitchDefault |
| RedundantThrows | http://checkstyle.sourceforge.net/config_coding.html#RedundantThrows |
| SimplifyBooleanExpression | http://checkstyle.sourceforge.net/config_coding.html#SimplifyBooleanExpression |
| SimplifyBooleanReturn | http://checkstyle.sourceforge.net/config_coding.html#SimplifyBooleanReturn |
| FinalClass | http://checkstyle.sourceforge.net/config_design.html#FinalClass |
| InterfaceIsType | http://checkstyle.sourceforge.net/config_design.html#InterfaceIsType |
| VisibilityModifier | http://checkstyle.sourceforge.net/config_design.html#VisibilityModifier |
| FinalParameters | http://checkstyle.sourceforge.net/config_misc.html#FinalParameters |
| LooseCoupling | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/typeresolution.html#LooseCoupling |
| SignatureDeclareThrowsException | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/typeresolution.html#SignatureDeclareThrowsException |
| DefaultLabelNotLastInSwitchStmt | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/design.html#DefaultLabelNotLastInSwitchStmt |
| EqualsNull | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/design.html#EqualsNull |
| CompareObjectsWithEquals | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/design.html#CompareObjectsWithEquals |
| PositionLiteralsFirstInComparisons | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/design.html#PositionLiteralsFirstInComparisons |
| UseEqualsToCompareStrings | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/strings.html#UseEqualsToCompareStrings |
| IntegerInstantiation | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#IntegerInstantiation |
| ByteInstantiation | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#ByteInstantiation |
| ShortInstantiation | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#ShortInstantiation |
| LongInstantiation | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#LongInstantiation |
| BooleanInstantiation | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/migrating.html#BooleanInstantiation |
| SimplifyStartsWith | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/optimizations.html#SimplifyStartsWith |
| UnnecessaryReturn | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/unnecessary.html#UnnecessaryReturn |
| UnconditionalIfStatement | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/basic.html#UnconditionalIfStatement |
| UnnecessaryFinalModifier | http://pmd.sourceforge.net/pmd-5.1.0/rules/java/unnecessary.html#UnnecessaryFinalModifier |

# LinTraP: Primitive Operators for the Execution of Model Transformations with LinTra

Loli Burgueño
Universidad de Málaga
Malaga, Spain
loli@lcc.uma.es

Eugene Syriani
University of Alabama
Tuscaloosa AL, USA
esyriani@cs.ua.edu

Manuel Wimmer
Vienna University of Technology
Vienna, Austria
wimmer@big.tuwien.ac.at

Jeff Gray
University of Alabama
Tuscaloosa AL, USA
gray@cs.ua.edu

Antonio Vallecillo
Universidad de Málaga
Malaga, Spain
av@lcc.uma.es

## ABSTRACT

The problems addressed by Model-Driven Engineering (MDE) approaches are increasingly complex, hence performance and scalability of model transformations are gaining importance. In previous work, we introduced LinTra, which is a platform for executing out-place model transformations in parallel. The parallel execution of LinTra is based on the Linda coordination language, where high-level model transformation languages (MTLs) are compiled to LinTra and eventually executed through Linda. In order to define the compilation modularly, this paper presents a minimal, yet sufficient, collection of primitive operators that can be composed to (re-)construct any out-place, unidirectional MTL.

## Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.1.3 [**Programming Techniques**]: Concurrent Programming; C.4 [**Computer Systems Organization**]: Performance of systems

## Keywords

Model Transformation, Linda, LinTra

## 1. INTRODUCTION

Model-Driven Engineering [2] is a relatively new paradigm that has grown in popularity in the last decade. Although there is a wide variety of approaches and languages with different characteristics and oriented to different types of model transformations (MT), most model transformation engines are based on sequential and local execution strategies. Thus, they have limited capabilities to transform very large models (with thousands or millions of elements), and provide even less capabilities to perform the transformation in a reasonable amount of time.



In previous works [3, 4], we investigated concurrency and distribution for out-place transformations to increase their performance and scalability. Our approach, LinTra, is based on Linda [8], a mature coordination language for parallel processes that supports reading and writing data in parallel into distributed tuple spaces. A tuple space follows the Blackboard architecture [5], which makes the data distributed among different machines transparent to the user.

To execute transformations on the LinTra architecture, LinTra specifies how to represent models and metamodels, how the trace links between the elements in the input model and the elements created from them are encoded for efficient retrieval, which agents are involved in the execution of the MTs and their role, how the MTs are executed in parallel, and how the models are distributed over the set of machines composing the cluster where each MT is executed.

The implementation of several case studies using the Java implementation of LinTra (jLinTra) is available on our website[1], together with the performance comparison with several well-known model transformation languages (MTLs) such as ATL [11], QVT-O [14] and RubyTL [7].

In order to hide the underlying LinTra architecture and in order to ease the compilation from any existing out-place MTL to the LinTra engine, in this paper we propose a collection of minimal, yet sufficient, primitive operators that can be composed to (re-)construct any out-place and unidirectional MTL. These primitive operators encapsulate the LinTra implementation code that makes the parallel and distributed execution possible, serving as an abstraction of the implementation details of the general-purpose language in which LinTra is implemented.

The rest of the paper is structured as follows. Section 2 introduces the collection of primitives. Section 3 illustrates examples of primitive combinations in order to write MTs. Section 4 discusses the related work to our approach. Finally, Section 5 presents our conclusions and an outlook on future work.

[1] `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/MTBenchmark`

## 2. COLLECTION OF PRIMITIVES

This section shortly introduces LinTra, presents the set of primitive operators, and describes the mapping of the primitive operators to LinTra.

### 2.1 Background on LinTra

LinTra uses the Blackboard paradigm [5] to store the input and output models, as well as the required data to keep track of the MT execution that coordinates the agents that are involved in the process.

One of the keys of our approach is the model and metamodel representation. In this representation, we assume that every entity in the model is independent from another. Each entity is assigned an identifier that is used for representing relationships between entities and by the trace model. Relationships between entities are represented by storing in the source entity the identifier(s) of its target entity(ies).

Traceability is frequently needed when executing an out-place model transformation because the creation of an element might require information about some other elements previously transformed, or even information about elements that will be transformed in the future. This means that there might be dependencies that can affect the execution performance, *e.g.*, when one element needs access to an element that has not been created yet. In LinTra, traceability is implemented implicitly using a bidirectional function that receives as a parameter the entity identifier (or all the entity identifiers in the case that the match comprises more than one entity) of the input model and returns the identifier of the output entity(ies), regardless whether the output entities have already been created or not. This means that LinTra does not store information about the traces explicitly; thus, the performance is not affected by the access to memory and the search for trace information.

Together with the Blackboard, LinTra uses the Master-Slave design pattern [5] to execute MTs. The master's job is to launch slaves and coordinate their work. Slaves are in charge of applying the transformation in parallel to submodels of the input model (i.e., partitions) as if each partition is a complete and independent model. Since LinTra only deals with out-place transformations, the complete input model is always available. Thus, if the slaves have data dependencies with elements that are not in the submodels they were assigned, they only have to query the Blackboard to retrieve them.

### 2.2 Primitives

Two different kinds of primitives can be distinguished in LinTra: the primitive constructs to encapsulate the concurrent execution platform and the primitive constructs needed by the MTL.

**Primitives for the Concurrent Platform.** Despite the fact that due the representation of models in LinTra, all model elements are independent from each other, LinTra requires the user to specify the size of every partition, i.e., how many elements belong to each one. Furthermore, although there is no need of specifying how the elements are partitioned or which elements belong to the same partition, LinTra offers that possibility.

The `PartitionCreator` primitive receives the input model, an OCL expression, *OE*, and the maximum number of model entities, *S*, that each partition will contain. The `PartitionCreator` queries the input model using *OE* and partitions the resulting submodel into partitions of size *S*. The combination of `PartitionCreators` with different OCL expressions may lead to overlapping partitions; thus, the LinTra engine checks internally that the intersection of all the partitions is empty and the union is the whole model. The purpose of OE is to give the user the possibility to optimize the MT execution.

**Primitives for the Model Transformation Language.** The minimum set of primitive constructs needed to define out-place model transformations are: `Composer`, `Tracer`, `Creator`, `CondChecker`, `Finder`, `Declarer` and `Assigner`.

`Composer` is a primitive that allows the grouping of a combination of primitives and assigns the combination a name. Its syntax is *Composer <composerName>* { *<combination of primitives>* } and it is mainly used by the `Tracer`.

The `Tracer` provides access to the trace model needed by out-place MT engines for linking the entities in the output model. Given an input entity or set of entities that match the pre-condition of a rule, the traces give access to the entities that were created in the post-condition, and vice versa. In this case, to identify which primitive belongs to which rule, we propose to encapsulate them in a `Composer` so that the `Tracer` receives as a parameter the name of the Composer and the set of entities from the pre or post-condition and gives the reference to the other entities. Its signature is *Tracer(composer : Composer, e : Entity) : Collection(Entity)* and *Tracer(composer : Composer, e : Collection(Entity)) : Collection(Entity)*. The *Collection* corresponds to the four collection types in OCL: Set, OrderedSet, Bag, and Sequence. Furthermore, in a Composer, more than one element might be created; thus, in the `Tracer`, the concrete `Creator` might need to be specified given its name, being its syntax *Tracer(composer : Composer, e : Collection(Entity), creatorName : String) : Collection(Entity)*.

`Creator` creates an entity given its data type and its features (attributes and bindings) and writes it in the Blackboard. The primitive receives as parameter the entity type and a dictionary which stores the values of every feature. The dictionary is a collection of *key-value* pairs where the first element is the name of the feature and the second its value. The type of the values received by the dictionary are of two kinds: OCL primitive data types, which correspond to the basic data types of the language (string, boolean and numbers in their different formats), and the types defined by all the classes given by the output metamodel. Furthermore, the values can be an OCL collection of the previous types. Its syntax is *Creator(type : Factory, features : Dictionary<feature : String, value: OCLDataType | Entity>)*. Moreover, the Creator might have an optional parameter of type *String* specifying its name, *Creator(type : Factory, features : Dictionary<feature : String, value: [OCLDataType | Entity]>, name : String)*. This is needed in case that it is referenced by a *Tracer*.

`CondChecker` allows the querying of the input model in the

Blackboard with an OCL expression that evaluates to a boolean value. It receives as input the OCL expression, queries the Blackboard and returns the result. Its signature is *CondChecker(expr : OCLExpression) : Boolean*.

`Finder` allows the retrieval of elements from the Blackboard that satisfy a constraint. It receives as a parameter an OCL expression and returns the set of entities (submodel) that fulfils the OCL expression. Its signature is *Finder(expr : OCLExpression) : Collection(Entity)*.

`Declarer` allows to create a global variable that can be accessed by its name from any other primitive and that is accessed by all the Slaves involved in the transformation process. Its syntax is *Declarer(type : [OCLDataType | Entity], name : String)*. The value of the variable is set by `Assigner`.

`Assigner` sets the value of a variable defined by `Declarer`. `Assigner` receives as a parameter the name of the variable and its value. Its syntax is *Assigner(varName : String, value : [OCLDataType | Entity | Creator])*. In the case that the second parameter is a `Creator`, the element is stored in the Blackboard and the variable points to it. In case the variable is stored in the Blackboard, every time it is updated, the corresponding value in the Blackboard is overwritten. If the second parameter is an OCL primitive data type or an entity, the variable is stored in memory and accessed while the MT is executed but it is not a persistent value in the Blackboard.

Figure 1 shows a class diagram with all the primitives and their relationships.

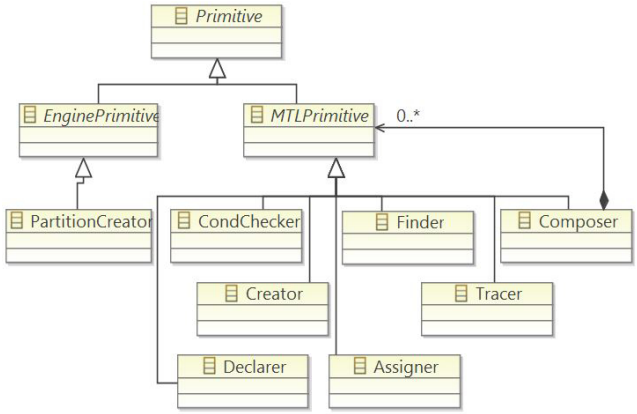



**Figure 1: Primitives Class Diagram.**

## 2.3 Integrating the Primitives with the LinTra Engine

When executing a transformation with LinTra there are several steps. Some of the steps are done automatically by the engine and others require that the user gives certain guidelines on how to proceed by means of the primitives. Two different phases can be distinguished: the setup and the MT execution itself.

The semantics of some MTs might require that a certain set of rules are applied to the whole input model before applying or after having applied some others. This is the case, for example, of top rules in QVT-R [14], and entrypoint and endpoint rules in ATL [11]. In order to be able to express this behaviour, in the setup phase, the rule schedule must be extracted from the transformation given by the user and a collection of rules (or rule layers) must be created. All the rules belonging to the same layer can be executed in parallel, but all rules in one layer must have terminated before rules in a subsequent layer can begin.

Furthermore, during the setup, the transformation written in a high-level MTL is compiled to the MTL primitives, and the input model is parsed to the tuple space representation and stored into the Blackboard. Then, the `PartitionCreator` provided by the user is executed and the model partitions are created. Finally, the tasks to be executed by the slaves are created and stored in order in the Blackboard. A task is a pair consisting of a rule layer and a model partition. The tasks are produced by computing all the possible combinations between the partitions and the rule layers.

After the setup phase is finished, the LinTra MT engine starts using the Master-Slave design pattern. The master creates slaves that execute the tasks that share the same rule layer and waits for all the tasks to be finished before starting to execute the ones that involve the following layer. Every slave executes the assigned task sequentially and all the slaves work in parallel. The master behaviour after launching the slaves is given by the pseudo-code presented in Listing 1.

**Listing 1: Master.**

```
1  params:: Integer : nSlaves
2  index := 1
3  slavePool := createSlaves(nSlaves)
4  task := Blackboard.Tasks.dequeue()
5  while ( task != null ){
6    while (task != null
7      and task.ruleLayer.index = index)
8      slave := slavePool.getIdleSlave() -- blocking
9      slave.execute(task)
10     task := Blackboard.Tasks.dequeue()
11   }
12   join() -- wait for all the slaves to finish
13          -- before starting to transform the
14          -- tasks involving the next ruleLayer
15   index := index + 1
16 }
```

When a slave receives a task, it transforms the submodel given by its partition with the rules given by its rule layer. These rules are a collection of MT primitives. The code executed by the slaves is shown in Listing 2. An overview of how the system works can be seen in the activity diagram presented in Figure 2.

**Listing 2: Slave - execute method**

```
17 for each e ∈ task.partition {
18     task.ruleLayer.transforms(e)
19 }
```

The sequential execution of a MT is a concrete scenario in LinTra. There are several ways to achieve it. The MT is executed sequentially either by not partitioning the input model (therefore, only one task is created and executed sequentially by a single slave) or by launching only one slave

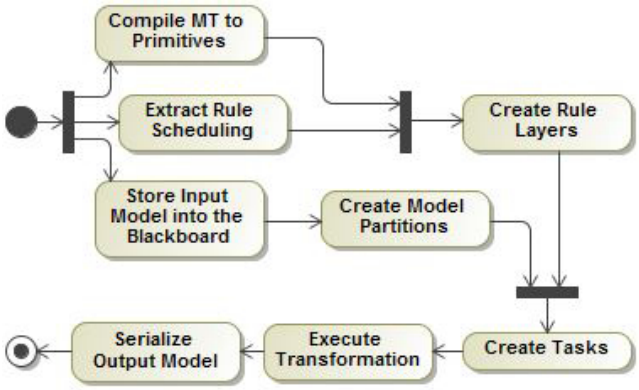



**Figure 2: Activity Diagram of the Transformation Process.**

that transforms all the tasks.

A class diagram showing all the elements involved in LinTra and how they are related to each other can be found in Figure 3. It contains the Master and Slave where every slave executes a Transformation which is a collection of MT Primitives that accesses to a Blackboard which is composed by Areas that contain both Tasks - formed by a Rule Layer and Partitions - and and the Entities that belong to a certain Model. *MTLPrimitive* in this diagram corresponds with the root class in the diagram presented in Figure 1.

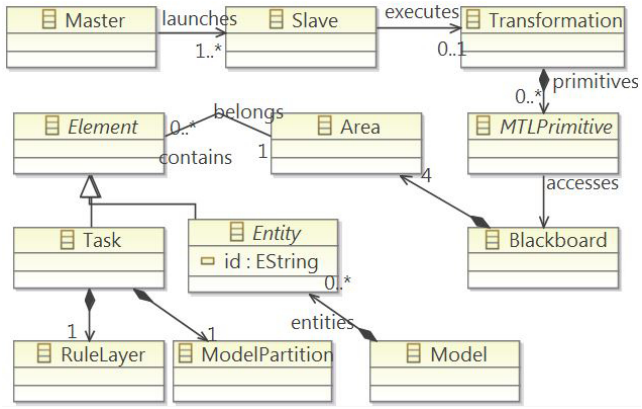



**Figure 3: LinTra Class Diagram Metamodel.**

## 3. EXAMPLES

This section demonstrates how the introduced primitives are used for concrete transformation examples.

### 3.1 Activity Diagram to Petri Net

This case study is a simplification of the transformation from UML Activity Diagrams to Petri Nets described in [15]. The metamodels are represented in Figures 5 and 6 and, for simplicity, only contain the elements needed by our simplified transformation.

The MT simplification consists of an unaltered subset of the original MT which focuses on transforming only several elements belonging to the input model instead of the

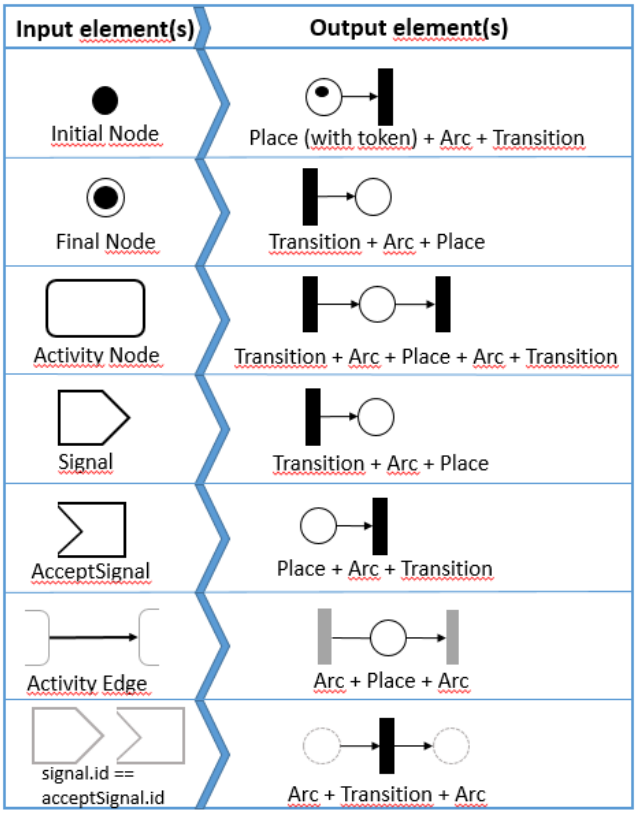



**Figure 4: Activity Diagram to Petri Net Transformation.**

whole model. Every *Initial Node* is transformed to a processing *Place* with one token, an *Arc* pointing to a *Transition* and other *Transition*. Every *Final Node* is transformed to a *Transition*, an *Arc* pointing to a *Place* and such *Place*. From every *Action Node*, an entry *Transition*, an *Arc* pointing to a *Place*, such *Place*, an *Arc* from it to another *Transition*, and such *Transition* are created. Every *Signal* is transformed in the same way as a *Final Node* and every *Accept Signal* as an *Initial Node* but with no token. *Activity Edges* between any kind of nodes are transformed as an *Arc* pointing to a *Place*, the *Place* and another *Arc* coming from it. Every pair *Signal-Accept Signal* with the same value for their feature *signalId* are transformed in the same way as *Activity Edges*. For a better understandability, the previous transformation rules are represented graphically in Figure 4. Finally, only one entity of *PetriNet* is created in the output model whose name is the String "PNet" concatenated with the number of arcs, the number of places and the number of transactions in the output model after the whole transformation process. All places, arcs and transitions must be linked to that *PetriNet* entity.

Let us assume that the user does not specify how the entities are assigned to the different partitions and the partition size is 100. The partition creator is invoked as *PartitionCreator(inModel, Entity.allInstances, 100)*. Let us suppose that it returns three partitions, $P = \{p1, p2, p3\}$. From the MT, the rule schedule is extracted and the rule layers are created. Given the MT definition, three different rule lay-

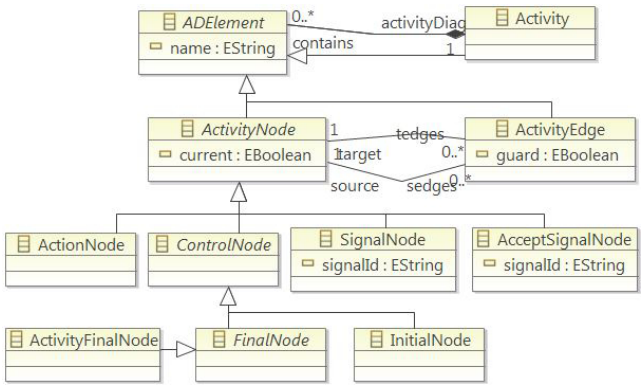



**Figure 5: Activity Diagram Metamodel.**

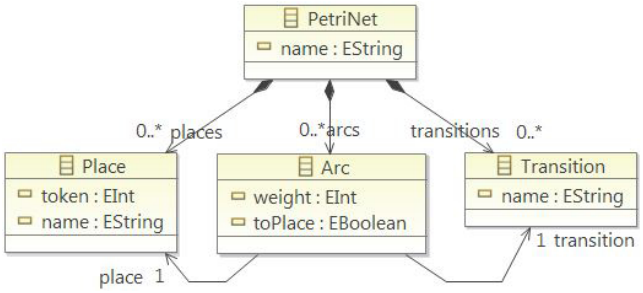



**Figure 6: Petri Net Metamodel.**

ers are created: $RL = [l1, l2, l3]$ where *l1* contains the first composer where a global variable for the unique Petri Net that will be referenced by the rest of entities is created, in *l2* all the elements are created and in *l3*, the name of the Petri Net is changed. Given the partitions and the layers, the tasks to be executed are $T = [T1, T2, T3]$, where $T1 = \{(p1, l1), (p2, l1), (p3, l1)\}$, $T2 = \{(p1, l2), (p2, l2), (p3, l2)\}$ and $T3 = \{(p1, l3), (p2, l3), (p3, l3)\}$. We make the distinction between *T1*, *T2* and *T3* to clarify that all tasks in *T1* are relative to *l1*, all tasks in *T2* are relative to *l2* and all tasks in *T3* are relative to *l3*; thus, until all tasks from *T1* have been executed, tasks from *T2* cannot start and until all tasks in *T2* have been executed, tasks from *T3* cannot start.

The compilation process from the high-level MT to the primitives produces the code shown in Listings 3, 4, and 5.

**Listing 3: MTL Primitives for the first rule layer (l1).**

```
1 Composer First {
2   Declarer (PetriNet, pNet)
3   Assigner(pNet,
4     Creator(PetriNet, {[name, 'PNet']}))
5 }
```

As the case study requires that only one *PetriNet* instance is created and the rest of the elements in the output model reference it, there is a need for a global variable that must be available before the rest of the rules are applied. Listing 3 declares in line 1 a composer which, encapsulates the declaration of a variable called `pNet` (line 2) and the creation of the *PetriNet* entity (lines 3 and 4). Note that, as the entity created is a persistent entity which is part of the output model (instead of a temporary variable), the second parameter of the assigner is a creator, which means that the value is stored in the Blackboard and the variable is a pointer to it.

Listing 4 shows part of the primitives that compose the second rule layer. In particular, this listing shows the collection of primitives to transform *ActionNodes* and *SignalNodes* and to match the output entities created from *SignalNodes* and *AcceptSignalNodes*.

Lines 2, 21 and 33 show the condition checkers which impose the pre-conditions that the entity, e, given by a task, has to fulfil to be transformed by the set of primitives inside the `if` the condition checker. For instance, given e, if the condition checker in line 2 is fulfilled, it means that e is of type *SignalNode* and from it, the entities specified by the creators in lines 3, 5, 10, 12 and 17 will be created. For example, in the creator in line 5, an `Arc` is created where *transition* points to the entity created by the creator called *t1*, *place* points to the entity created by creator `p`, *toPlace* is set to `true` and `net` points to the element given by the global variable `pNet`. The name of the creators is optional, and in this example, it is only given when it is needed by a tracer. For example, the tracer in line 6 gives the reference to the entity created from e in *ActNode* by a creator called *t1*.

A tracer can give the reference to an entity that has been created either in the same composer or in a different composer. It can also point either to a composer located in the same rule layer or in a different rule layer. An example of the first case is the tracer in line 39, which points to a creator in the composer *Signal*.

The last composer encompasses the entities created by every pair *Signal-Accept Signal* with the same *signalId*. This is a particular case where from every entity, e, received in the task and fulfilling the condition checker in line 33 (i.e. whose type is *SignalNode*), it is needed to find in the Blackboard all the elements of type *AcceptSignalNode* with the same signal identifier as e. This is achieved by using the Finder primitive in line 34.

**Listing 4: MTL Primitives for the second rule layer (l2).**

```
1  Composer ActNode {
2    if (CondChecker(e.oclIsTypeOf(ActionNode)))
3      Creator(Transition,
4        {[name, e.name], [net, pNet]}, 't1')
5      Creator(Arc,
6        {[transition,Tracer(ActNode,e, 't1')],
7        [place,Tracer(ActNode,e,'p')],
8        [toPlace, true],
9        [net, pNet]})
10     Creator(Place,
11       {[name,e.name],[net,pNet],[token,0]}, 'p')
12     Creator(Arc,
13       {[transition,Tracer(ActNode, e, 't2')],
14       [place,Tracer (ActNode, e, 'p')],
15       [toPlace, false],
16       [net, pNet]})
17     Creator(Transition,
18       {[name, e.name], [net, pNet]}, 't2')
19 }
```

```
20 Composer Signal {
21   if (CondChecker(e.oclIsTypeOf(SignalNode)))
22     Creator(Transition,
23       {[name, e.name], [net, pNet]}, 't')
24     Creator(Arc,
25       {[transition,Tracer(Signal,e,'t')],
26       [place, Tracer(Signal, e, 'p')],
27       [toPlace, true],
28       [net, pNet]})
29     Creator(Place,
30       {[name, e.name],[net, pNet]}, 'p')
31 }
32 Composer MatchSignals {
33   if(CondChecker(e.oclIsTypeOf(SignalNode)))
34     for(a in Finder(AcceptSignalNode.allInstances
35       ->select(as|e.activityDiag = as.activityDiag
36       and e.signalId = a.signalId)
37       Creator (Arc,
38         {[place,
39           Tracer(Signal, e, 'p')],
40         [transition,
41           Tracer(MatchSignals, {e, a}, 't')],
42         [toPlace, false ],
43         [net, pNet]})
44       Creator (Transition,
45         {[name, e.name+'-'+a.name],
46         [net, pNet]}, 't')
47       Creator (Arc,
48         {[place,
49           Tracer(AcceptSignal, e, 'p')],
50         [transition,
51           Tracer(MatchSignals, {e, a}, 't')],
52         [toPlace, true ],
53         [net, pNet]})
54 }
55 ...
```

Finally, once all the output entities have been created, the third rule layer, where the name of the only *PetriNet* is updated, can be executed. Listing 5 shows how it is done using an `Assigner` and a `Creator` inside of it that overwrites the value of the `pNet`.

**Listing 5: MTL Primitives for the third rule layer (l3).**

```
1 Composer Last {
2   Assigner (pNet,
3     Creator(PetriNet,
4       {[name, pNet.name+(pNet.arcs.size()
5         +pNet.places.size()
6         +pNet.transitions.size())]}))
7 }
```

The complete case study can be downloaded from our website[2]. Note that, although the case study in [15] is an out-place MT, i.e. the input and output metamodels are different and the input model is not modified, the authors have used an in-place MTL; thus, although the semantics of the MT is the same, our solution is different to theirs.

## 3.2 Filtering Families

In this subsection, we introduce a second case study where the input and output metamodel are the *Family* metamodel shown in Figure 7. The MT consists of filtering the input model so that the output metamodel is a subset of the input model that contains only the families which have exactly two daughters, two sons and their family members. This means that the members belonging to families with more than two daughters and two sons are not in the output model.

[2] `http://atenea.lcc.uma.es/index.php?title=Main_Page/Resources/Linda/ActivityDiag2PetriNet`

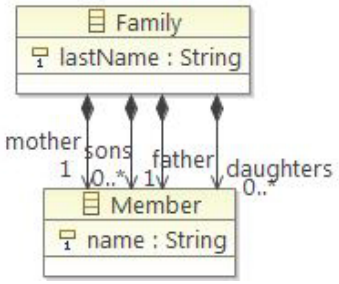



**Figure 7: Family Metamodel.**

For example, this behaviour is done in ATL using a particular kind of rule called a lazy rule. Lazy rule are not completely declarative, but they must be invoked explicitly. In this way, the transformation for this example has a main rule that checks if a family fulfilled the requirements and in that case, a lazy rule that transforms its members is called. Although in most of the cases there is a direct relation between rules in the high-level MTL and composers, this case is an exception. With our collection of primitives, this is done by means of a unique Composer.

Listing 6 shows the MTL primitives for this case study. An entity, e, fulfils the condition in line 2, in line 5 a *Family* is created. Then, the condition checkers in lines 11 and 15 and creators in lines 12 and 16 transform every mother and father of that family. All sons and daughters are transformed in lines 20 and 24. Tracers in lines 6 and 7 reference creators that can be invoked or not because they are inside `if`s, in the case that no entity is created, the reference points to null. Tracers in lines 8 and 9 point to entities created inside a `for`, those tracers return the pointers to all the elements created in that creator. The complete case study can be found on our website[3].

**Listing 6: MTL primitives for the Filtering Families case study.**

```
1 Composer R {
2   if(CondChecker(e.oclIsTypeOf(Family)
3     and e.daughters.size()=2
4     and e.sons.size()=2 ))
5     Creator(Family, {[lastName, e.lastName],
6       [father, Tracer(R, e, 'f')],
7       [mother, Tracer(R, e, 'm')],
8       [daughters, Tracer(R, e, 'ds')],
9       [sons, Tracer(R, e, 'ss')]},
10      'fam')
11    if(CondChecker(not e.father.isOclUndefined()))
12      Creator(Member,{[name,e.father.name],
13        [familyFather, Tracer(R, e, 'fam')]},
14        'f')
15    if(CondChecker(not e.mother.isOclUndefined()))
16      Creator(Member,{[name,e.mother.name],
17        [familyMother, Tracer(R, e, 'fam')]},
18        'm')
19    for(daughter in e.daughters)
20      Creator(Member,{[name,daughter.name],
21        [familyDaughter, Tracer(R, e, 'fam')]},
22        'ds')
23    for(son in e.sons)
24      Creator(Member,{[name,son.name],
25        [familySon, Tracer(R, e, 'fam')]},
26        'ss')
27 }
```

[3] `http://atenea.lcc.uma.es/index.php?title=Main_Page/Resources/Linda/FilteringFamilies`

## 4. RELATED WORK

With respect to the contribution of this paper, we first elaborate on related work considering the performance of model transformations in general and concerning parallel execution in particular and second we discuss how the work on primitives for model transformations is extended by this work.

The performance of model transformations is now considered as an integral research challenge in MDE [12]. For instance, Amstel et al. [18] considered the runtime performance of transformations written in ATL and in QVT. In [19], several implementation variants using ATL, e.g., using either imperative constructs or declarative constructs, of the same transformation scenario have been considered and their different runtime performance has been compared. However, these works only consider the traditional execution engines following a sequential rule application approach. One line of work we are aware of dealing with the parallel execution of ATL transformations is [6] where Clasen et al. outlined several research challenges when transforming models in the cloud. In particular, they discussed how to distribute transformations and elaborated on the possibility to use the Map/Reduce paradigm for implementing model transformations. A follow-up work on this is presented in Tisi et al. [17] where a parallel transformation engine for ATL is presented.

There is also some work in the field of graph transformations where multi-core platforms are used for the parallel execution of model transformation rules [1, 9] especially for the matching phase of the left-hand side of graph transformation rules. A recent work exploiting the Bulk Synchronous Parallel model for executing graph transformations based on the Henshin transformation tool is presented in [13]. Finally, model queries are executed for large models in a distributed manner in an extension of EMF Inc-Query by combining incremental graph search techniques and cloud computing [10].

With LinTra [3, 4], and its current implementation written in Java, jLinTra[4], we provide a framework to execute parallel and distributed model transformations that requires all MTs to be executed in Java. With the goal of designing a Domain-Specific Language (DSL), we based our work on T-Core [16], with specific focus on T-Core's collection of primitive operators that allows to write in-place MTs in an intermediate level of abstraction which is between the high-level MTLs and the low-level code used by the engines.

The main difference between T-Core and LinTraP is that T-Core focuses on in-place MT while LinTra focuses on out-place MT. This means that the nature of the problems to address is different and also the way in which the MTs are written. For instance, while in T-Core there exists the primitive Rewriter that update the input model, in LinTra there exists the primitive Creator that creates entities in the output model.

## 5. CONCLUSION AND FUTURE WORK

In this paper, we have presented a collection of primitives which will be combined for running concurrent and distributed out-place model transformations using LinTra.

[4]`http://atenea.lcc.uma.es/index.php/Main_Page/Resources/MTBenchmark`

After having analyzed different high-level MTLs and the LinTra characteristics and having discovered the complete set of primitive operators, there are several other lines of work we would like to explore. First, we will implement the primitives and encapsulate the LinTra code written in Java (jLinTra) into them. To achieve that, we will explore how to formulate, in the most efficient way, the OCL constraints using the methods available in LinTra to query the Blackboard. Second, we plan to create compilers from the most common languages such as ATL or QVT-O to the primitives, so that distributed models can be transformed in parallel reusing MTs written in those languages by means of executing them in the LinTra engine. Third, we want to investigate some annotations for the high-level MTL, so that the user can provide the engine details such as how the parallelization must be done, how the input model should be partitioned, etc. to improve the performance of the transformation. Finally, we plan to investigate the possibility of creating a new and more specific high-level MTL for parallel transformations.

## 6. REFERENCES


[1] G. Bergmann, I. Ráth, and D. Varró. Parallelization of graph transformation based on incremental pattern matching. *ECEASST*, 18, 2009.

[2] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[3] L. Burgueño. Concurrent Model Transformations based on Linda. In *Proceedings of Doctoral Symposium @ MODELS*, pages 9–16, 2013.

[4] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. On the Concurrent Execution of Model Transformations with Linda. In *BigMDE Workshop @ STAF*, 2013.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[6] C. Clasen, M. Didonet Del Fabro, and M. Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. In *Proceedings of CloudMDE Workshop @ ECMFA*, 2012.

[7] J. S. Cuadrado, J. G. Molina, and M. M. Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *Proceedings of ECMFA*, pages 158–172, 2006.

[8] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992.

[9] G. Imre and G. Mezei. Parallel Graph Transformations on Multicore Systems. In *Proceedings of MSEPT*, pages 86–89, 2012.

[10] B. Izsó, G. Szárnyas, I. Ráth, and D. Varró. IncQuery-D: Incremental Graph Search in the Cloud. In *Proceedings of BigMDE Workshop @ STAF*, pages 4:1–4:4, 2013.

[11] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[12] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth,

D. Varró, M. Tisi, and J. Cabot. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proceedings of BigMDE Workshop @ STAF*, 2013.

[13] C. Krause, M. Tichy, and H. Giese. Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In *Proceedings of FASE*, pages 325–339, 2014.

[14] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group, 2011.

[15] E. Syriani and H. Ergin. Operational semantics of UML activity diagram: An application in project management. In *Proceedings of MoDRE Workshop @ RE*, pages 1–8, 2012.

[16] E. Syriani, H. Vangheluwe, and B. LaShomb. T-core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, pages 1–29, 2013.

[17] M. Tisi, S. M. Perez, and H. Choura. Parallel Execution of ATL Transformation Rules. In *Proceedings of MoDELS*, pages 656–672, 2013.

[18] M. van Amstel, S. Bosems, I. Kurtev, and L. F. Pires. Performance in Model Transformations: Experiments with ATL and QVT. In *Proceedings of ICMT*, pages 198–212, 2011.

[19] M. Wimmer, S. Martínez, F. Jouault, and J. Cabot. A Catalogue of Refactorings for Model-to-Model Transformations. *Journal of Object Technology*, 11(2):2:1–40, 2012.

# Improving Memory Efficiency for Processing Large-Scale Models

Gwendal Daniel
AtlanMod team (Inria, Mines Nantes, LINA)
gwendal.daniel@etu.univ-nantes.fr

Gerson Sunyé
AtlanMod team (Inria, Mines Nantes, LINA)
gerson.sunye@inria.fr

Amine Benelallam
AtlanMod team (Inria, Mines Nantes, LINA)
amine.benelallam@inria.fr

Massimo Tisi
AtlanMod team (Inria, Mines Nantes, LINA)
massimo.tisi@inria.fr

## ABSTRACT

Scalability is a main obstacle for applying Model-Driven Engineering to reverse engineering, or to any other activity manipulating large models. Existing solutions to persist and query large models are currently inefficient and strongly linked to memory availability. In this paper, we propose a memory unload strategy for Neo4EMF, a persistence layer built on top of the Eclipse Modeling Framework and based on a Neo4j database backend. Our solution allows us to partially unload a model during the execution of a query by using a periodical dirty saving mechanism and transparent reloading. Our experiments show that this approach enables to query large models in a restricted amount of memory with an acceptable performance.

## Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

## General Terms

Performance, Algorithms

## Keywords

Scalability, Large models, Memory footprint

## 1. INTRODUCTION

The Eclipse Modeling Framework (EMF) is the *de facto* standard for the Model Driven Engineering (MDE) community. This framework provides a common base for multiple purposes and associated tools: code generation [4, 12], model transformation [9, 13], and reverse engineering [17, 6, 5].

*BigMDE '14* July 24, 2014. York, UK.


These tools handle complex and large-scale models when manipulating important applications, for example, during reverse-engineering or software modernization through model transformation. EMF was first designed to support modeling tools and has shown limitations in handling large models. A more efficient persistence solution is needed to allow for partial model loading and unloading, which are key points when dealing with large models.

While several solutions to persist EMF models exist, most of them do not allow partial model unloading and cannot handle models that exceed the available memory. Furthermore, these solutions do not take advantage of the graph nature of the models: most of them rely on relational databases, which are not fully adapted to store and query graphs.

Neo4EMF [3] is a persistence layer for EMF that relies on a graph database and implements an unloading mechanism. In this paper, we present a strategy to optimize the memory footprint of Neo4EMF. To evaluate this strategy, we perform a set of queries on Neo4EMF and compare them against two other persistence mechanisms, XMI and CDO. We measure performances in terms of memory consumption and execution time.

The paper is organized as follows: Section 2 presents the background and the motivations for our unloading strategy. Section 3 describes our strategy and its main concepts: *dirty saving*, *unloading*, and *extended on-demand loading*. Section 4 evaluates the performance of our persistence layer. Section 5 compares our approach with existing solutions and finally, Section 6 concludes and draws the future perspectives of the tool.

## 2. BACKGROUND

### 2.1 EMF Persistence

As many other modeling tools, EMF has adopted XMI as its default serialization format. This XML-based representation has the advantage to be human readable, but has two drawbacks: (i) XMI sacrifices compactness for an understandable output and (ii) XMI files have to be entirely parsed to get a readable and navigational model. The former drawback reduces efficiency of I/O access, while the latter

increases the memory needed to load a model and limits on-demand loading and proxy uses between files. XMI does not provide advanced features such as model versioning or concurrent modifications.

The CDO [8] model repository was built to solve those problems. It was designed as a framework to manage large models in a collaborative environment with a small memory footprint. CDO relies on a client-server architecture supporting transactional accesses and notifications. CDO servers are built on top of several persistence solutions, but in practice only relational databases are used to store CDO objects.

## 2.2 Graph Databases

Graph databases are one of the NoSQL data models that have emerged to overcome the limitations of relational databases with respect to scale and distribution. NoSQL databases do not ensure ACID properties, but in return, they are able to handle efficiently large-scale data in a distributed environment.

Graph databases are based on nodes, edges, and properties. This particular data representation fits exactly to EMF models, which are intrinsically graphs (each object can be seen as a node and references as edges). Thus, graph databases can store EMF models without a complex serialization process.

# 3. NEO4EMF

Neo4EMF is a persistence layer built on top of the EMF framework that aims at handling large-models in a scalable way. It provides a compatible EMF API and a graph-database persistence backend based on Neo4j [16].
Neo4EMF is open source and distributed under the terms of the (A)GPLv3 [1].

In previous work [3], we introduced the basic concepts of Neo4EMF : *model change tracking* and *on-demand loading*. Model change tracking is based on a global changelog that stores the modifications done on a model during an execution (from creation to save). Tracking the modifications is done using EMF notification facilities: the changelog acts as a listener for all the objects and creates its entries from the received notifications. Neo4EMF uses an on-demand loading mechanism to load object fields only when they are accessed. Technically, each Neo4EMF object is instantiated as an empty container. When one of its fields (**EReferences** and **EAttributes**) is accessed, the associated content is loaded. This mechanism presents two advantages: (i) the entire model does not have to be loaded at once and (ii) unused elements are not loaded.

Neo4EMF does not use the **EStore** mechanism. Indeed, **EStore** allows the **EObject** data storage to be changed by providing a stateless object that translates model modifications and accesses into backend calls. Every generated accessor and modifier delegates to the reflexive API. As a consequence, **EObjects** have to fetch through the store each time a field is requested, engendering several database queries. On the contrary, Neo4EMF is based on regular **EObjects** (with in-memory fields) which are synchronized with a database backend.

In this paper we focus on Neo4EMF memory footprint. We introduce a strategy to unload some parts of a processed model and save memory during a query execution. In the previous implementation, the on-demand loading mechanism allows us to load only the parts of the model that are needed, but there is no solution to remove unneeded objects from memory, especially when they were changed but not saved yet.

A reliable unload strategy needs to address two main issues:

- **Accessibility:** Contents of unloaded objects (attributes and referenced objects) have to remain accessible through standard EMF accessors.
- **Transparency:** The management of the object life cycle has to be independent from users, but customizable to fit specific needs, e. g., size of the Java virtual machine, requirements on execution time, etc.

Our strategy faces these issues by providing a *dirty-saving* mechanism, which provides temporary and transparent model persistence. The object life cycle has also been modified to include unloading of persisted elements.

In this next sections, we provide an overview of the changelog used to record the modifications of the processed model. Then, we present *dirty saving*, based on the basic Neo4EMF save mechanism, and we describe the Neo4EMF object life cycle. Finally, we describe the modifications done on the *on-demand loading* feature to handle this new strategy.

## 3.1 Neo4EMF Changelog

Neo4EMF needs a mechanism to ensure synchronization between the in-memory model and its backend representation, avoiding systematic unnecessary calls to the database.

Despite the existence in EMF of a modification tracking mechanism, the **ChangeRecorder** class, we decided to develop an alternative solution that minimizes memory consumption.

Neo4EMF tracks model modifications in a changelog, a sequence of entries of five types:

**Object creation:** A new object has been created and attached to a Neo4EMF resource.

**Object deletion:** An object has been deleted or removed from a Neo4EMF resource.

**Attribute modifications:** Attribute setting and unsetting.

**Reference addition:** Assignment of a new single-valued reference or addition of a new referenced object in a multi-valued one.

**Reference deletion:** Unsetting a single-valued reference or removing a referenced object in a multi-valued one.

We distinguish *unidirectional* and *bidirectional* reference modifications for performance reasons (they are not serialized the

same way during the saving process).
Figure 1 summarizes our changelog model. All changelog entries are subclasses of **Entry**, which defines some shared properties: the object concerned by the modification (for instance the object containing a modified attribute or reference, or the new object in case of a CreateObject entry) and a basic serialization method.

Attribute and reference modification entries (**SetAttribute**, **AddLink**, **RemoveLink** and their subclasses) have three additional fields to track fine-grained modifications: the updated feature (attribute or reference identifier) which corresponds to the modified field of the concerned object, the new and old values of the feature (if available).

This decomposition provides a direct access to the information required during the serialization process, without accessing the concerned objects. The fine-grained entry management also decreases memory consumption. For instance modifications on bidirectional references correspond to a single changelog entry, while they needed two basic entries before. Serialization of those entries is also more efficient since it reduces the number of database accesses.

In the previous version of Neo4EMF, we used the EMF notification framework to create changelog entries. This implementation had a major drawback: notifications were handled in a dedicated thread, and we could not ensure that all the notifications were sent to the changelog before its serialization. This behavior could create an inconsistency between the in-memory model and the saved one. This is another reason we do not use the EMF **ChangeRecorder** facilities, which relies on notifications.

In this new version, changelog entries are directly created into the body of the generated methods. This solution removes synchronization issues and is also more efficient, because entries are created directly, and all the information needed to construct them is available in the method body (current object, feature identifier, new and old values). We also do not have to deal with the generic notification API, which was resulting in a lot of casts and complex processing to retrieve this information. Synchronizing the changelog brings another important benefit: the causality between model modifications and entries order is ensured and there is no need to reorder the entry stack before its serialization.

Finally, we modify the changelog life cycle. In the previous version, the changelog was a global singleton object, containing the record of a full execution, mixing modifications of multiple resources. This solution is not optimal because saving is done per resource in EMF, and to save a single resource the entire modification stack needed to be processed to retrieve the corresponding entries. We choose to create a dedicated changelog into each Neo4EMF resource that handles modifications only for the objects contained in the associated resource. This modification reduces the complexity of the save processing: the resource changelog is simply iterated and its entries are then serialized into database calls. The synchronized aspect of the changelog allows us to process the entries in the order they are added, which was not possible in the previous version.
Furthermore, associating a changelog with a resource ensures that, when the resource is deleted, all the related entries are also deleted. In the previous version, entries could not be deleted from the global changelog, and were kept in memory during the execution.

**Figure 2: Excerpt of MoDisco Java Metamodel**

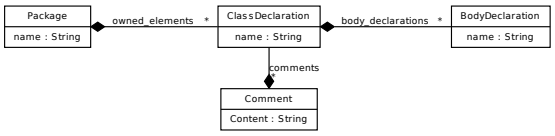
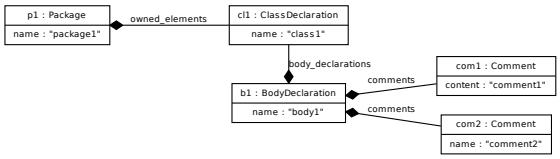



**Figure 3: Sample instance of Java Metamodel**




## 3.2 Dirty Saving

Neo4EMF relies on a mapping between EMF entities and Neo4j concepts to save its modifications. Figure 2 shows an excerpt of the Java metamodel, used in the MoDisco [17] project. This metamodel describes Java applications in terms of **Packages**, **ClassDeclarations**, **BodyDeclarations**, and **Comments**. A **Package** is a named container that gathers a set of **ClassDeclarations** through its **owned_elements** composition. A **ClassDeclaration** is composed of a name, a set of **Comments** and a set of **BodyDeclarations**.
Figure 3 shows a simple instance of this metamodel: a **Package** (package1), containing one **ClassDeclaration**, (class1). This *ClassDeclaration* contains two **Comments** (comment1 and comment2) and one single **BodyDeclaration** (body1). Figures 2, 3, and 4 show that:

**Model elements** are represented as nodes. Nodes with identifier `p1`, `cl1`, `b1`, and `com1` are examples corresponding to `p1`, `cl1`, `b1`, and `com1` in Figure 3. The ROOT node represents the entry point of the model (the resource directly or indirectly containing all the other elements) and is not associated to a model object.

**Elements attributes** are represented as node properties. Node properties are $\langle name, value \rangle$ pairs, where *name* is the feature identifier and *value* the value of the feature. Node properties can be observed for `p1`, `cl1`, and `b1`.

**Metamodel elements** are also represented as nodes and are indexed to facilitate their access. Metamodel nodes have two properties: the metaclass name and the metamodel unique identifier. `P`, `Cl`, `B` and `Com` are examples of metamodel element nodes, they correspond to **PackageDeclaration**, **ClassDeclaration**, **BodyDeclaration**, and **Comment**, respectively in Figure 2

**Figure 1: Changelog Metamodel**

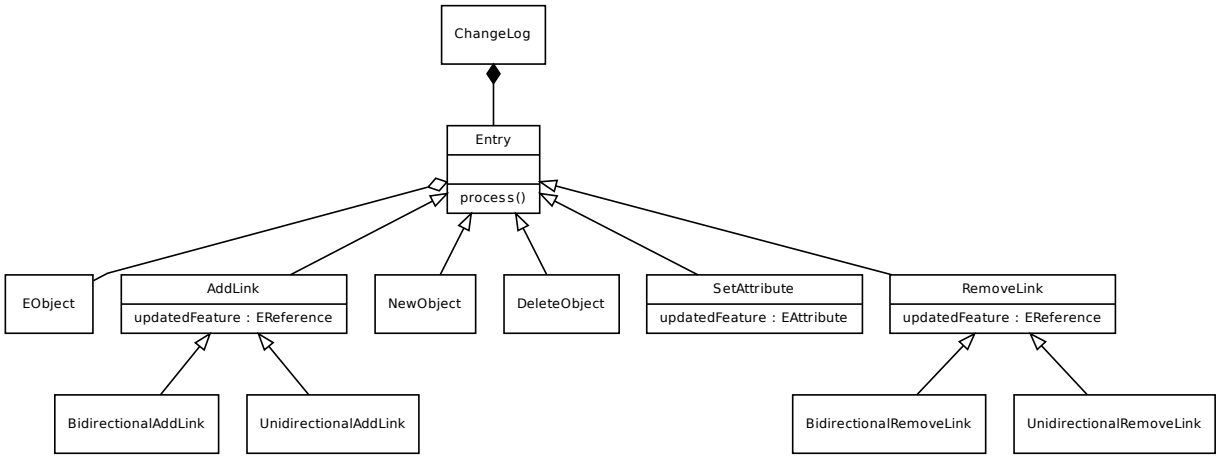
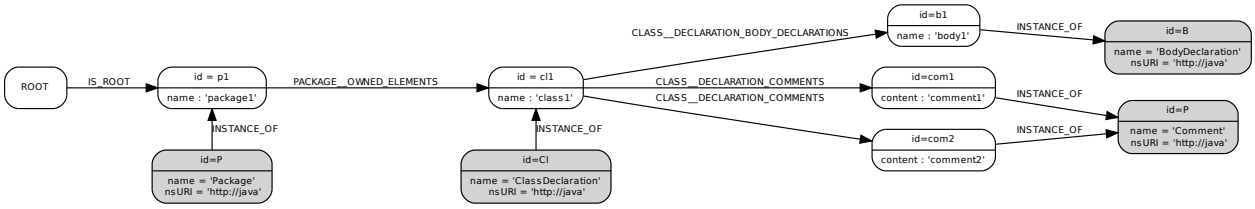



**Figure 4: Sample instance database representation**




**InstanceOf relationships** are outgoing relationships between the elements nodes and the nodes representing metaclasses. They represent the conformance of an object instance to its class definition

**References** between objects are represented as relationships. To avoid naming conflicts relationships are named using the following convention:
CLASS_NAME__REFERENCE_NAME.

When a save is requested, changelog entries are processed to update the database backend. Each entry is serialized into a database operation. The **CreateObject** entry corresponds to the creation of a new node and its meta-information (INSTANCEOF to its meta-class, ISROOT if the object is directly contained in the resource). All the fields of the object are also serialized and directly saved in the database. A **SetAttribute** entry corresponds to an update of the related node's property with the corresponding name. **AddLink**, **RemoveLink**, and their subclasses respectively record the creation and removal of a relationship, storing the containing **class** and **feature** name.

We decide to serialize at the same time a created object and all its references and attributes. New objects need to be entirely persisted, and there is no reason to record their modifications before their first serialization (the final state of the object is the one that needs to be persisted). This full serialization behavior has the advantage of generating only one single entry for a new object, independently from the number of its modified fields.

This approach works well for small models, but has issues when a large modification set needs to be persisted: the changelog grows indefinitely until the user decides to save it. This is typically the case in reverse engineering, where the extracted objects are first all created in memory and only afterwards they are saved.

To address this problem we introduce *dirty-saving*, a periodical save action not requested by the user. The period is determined by the changelog size, configurable through the Neo4EMF resource. Since these save operations are not requested by the user they have to ensure two properties:

- **Reversibility:** if the modifications are canceled or if the user does not want to save a session the database should rollback to an acceptable version. This version is either (i) the previous regularly saved database if an older version exists or (ii) an empty database.

- **Persistability:** if a regular save is requested by the user, the temporary objects in the database have to be definitely persisted. They can then constitute a new acceptable version of the database if a rollback is needed.

We introduce a new mapping for changelog entries with the purpose of temporary dirty saving. This mapping is based on the same entries as the regular mapping but the associated Neo4j concepts allow the system to easily extract dirty objects and regular ones. In addition we create two indexes: `tmp_relationships` and `tmp_nodes` which respectively contain the dirty relationships and nodes (i. e., created in a dirty saving session). Figure 5 summarizes the mapping between changelog entries and *neo4j* concepts:

- **CreateObject**: creation of a new node (as in the regular saving process) and addition to the `tmp_nodes` index.
- **SetAttribute**: creation of a dedicated node containing the dirty attributes. The idea is to keep a stable version (i. e., the previous regularly saved version) to easily reverse it. A **SetAttribute** relationship is created to link the base object and its attribute node
- **AddLink**: creation of a generic **AddLink** relationship, containing the reference identifier as a property. This special relationship format is needed to easily process dirty relationships and retrieve their corresponding image if a regular save operation is requested
- **RemoveLink**: creation of a generic **RemoveLink** relationship, containing the reference identifier as a property. **AddLink** and **RemoveLink** relationships with the same reference identifier and target object are mutually exclusive to limit the number of temporary objects into the database
- **DeleteObject**: creation of a special **Delete** relationship looping on the related node. The base version of the node is kept alive if a rollback is needed.

The objective of this mapping is to preserve all the information contained after a regular save, to easily handle a rollback. That is why object deletion is done using a relationship: if the modifications are aborted it is simpler to remove the relationship than creating a new instance of the node with backup information. We do not use a property to tag deleted objects for performance reasons (access to node properties is slower than edge navigation).
To persist definitely dirty objects in the database into regularly saved ones a serialization process is invoked. As changelog entries, each Neo4j element contains all the information needed to create their regular equivalents: new objects are simply removed from the `tmp_nodes` index, **AddLink** relationships are turned into their regular version using their properties and **RemoveLink** entries correspond to the deletion of their existing regular version.

For example if we update the model given in Figure 3 by removing `com1` and creating a new **BodyDeclaration** `body2` then calling a dirty save, the database will be updated as in Figure 6. Note that a **Delete** relationship has been created because the removed **Comment** is not contained in the resource anymore. Red relationships and nodes are indexed respectively in `tmp_relationships` and `tmp_nodes` indexes.

This example shows that our mapping is built on top of the existing one: there is no modification done on the previous version, represented with black nodes. This simplifies the rollback process, which consists of a deletion of all the temporary Neo4j objects.

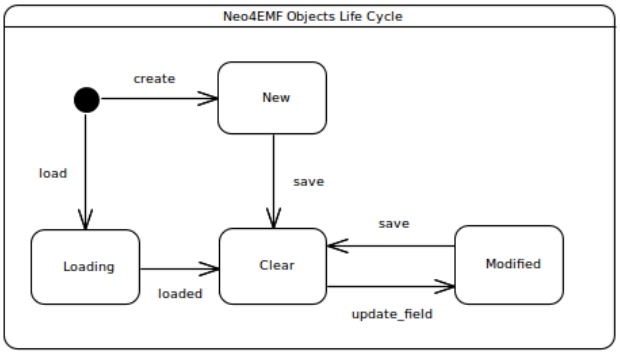
## 3.3 Object Life Cycle

We modify the Neo4EMF object life cycle to enable unloading. When a *dirty saving* is invoked, all the modifications contained in the changelog are committed to the database. Because of this persistence, persisted objects can be safely released from memory and reloaded using *on-demand loading*, if needed.

Figure 7 shows the different life cycle states of a Neo4EMF object. When a Neo4EMF object is created it is **New**: it has not been persisted into the database and cannot be released. When a save is requested or a dirty save is invoked, the new object is persisted into the database and it is tagged as **Clear**: all the known modifications related to the object have been saved and it is fetchable from the database without information loss. In this state the object can be removed from memory without consistency issues. When a modification is done on the object (setting an attribute or updating a reference) then it is tagged as **Modified**.

Modified objects cannot be released, because their database-mapped nodes do not contain the modified information. When a save is processed, the **Modified** objects revert to **Clear** state and can be released again. **Loading** objects also have a particular state that avoids garbage collection of an object when it is loading.

**Figure 7: Neo4EMF EObject life cycle**




To allow garbage collection of Neo4EMF objects, we use Java Soft and Weak references to store object's fields. Weak and Soft referenced objects are eligible for garbage collection as soon as there is no strong reference chain on them. The

**Figure 5: Changelog to Neo4j entity mapping**

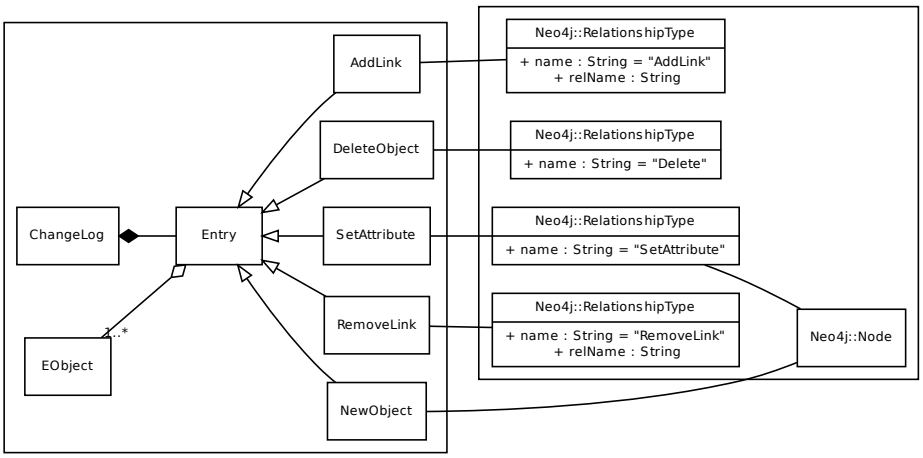
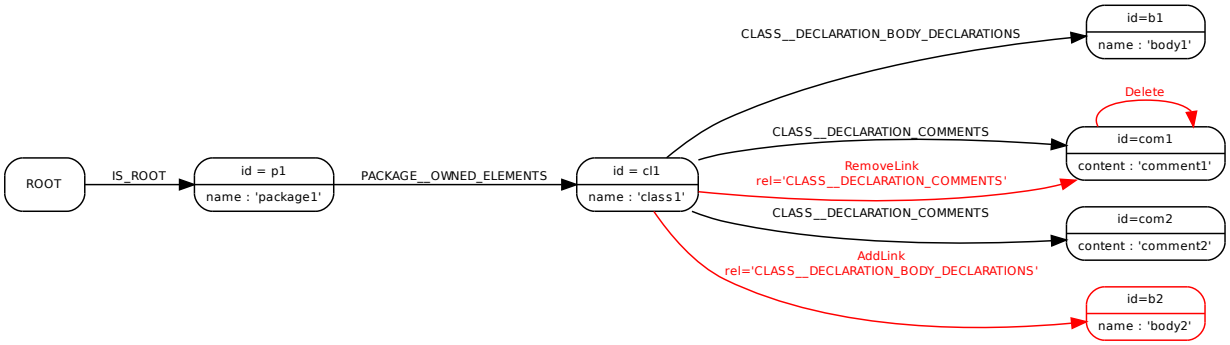



**Figure 6: Database state after modifications**




difference between the two kinds of references is the time they can remain in memory. Weak references are collected as soon as possible by the garbage collector, whereas Soft references can be retained in memory as long as the garbage collector does not need to free them (i.e., as long as there is enough available memory). This particular behavior is interesting for cache implementation and to optimize execution speed in a large available memory context. Reference type (Weak or Soft) can be set through Neo4EMF resource parameters.

In Section 3.1, we describe that changelog entries contain all the information related to the serialization of the concerned object. This information constitutes the strong reference chain on the related object fields. When a save is done, entries are processed and deleted, breaking the strong reference chain and making objects eligible for garbage collection.

Neo4j's objects are not impacted by this new life-cycle. The database manages its objects life cycle through a policy defined at the resource creation (memory or performance preferences).

## 3.4 Extended On-Demand Loading

To handle the new architecture of our layer, we have to extend the *on-demand loading* feature to support temporary persisted objects. On-demand loading uses two parameters: (i) the object that handles the feature to load and (ii) the identifier of the feature to load. This behavior implies that a Neo4EMF object is always loaded from another Neo4EMF object.

Figure 6 shows our Java metamodel instance state after a dirty save. The database content is a mix between regularly saved objects (in black) and dirty-saved ones (in red). Loading referenced **Comments** instances from **ClassDeclaration** `cl1` is done in three steps to ensure the last dirty-saved

operations have been considered.
First, CLASS_DECLARATION__COMMENTS relationships are processed and their end nodes are saved. Second, the **AddLink** relationships containing the corresponding **rel** property are processed and their end nodes are added to the previous ones. This operation retrieves all the associated nodes for the given feature, regular ones and dirty ones. Third, **RemoveLink** relationships are processed the same way and their end nodes are removed from the loaded node set.

Attribute fetching behavior is a bit different: if a node representing an object has relationships to a dedicated attribute node, then the data contained in this node is returned instead of the base node property.

To improve the performances of our layer, we create a cache that maps Neo4j identifiers to their associated object. When *on-demand loading* is performed, the cache is checked first, avoiding the cost of a database access. This cache is also used to retrieve released objects.

# 4. EVALUATION

In this section, we evaluate how the memory footprint and the access time of Neo4EMF scale in different large model scenarios, and we compare it against CDO and XMI. These experiments are performed over two EMF model extracted with the MoDisco Java Discoverer [17]. Both models are extracted from Eclipse plug-ins: the first one is an internal tool and the second one is the Eclipse *JDT* plugin. The resulting XMI files are 20 MB and 420 MB, containing respectively around 80 000 and 1 700 000 elements.

## 4.1 Execution Environment

Experiments are executed on a computer running Windows 7 professional edition 64 bits. Interesting hardware elements are: an Intel Core I5 processor 3350P (3.5 GHz), 8 GB of DDR3 SDRAM (1600 MHz) and a Seagate barracuda 7200.14 hard disk (6 GB/s). Experiments are executed on Eclipse 4.3 running Java SE Runtime Environment 1.8.

To compare the three persistence solutions, we generate three different EMF models from the MoDisco Java Metamodel: (i) the standard EMF model, (ii) the CDO one and (iii) the Neo4EMF one. We import both models from XMI to CDO and Neo4EMF and we verify they contain the same data after the import.

Neo4EMF uses an embedded Neo4j database to store its objects. To provide a meaningful comparison in term of memory consumption we choose to use an embedded CDO server.

**Experiment 1: Object creation.** In this first experiment, we execute an infinite loop of object creation and simply count how many objects have been created before a **OutOfMemoryException** is thrown. We choose a simple tree structure of three classes to instantiate from the MoDisco Java metamodel: a parent **ClassFile** containing 1000 **BlockComment** and **ImportDeclaration**. The resulting model is a set of independent element trees. For this experiments we choose a 1 GB Java virtual machine and an arbitrarily fixed changelog size of 100 000 entries. Table 1 summarizes the results.

| **Persistence Layer** | XMI | CDO | Neo4EMF |
|---|---|---|---|
| **#Created Elements** | 22 939 780 | 4 378 990 | >40 000 000[1] |

**Table 1: Number of Created Elements Before Memory Overhead**

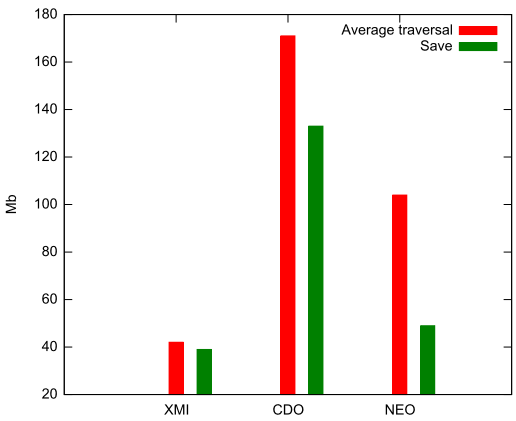



**Figure 8: Memory Consumption: Model Traversal and Save** (20 MB)

Note that the number given for Neo4EMF is an approximation: we stop the execution before any **OutOfMemory** error. The average memory used to create elements was around 500 MB and does not seem to grow. This performance is due to the *dirty-saving* mechanism: created objects generate entries in the changelog. When the changelog is full, changes are saved temporarily in the database, freeing the changelog for next object creations.

**Experiment 2: Model traversal.** In this experiment, we load a model and execute a traversal query that starts from the root of the model, traverses all the containment tree and modifies the **name** attribute of all **NamedElements**. All the modifications are saved at the end of the execution. During the traversal, we measure the execution time for covering the entire model and the average memory used to perform the query. In addition, we measure the memory needed to save the modifications at the end of the execution. Figures 8 and 9 summarize memory results. As expected, the Neo4EMF traversal footprint is higher than the XMI one because we include the Neo4j embedded database and runtime in our measures. Unloading brings a real interest when comparing the results with CDO: when removing unused (i. e., unreferenced) objects we save space and process the request in a reduced amount of memory. For this experiment we use a 4 GB Java virtual machine, with the `ConcMarkSweepGC` garbage collector, recommended when using Neo4j.

**Experiment 3: Time performance.** This experiment is similar to the previous one, but we focus on time performances. We measure the time needed to perform traversal and save. Figures 10 and 11 summarize the results. To provide a fair comparison between full and on-demand loading strategies we also include model loading time with the traversal queries.

[1]The execution was stopped before any memory exception.

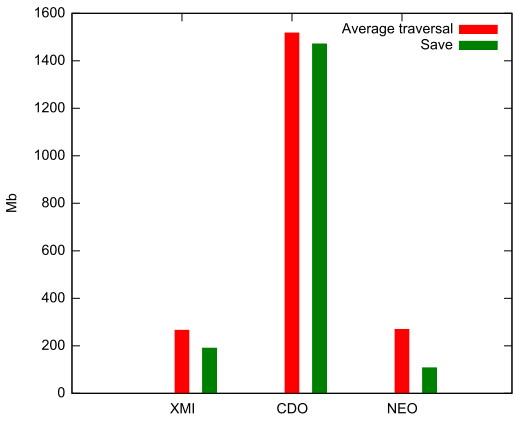



**Figure 9: Memory Consumption: Model Traversal and Save** (420 MB)

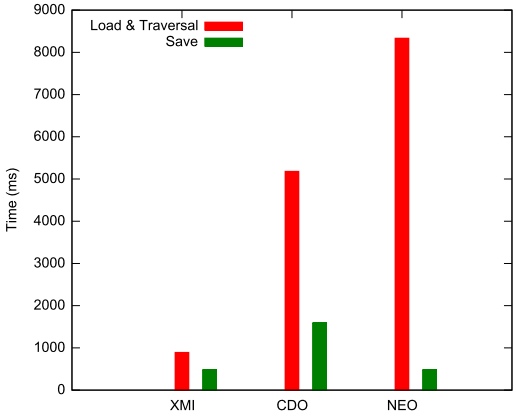



**Figure 10:** 20 MB **model traversal and save performances**

Neo4EMF save performances can be explained with *dirty-saving*: during the traversal, entries are generated to track the *name* modifications. These entries are then saved in the database when the changelog is full, reducing the final save cost. This behavior also explains a part of the traversal time overhead, when compared to CDO: Neo4EMF traversal implies database write access for *dirty saving* where CDO does not, related I/O accesses considerably impact performance.

## 4.2 Discussion

The results of these experiments show that dirty-saving coupled with on-demand loading decrease significantly the memory needed to execute a query. As expected, this memory footprint improvement worsens the time performances of our tool, in particular because of dirty-saving, which generates several database calls. That is why we provide *dirty saving* configuration through the Neo4EMF resource. The experiments also show that Neo4EMF is able to handle large queries and modifications in a limited amount of memory, compared to existing solutions.

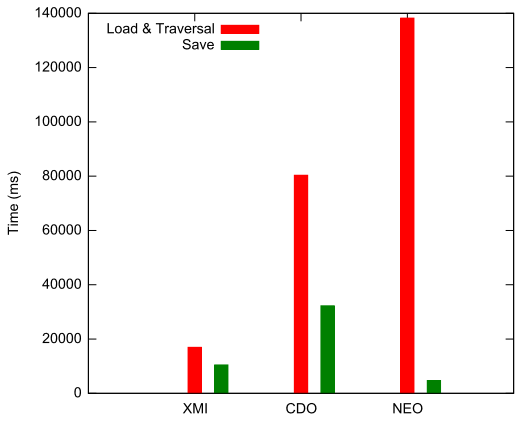



**Figure 11:** 420 MB **traversal and save performances**

We also run our benchmarks on different operating systems (Ubuntu 12.04 and 13.10) and we find that CDO and Neo4EMF time performances seem to be linked to the file partition format (especially in I/O accesses): Neo4j has better performances on these operating system (with a factor of 1.5) and CDO has slower times (with approximately the same factor). More investigation is needed to optimize our tool in different contexts.

Our experiments show that Neo4EMF is an interesting alternative to CDO to handle large models in memory constrained environment. On-demand loading and transparent unloading offer a small memory footprint (smaller than CDO in our experiments), but our solution does not provide advanced features like collaborative edition and versioning provided by CDO.

The unload strategy is transparent for the user, but may be intrusive in some cases, for instance if the hard-drive memory space is limited or the time performances are critical. This is why we introduce configuration for *dirty saving* and changelog size through the Neo4EMF resource.

# 5. RELATED WORK

Models obtained by reverse engineering with EMF-based tools such as MoDisco [17, 5, 11] can be composed of millions of elements. Existing solutions to handle this kind of models have shown clear limitations in terms of memory consumption and processing.

CDO is the *de facto* standard to handle large models using a server and a relational database. However, some experiments have shown that CDO does not scale well to very large models [2]. Pagán et al. [14, 15] propose to use NoSQL databases to store models, especially because those kind of databases should fit better to the interconnected nature of EMF models.

Mongo EMF [7] is a NoSQL approach that stores EMF models in MongoDB, a document-oriented database. However, Mongo EMF storage is different from the standard EMF persistence backend, and cannot be used *as is* to replace an other persistence solution in an existing system. Modifications on the client software are needed to integrate it.

Morsa [14] is an other persistence solution based on MongoDB database. Similarly to Neo4EMF, Morsa uses a standard EMF mechanism to ensure persistence, but it uses a client-server architecture, like CDO. Morsa has some similarities with Neo4EMF, notably in its *on-demand loading* mechanism, but does not use a graph database.

EMF Fragments [10] is another EMF persistence layer based on a NoSQL database. The EMF Fragments approach is different from other NoSQL persistence solutions: it relies on the proxy mechanism provided by EMF. Models are automatically partitioned and loading is performed by partition. Loading on demand is only performed for cross-partition references. Another difference with Neo4EMF is that EMF Fragments needs to annotate the metamodels to provide the partition set, whereas our approach does not require model adaptation or tool modification.

# 6. CONCLUSION AND FUTURE WORK

In this paper, we presented a strategy to optimize the memory footprint of Neo4EMF, a persistence layer designed to handle large models through *on-demand loading* and transparent unloading. Our experiments show that Neo4EMF is an interesting alternative to CDO for accessing and querying large models, especially in small available memory context, with a tolerable performance loss. Neo4EMF does not have collaborative model editing or model versioning features, which biases our results: providing those features may imply a more important memory consumption.

In future work, we plan to improve our layer by providing partial collection loading, allowing the loading of large collections subparts from the database. In our experiments, we detected some memory consumption overhead in this particular case: when an object contains a huge number of referenced objects (through the same reference) and they are all loaded at once.

We then plan to study the inclusion of attribute and reference meta-information directly in the database to avoid unnecessary object loading: some EMF mechanisms, like `isSet` may induce *load on demand* of the associated attribute, just in order to make a comparison. It could be interesting to provide this information from the database without a complete and costly object loading.

Finally, we want to introduce loading strategies such as prefetching or model partitioning (using optional metamodel annotations or a definition of the model usage) to allow users to customize the object life cycle.

# 7. REFERENCES


[1] AtlanMod. Neo4EMF, 2014. URL: `http://www.neo4emf.com/`.

[2] K. Barmpis and D. S. Kolovos. Comparative analysis of data persistence technologies for large-scale models. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pages 33–38, New York, NY, USA, 2012. ACM.

[3] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4emf, a scalable persistence layer for emf models. July 2014.

[4] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013.

[5] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014.

[6] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. Modisco: A generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.

[7] Bryan Hunt. MongoEMF, 2014. URL: `https://github.com/BryanHunt/mongo-emf/wiki/`.

[8] Eclipse Foundation. The CDO Model Repository (CDO), 2014. URL: `http://www.eclipse.org/cdo/`.

[9] INRIA and LINA. ATLAS transformation language, 2014.

[10] Markus Scheidgen. EMF fragments, 2014. URL: `https://github.com/markus1978/emf-fragments/wiki/`.

[11] Modeliosoft Solutions, 2014. URL: `http://www.modeliosoft.com/`.

[12] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. Acceleo user guide, 2006.

[13] OMG. MOF 2.0 QVT final adopted specification (ptc/05-11-01), April 2008.

[14] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A scalable approach for persisting and accessing large models. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 77–92, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] J. E. Pagán and J. G. Molina. Querying large models efficiently. *Information and Software Technology*, 2014. IN PRESS, ACCEPTED MANUSCRIPT. URL: `http://dx.doi.org/10.1016/j.infsof.2014.01.005`.

[16] J. Partner, A. Vukotic, and N. Watt. *Neo4j in Action*. O'Reilly Media, 2013.

[17] The Eclipse Foundation. MoDisco Eclipse Project, 2014. URL: `http://www.eclipse.org/MoDisco/`.

# MONDO-SAM: A Framework to Systematically Assess MDE Scalability

Benedek Izsó, Gábor Szárnyas, István Ráth and Dániel Varró
Fault Tolerant Systems Research Group
Department of Measurement and Information Systems
Budapest University of Technology and Economics
H-1117, Magyar Tudósok krt. 2.
Budapest, Hungary
{izso, szarnyas, rath, varro}@mit.bme.hu *

## ABSTRACT

Processing models efficiently is an important productivity factor in Model-Driven Engineering (MDE) processes. In order to optimize a toolchain to meet scalability requirements of complex MDE scenarios, reliable performance measures of different tools are key enablers that can help selecting the best tool for a given workload. To enable systematic and reproducible benchmarking across different domains, scenarios and workloads, we propose MONDO-SAM, an extensible *MDE benchmarking framework*. Beyond providing easily reusable features for common benchmarking tasks that are based on best practices, our framework puts special emphasis on metrics, which enables scalability analysis along different problem characteristics. To illustrate the practical applicability of our proposal, we demonstrate how different variants of a model validation benchmark featuring several MDE tools from various technological domains have been integrated into the system.

## 1. INTRODUCTION

As Model-Driven Engineering (MDE) has gained mainstream momentum in complex system development domains over the past decade, scalability issues associated to MDE tools and technologies are nowadays well known [6]. To address these challenges, the community has responded with a multitude of benchmarks.

The majority of these efforts have been created by tool providers for the purpose to measure performance developments of specific engines [8, 2]. As a notable exception, the Transformation Tool Contest (TTC) [1] attempts cross-technology comparison by proposing multiple cases which are solved by the authors of (mainly EMF based) MDE tools.

*This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003) and MONDO (EU ICT-611125) projects partly during the fourth author's sabbatical.

*BigMDE'14* July 24, 2014. York, UK.


TTC cases focus on measuring query and transformation execution time against instance models of increasing size. TTC promotes reproducibility by providing pre-configured virtual machines on which individual tools can be executed; however, the very nature of this environment and the limited resources make precise comparison difficult.

Benchmarks are also used outside of the MDE community. The *SP²Bench* [7] and *Berlin SPARQL Benchmark (BSBM)* [3] are SPARQL benchmarks over semantic databases (triple stores). The first uses RDF models based on the real world DBLP bibliography database, while the latter is centered around an e-commerce case study. Both benchmarks scale up in the size of models (up to 25M and 150B elements), however SP²Bench does not consider model modifications, and BSBM does not detail query and instance model complexity. SPLODGE [4] is another similar approach, where SPARQL queries were generated systematically, based on metrics for a predefined dataset. Queries are scaled up to three navigations (joins), but other metrics as the complexity of the instance model were not investigated. The common technological characteristics of these benchmarks is that they are frequently run on very large computer systems that are not accessible to most users, or rely on commercial software components that are hard to obtain.

To summarize, currently available graph based benchmarks are affected by two main issues: (i) technologically, they are frequently built on virtualized architectures or have exotic dependencies, making measurements hard to reproduce independently; and (ii) conceptually, they typically only analyze measurement results against a limited view of the problem: the execution time of a fixed task scaled against increasing model size. As a result, the relative complexity of current benchmarks can not be precisely quantified, which makes them difficult to compare them to each other.

In previous work [5], we have found that other metrics (such as various query complexity measures, instance model characteristics, and the combination of these) can affect results very significantly. Building on these results, in this paper we propose the extensible MONDO-SAM framework that is integrated into the official MONDO benchmark open repository[1]. MONDO-SAM provides *reusable benchmark-*

[1] `http://opensourceprojects.eu/p/mondo/d31-transformation-benchmarks/`

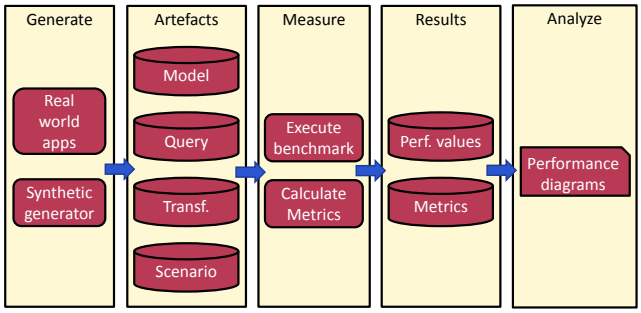



Figure 1: Benchmarking process.

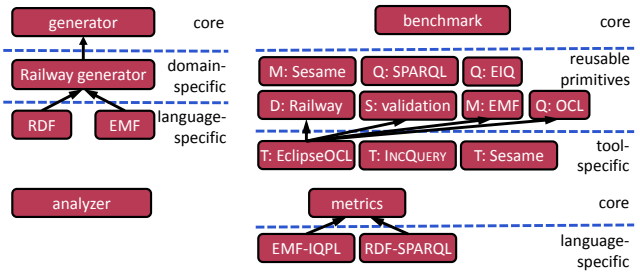



Figure 2: Benchmark framework architecture.

*ing primitives* (like metrics evaluation, time measurement, result storage) that can be flexibly organized into *benchmarking workflows* that are specific to a given case study. MONDO-SAM also provides an API so that tehnologically different tools can be integrated into the framework in a uniform way. A unique emphasis of the framework is built-in support for metrics calculation that enables characterization of the benchmarking problems as published in [5]. The built-in reporting facility allows to investigate the scalability of MDE tools along different metrics in diagrams. Finally, the entire framework and integrated case studies can be compiled and run using the Maven build system, making deployment and reproducible execution in a standard, Java-enabled computing environment feasible.

# 2. OVERVIEW OF THE FRAMEWORK

## 2.1 A process model for MDE benchmarks

The benchmarking process for MDD applications is depicted in Fig. 1. Inputs of the benchmark are the instance model, queries run on the instance model, the transformation rules or modification logics and a scenario definition (or workflow) describing execution sequences. In this case, scenario can describe MDD use cases (like model validation, model transformation, incremental code generation), including warmup and teardown operations, if required. Inputs can also be derived from real-world applications, or are synthetically generated providing complete control over the benchmark. Complexity of the input is characterized by metrics, while scenario execution implementations are instrumented to measure resource consumption (wall-clock times, memory and I/O usage). Finally, these measured values and calculated metrics are visualized on diagrams automatically to find the fastest tool, or to identify performance improvements of a specific tool.

## 2.2 Architecture

The benchmark framework consisting of four components is depicted in Fig. 2. The *generator component* allows synthetic generation of benchmark inputs. The core module handles configuration, domain-specific modules describe generation method of input data (like generation of instance models, queries), and language-specific modules serialize generated logical artifacts into files (like EMF models or OCL queries). The selected domain constrains languages, as domain description concepts must be supported. For example transitivity or multi-level metamodeling is not supported by EMF, but the latter is required by the e-commerce case study of BSBM. Generated models should be semantically equivalent, however, it is a question whether structural equality should be preserved. E.g. in certain cases EMF models must have a dedicated container object with containment relations to all objects which is not required in RDF.

## 2.3 Core features

*Benchmark component.* The *benchmark component* (in Fig. 2) measures performance of different tools for given cases. A case can be defined as a quintuple of $(\mathcal{D}, \mathcal{S}, \mathcal{M}, \mathcal{Q}, \mathcal{T})$, where $\mathcal{D}$ defines the domain, $\mathcal{S}$ the scenario, $\mathcal{M}$ the modification and $\mathcal{Q}$ the query. The $\mathcal{T}$ modules implement tool specific glue code and select $\mathcal{D}, \mathcal{S}, \mathcal{M}, \mathcal{Q}$. All modules reuse common functions of the core, like configuration (with default values and tool-specific extensions), wall-clock time measurement which is done with highest (nanosecond) precision (that does not mean same accuracy), and momentary memory consumption, which are recorded in a central place. At runtime, language-specific modifications (transformations), queries, and instances of the selected domain must be available.

*Model instantiator.* A common aspect of the *generator and the benchmark module* is reproducibility. In tool-specific scenario implementations boundaries are well separated by the scenario interfaces, and where generation or execution is randomized, a pseudo-random generator is used with the random seed set to a predefined value. However, nondeterministic operations (like choosing an element from a set) and tool implementations can disperse results between runs.

*Metrics evaluator.* To describe benchmark input with quantitative values, they are characterized by metrics which are evaluated by the *metrics component*. Language specific implementations analyze model-query pairs, and store calculated metric values centrally gathered by the core which are analyzed later together with the measured values.

*Result reporting and analysis.* When measurement and metrics data become available, the *analyzer component* (implemented in R) automatically creates HTML report with diagrams. To show scalability according to different measures, on the $x$ axis metrics can be selected, while the $y$ axis represents resource consumption. Raw data can be post-

processed, i.e. dimensions can be changed (e.g. to change time to ms dimension to reflect its accuracy), and derived values can be calculated (e.g. the median of incremental recheck steps, or total processing time).

## 2.4 Best Practices to Minimize Validity Threats

During the execution of the cases, noise coming from the environment should be kept at minimum. Possible sources of noise include the caching mechanisms of various components (e.g. file system and the database management system), warm-up effect of the runtime environment (e.g. the Java Virtual Machine), scheduled tasks (e.g. cron) and swapping. For large heaps, the Garbage Collector of the JVM can block the run for minutes, so minimizing its call is advised which is achieved by setting minimal and maximal heap size to an equal value, thus eliminating GC calls at memory expansions.

In the implementation of framework components, only the minimal amount of libraries should be loaded. On one hand, proper organization of the dependencies is the responsibility of the developer. On the other hand it is enforced by the framework architecture, as tool-specific implementations are independent, and functions as entry points calling the framework that uses inversion of control (IoC) without the usage of additional execution environments, such as OSGi.

To alleviate random disturbances, each test case is run several times (e.g. ten times) by the framework and aggregated by the analyzer.

# 3. INTEGRATED CASE STUDIES

The usability of the framework is demonstrated by four examples. Three variations of the previously published Train Benchmark, and a new, soon to be released model comprehension benchmark are integrated into the framework.

## 3.1 Basic Train Benchmark

The first version of the Train Benchmark [9] compares the performance of EMF-IncQuery with Eclipse OCL and its incremental version, the OCL Impact Analyzer in an incremental model validation use case. Instance models are generated from a railway domain, and four hand-written queries (with different complexity) perform model validation tasks. The scenario starts with a model loading phase, where the instance is read from a file, followed by a check phase, where a model validation query is executed (returning constraint violating elements). Afterwards (to simulate a user in front of an editor), multiple (100) edits and rechecks performed. In this case batch, incremental validation time and memory consumption was measured.

One kind of diagrams display execution times as the function of model and query metrics. Fig. 3 shows total execution time for a specific query and scenario in a logarithmic diagram for different tools. On the $x$ axis model size (the number of nodes and edges) is displayed, together with the number of results, and the number of changes in the result set. Although model size is the most influencing performance factor during the load phase, in the check phase, especially for incremental tools other metrics come into the picture as most influencing factors, like the result set size, or the number of variables in a query [5].

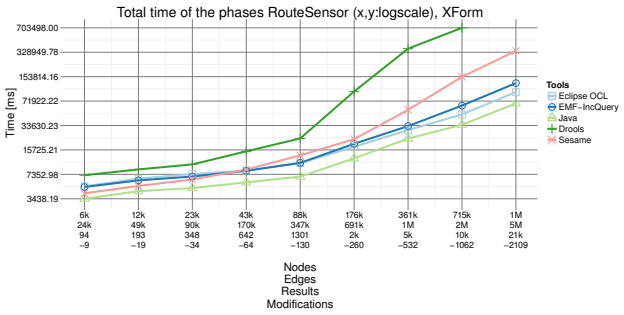



Figure 3: Required time to perform a task.

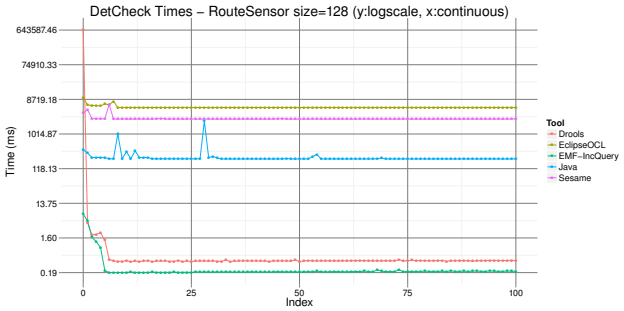


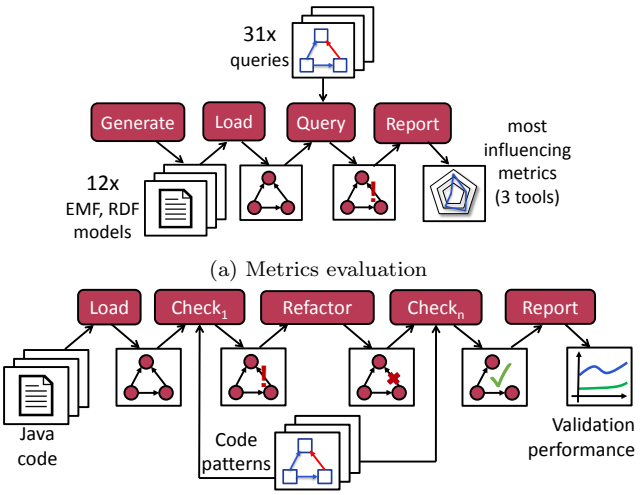

Figure 4: Check time during revalidations.

## 3.2 Extended Train Benchmark

The extended version is available online[2] which introduces new languages: in addition to EMF, RDF and GraphML model formats were added. New tools (Drools, Sesame, 4store and Neo4j) were added, and queries were translated to each tool's native language. From now not all tools have in-memory implementation, some use hard disk as storage, so to lower disk overhead, memory filesystems were used for storage. Also it should be noted that some databases compiled as JARs next to the benchmark code, some database use native server daemons that are also handled by the benchmark execution framework. In this case a new scenario variation is defined, where after the batch validation, larger modification is performed in one edit phase (to simulates automatic model correction), and finally recheck is executed.

As the benchmark framework records every check and edit time subsequently calls can be displayed on a diagram to show its changes. Fig. 4 depicts such a case for tools at a given model size and query. It can be observed that the first query time is almost always the highest, probably due to the lazy loading of classes and tool initialization. Another interesting point for the incremental EMF-IncQuery and Drools tools is around the tenth check, where evaluation times are dropped significantly. As the same queries are executed, this may be attributed to the changed model structure, or to the kicked in JIT compiler. This diagram also shows the required warmup time for each tool, and its changing in stages.

[2] `https://incquery.net/publications/trainbenchmark/`




(a) Metrics evaluation

(b) Code model

Figure 5: Different use-cases of the framework

### 3.3 Model Metrics for Performance Prediction

In the article [5] tools are narrowed down to a basic Java implementation, EMF-IncQuery, and Sesame. However, for a modified metamodel nine new instances were generated (belonging to different edge distributions). The benchmark was extended with 31 queries scaling along 5 query metrics. The goal of this paper was not to compare tool performances, but to identify which metrics influence processing time and memory usage the most. (See Fig. 5a.)

Detailed results are available in the paper, however it can be noted that for the EMF-IncQuery tool the number of matches, for Sesame the number of query variables showed high correlation with the check time, and low correlation of model size metrics that also emphasize considering other aspects than model size.

### 3.4 ITM Factory

The fourth case (inspired by [10]) integrated into the framework is currently under development, and it took another domain from the field of software comprehension. Input of the benchmark are not serialized models, but Java projects. In the first step, source code is read into a software model, transformations are code edits or complex refactor operations. After software modifications, correctness of the code base is validated (Fig. 5b).

In the code modeling case similar investigations can be done, however processing tools should scale in the lines of code (and not in the number of nodes or edges). This also motivates displaying performance as a function of different metrics.

## 4. CONCLUSION

In this paper we proposed MONDO-SAM, a framework that provides common functions required for benchmarking, and MDE-specific scenarios, models, queries and transformations as reusable and configurable primitives. As the main focus, integrated benchmark cases can be characterized by metrics, which enables the reporting module to analyze the scalability of tools against various complexity measures. We demonstrated the versatility of the framework is demonstrated by the integration of previous versions of the Train Benchmark [9, 5] and a new benchmark from the code model domain.

The extensible framework including the APIs, core components and documentated samples is available as open source code from the MONDO Git repository[3].

## 5. REFERENCES

[1] Transformation tool contest. `www.transformation-tool-contest.eu`, 2014.

[2] G. Bergmann, I. Ráth, T. Szabó, P. Torrini, and D. Varró. Incremental pattern matching for the efficient computation of transitive closure. In *Sixth International Conference on Graph Transformation*, volume 7562/2012, pages 386–400, Bremen, Germany, 09/2012 2012. Springer.

[3] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *International Journal On Semantic Web and Information Systems*, 5(2), 2009.

[4] O. Görlitz, M. Thimm, and S. Staab. SPLODGE: Systematic generation of SPARQL benchmark queries for Linked Open Data. In C.-M. et al., editor, *The Semantic Web – ISWC 2012*, volume 7649 of *LNCS*, pages 116–132. Springer Berlin Heidelberg, 2012.

[5] B. Izsó, Z. Szatmári, G. Bergmann, Á. Horváth, and I. Ráth. Towards precise metrics for predicting graph query performance. In *IEEE/ACM 28th International Conference on Automated Software Engineering*, pages 412–431, Silicon Valley, CA, USA, 2013. IEEE.

[6] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 2:1–2:10, New York, NY, USA, 2013. ACM.

[7] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL performance benchmark. In *Proc. of the 25th International Conference on Data Engineering*, pages 222–233, Shanghai, China, 2009. IEEE.

[8] M. Tichy, C. Krause, and G. Liebel. Detecting performance bad smells for henshin model transformations. In B. Baudry, J. Dingel, L. Lucio, and H. Vangheluwe, editors, *AMT@MoDELS*, volume 1077 of *CEUR Workshop Proceedings*. CEUR, 2013.

[9] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An Integrated Development Environment for Live Model Queries. *Science of Computer Programming*, 2014. Accepted.

[10] Z. Ujhelyi, Á. Horváth, D. Varró, N. I. Csiszár, G. Szőke, L. Vidács, and R. Ferenc. Anti-pattern detection with model queries: A comparison of approaches. In *IEEE CSMR-WCRE 2014 Software Evolution Week*. IEEE, 02/2014 2014.

[3] `https://opensourceprojects.eu/git/p/mondo/trainbenchmark`

# Tool Support for Model Splitting using Information Retrieval and Model Crawling Techniques

Daniel G. Strüber, Michael Lukaszczyk, Gabriele Taentzer
Philipps-University Marburg
Department for Mathematics and Computer Science
Hans-Meerwein-Str., 35032 Marburg, Germany
{strueber,lukaszcz22,taentzer}@informatik.uni-marburg.de

## ABSTRACT

To facilitate the collaboration in large-scale modeling scenarios, it is sometimes advisable to split a model into a set of sub-models that can be maintained and analyzed independently. Existing automated approaches to model splitting, however, suffer from insufficient consideration of the stakeholder's intentions and add a significant overhead for comprehending the created decompositions. We present a new tool that aims to create more informed model decompositions by leveraging existing domain knowledge in the form of textual descriptions. From the user perspective, the tool comprises a textual editor for assembling the descriptions and a visual editor for reviewing and post-processing the generated splitting suggestions. We preliminarily evaluate the tool in a case study involving a real-life model.

## Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: Tools; D.2.8 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

## 1. INTRODUCTION

As model-driven engineering is applied in ever-greater scenarios ranging over significant spans in time and space, the maintenance obstacles induced by large models increase in urgency. Large models without a proper decomposition are hard to comprehend, to change, to reuse, and to collaborate on. Even in projects where an initial decomposition is tailored with great care, changing requirements may deem it necessary to refactor for a finer-grained or even orthogonal one. As the manual refactoring of large models is non-trivial and expensive, this problem calls for automation.

Earlier automated approaches to model splitting, such as those presented in [7, 12], suggest techniques based on analysis of strongly connected components or clusters, not accounting for the semantics of the split and the intention for performing it. To address this shortcoming, a recent approach proposed in [11] aims to create model decompositions from existing domain knowledge in the form of textual descriptions: The user provides a set of descriptive texts, each describing one sub-model in the target decomposition. From this input, a splitting suggestion is created using a combined information retrieval and topology analysis approach. The descriptions can be assembled from available requirement or documentation artifacts. However, the input set is not required to be complete: In fact, the approach can support the user in *incrementally discovering* sub-model descriptions.

The contribution of this paper is a tool and supporting semi-automated user process making the outlined splitting technique available to modelers. We have tested it on large meta-models in the magnitude of 100 to 250 classifiers. As design goals, we target usability and extensibility for the splitting of instances of arbitrary meta-models. The remainder of this paper is divided as follows: In Section 2, we briefly illustrate the underlying technique. The user process is shown in Section 3. In Sections 4 and 5, we elaborate on the design goals and implementation. In Section 6, we present a case study preliminarily evaluating the proposed tool and user process. We discuss related work and conclude in Sections 7 and 8.

## 2. BACKGROUND

In this section, we give a brief overview on model splitting as perfomed by our tool. A detailed account is found in [11].

The technique, outlined in Fig. 1, takes three input parameters: The model to be split – in the proposed tool, an EMF meta-model –, a set of textual descriptions of each target sub-model, and a completeness condition. The completeness condition specifies whether the set of sub-model descriptions is complete or partial. The technique creates a set of mappings from model elements to sub-models, calling it *splitting suggestion*. In the case of a complete input set, each element

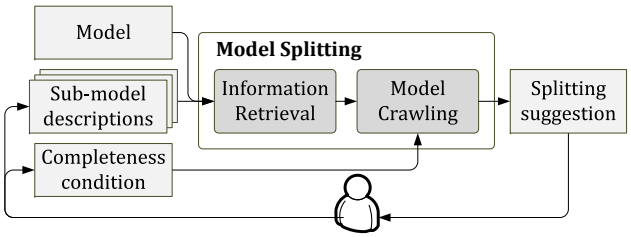



**Figure 1: Underlying model splitting technique.**

*BigMDE'14* July 24, 2014. York, UK

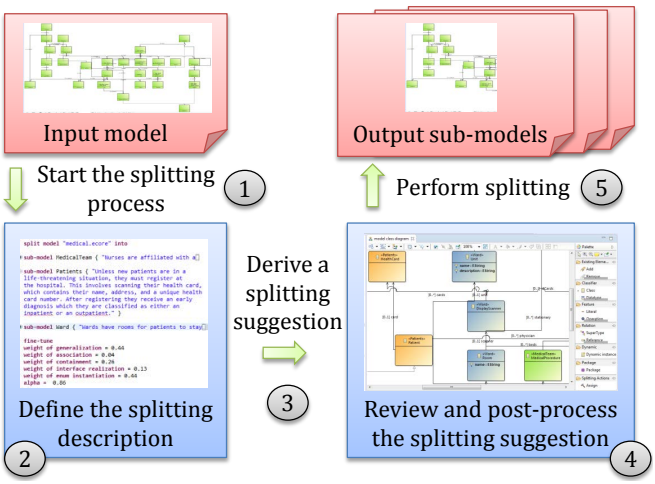



**Figure 2: Overview.**

```
split model "medical.ecore" into

sub-model MedicalTeam { "Nurses are affiliated with a

sub-model Patients { "Unless new patients are in a
life-threatening situation, they must register at
the hospital. This involves scanning their health card,
which contains their name, address, and a unique health
card number. After registering they receive an early
diagnosis which they are classified as either an
inpatient or an outpatient." }

sub-model Ward { "Wards have rooms for patients to stay

fine-tune
weight of generalization = 0.44
weight of association = 0.04
weight of containment = 0.26
weight of interface realization = 0.13
weight of enum instantiation = 0.44
alpha =  0.86
```

**Figure 3: Defining the splitting description.**

is assigned to one sub-model. In the partial case, some elements may remain unassigned. The user can inspect the unassigned elements to discover additional sub-models and describe them, incrementally creating a complete split.

**Information retrieval.** To obtain an initial mapping between the model and the textual sub-model descriptions, we apply an established statistical technique from information retrieval research: Latent Static Analysis (LSA) [8]. For a query (e.g., a sub-model description) over a fixed set of documents (e.g., a set of model element names), LSA scores the relevance of each document to the input query. To compute the scores, queries and documents are represented as vectors and the similarity between the query vector and each document vector is computed – intuitively speaking, the degree in that they point in the same direction. Mathematically, this is calculated in terms of the cosine, yielding a score between 0 and 1. The vector representation is based on a metrics called *term frequency-inverse document frequency (td–idf)*.

**Model crawling.** To create the splitting suggestion, we use the model elements ranked highest by LSA as *seeds*. Starting from these seeds, we crawl the model exhaustively to score each model element's relevance for each target sub-model. Afterwards, each model element is assigned to the sub-model it was deemed most relevant for, ties being broken randomly. Model crawling extends an approach proposed in [9]. The underlying intuition is that of a breadth-first search: We first visit and score the seeds' neighbours, then the neighbours' neighbours, et cetera. Scores of newly accessed elements are calculated based on the scores of previously scored elements. The scoring formula accounts for topological properties, such as the connectivity of newly accessed elements, and semantic implications of the respective relationship types (e.g., in meta-models, containment suggests strong connectivity).

## 3. USER PROCESS

The user process, shown in Fig. 2, comprises two manual tasks (2 and 4) and three automated tasks (1, 3 and 5). The manual tasks rely on human intelligence and domain knowledge. They are facilitated by textual and visual tool support. The automated tasks are triggered by context menu entries.

**(1) Start the splitting process.** Using a context menu entry on the meta-model to be split, the user triggers the creation of a splitting description file. The splitting description is automatically opened in a textual editor, shown in Fig. 3. By default, the file contains a small usage example.

**(2) Define the splitting description.** Using the editor, the user assembles the descriptions of the target sub-models. For a comfortable user experience, the editor provides syntax highlighting, static validation, and folding capabilities. The textual editor is also used for configuration: Adding the keyword `partially` and defining a numerical threshold, the user can set the completeness condition in order to obtain a partial split. Furthermore, the user can fine-tune internal parameters used during the execution of the underlying technique. In Fig. 3, the weights assigned to different relationship types and the *alpha* exponent that shapes the scoring function are modified. However, parameter tuning is an optional feature: In [11], we identified a default combination of parameter values that, when applied to six independent class models, achieved an average accuracy of 80% in comparison to hand-tailored decompositions.

**(3) Derive a splitting suggestion.** Using a context menu entry on the splitting description file, the user triggers the automated creation of a splitting suggestion. A splitting suggestion comprises a set of assignment entries, each holding a link to a model element, a link to a target sub-model, and the relevance score. To compute the splitting suggestion, the technique outlined in Section 2 is applied. The splitting suggestion is persisted to the file system.

**(4) Review and post-process the suggestion.** To obtain visual access to the splitting suggestion, the user can now open the model in a model editor. The user activates a dedicated layer called *model splitting*. This action triggers the color-coding of model elements corresponding to the splitting suggestion, shown in Fig. 4. As further visual aid, the assignment of a model element is also displayed textually above its name. For post-processing, the user may want to change some assignments for model elements that were not assigned to the proper target sub-model. This is done using

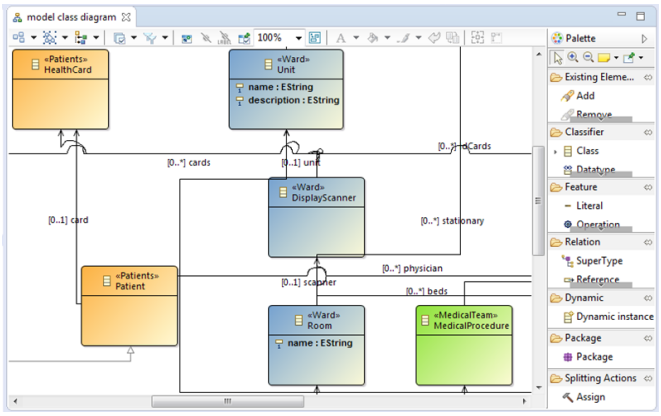



**Figure 4: Reviewing and post-processing the splitting suggestion.**

the palette tool entry *Assign*. When the user reassigns a model element, the respective entry in the splitting suggestion is automatically updated. It is worth mentioning that if the user is not satisfied with the results, he or she may iterate Steps 2 to 4 as often as required, tweaking the descriptions and parameter settings. One important scenario for this is the discovery of new sub-models: The user can set the completeness condition to *partial* in Step 2 which leads to some model elements not being assigned in Step 3. The user inspects these elements in Step 4 to create new sub-model descriptions.

**(5) Perform splitting.** Given that the user is satisfied with the post-processed splitting suggestion, the actual splitting can be triggered by the user. The user may choose from two context menu entries: One for splitting the input model into multiple physical resources, the other for splitting it into sub-packages within the same resource.

# 4. DESIGN GOALS

In this section, we shortly discuss design goals that were fundamental in the design of the proposed tool.

**Extensibility.** The underlying technique possesses an innate extensibility that should be carried over to the end-user. It is applicable to models conforming to arbitrary meta-models, given that they fulfill two properties: (i) Model elements must have meaningful textual descriptions that a splitting description can be matched against. (ii) Except for trivial reconciliation, constraints imposed by the meta-model may not be broken in arbitrary sub-models. We address this design goal by using a framework approach: To customize the tool for a new meta-model, the user subclasses a set of base classes. For instance, to define how input models are converted to a generic graph representation used during crawling, they subclass a class named *GraphBuilder*.

**Usability.** The design of the tool is informed by *Cognitive Dimensions*, a framework for the human-centered design of languages and tools [6]: Providing an editable visual layer on top of a standard editor is a major step towards *visibility* – visual accessibility of elements and their relations – and away from *viscosity* – resistance to change. *Closeness of mapping* is implemented by a domain-specific language for splitting descriptions with custom editor support. *Premature commitment* is inhibited and *progressive evaluation* is promoted by providing an incremental process that allows tweaking with input values while receiving rapid feedback. For *traceability*, our file-based approach to user input allows to keep the splitting description and use it later, e.g., for documentation purposes.

# 5. IMPLEMENTATION

Eclipse Modeling Framework [10] is the de-facto reference implementation of the EMOF modeling standard. Consequently, it was natural for us to design the new tool as an extension for EMF. As such, it can be plugged into an existing Eclipse installation without further effort. For the splitting description editor, we leveraged the powerful code generation facilities of Xtext [5]. We defined a simple domain-specific language for splitting descriptions. The editor with its syntax highlighting and code completion features was fully generated by Xtext. For customization, we added a couple of checks (e.g., forbidden characters, uniqueness of sub-model names). The visual splitting layer is an extension of EcoreTools 2.0 [2] which is based on the Sirius [4] framework and, as of June 2014, determined to be part of the new Eclipse release Luna 4.4. We used this new technology as we benefit from its support for multiple viewpoints, allowing us to tailor a *splitting viewpoint* to our needs.

# 6. CASE STUDY

In a case study, we investigated two research questions: (RQ1) How efficient is the proposed tool in comparison to manual splitting? (RQ2) Is the proposed tool usable?

## 6.1 Subjects and Task

**Model**: *Extended Joomla-Specific Language (eJSL)* is a meta-model for web applications based on the Joomla content management system [3]. It comprises 116 classes, 39 enumerations, 176 enumerated attributes, 41 generalizations, 145 containment references, and 47 plain references. eJSL was designed by a doctoral student affiliated with our research group we shall refer to as X. X has significant experience in modeling language design. Previous to our work, X manually split eJSL into five sub-models, calling them *Pages*, *Content*, *Menu*, *User*, and *Configuration*. According to his account, he invested a significant effort that spanned, among other duties, over the course of two weeks. He printed the diagram on paper, cut and reassembled fragments. Afterwards, he assigned colors to model elements in the diagram editor and layouted them by hand.

**Task**: We instructed another software engineer, referred to as Y, to decompose eJSL using the tool. Y is a doctoral student with significant experience in modeling language design, but unrelated to eJSL and model splitting. We asked X to provide the required domain knowledge in the form of descriptive texts briefly explaining his intuitions for the hand-tailored decomposition. The descriptions, each consisting of 85 words on average, were handed to Y in a text document. The task given to Y was to create *a decomposition that faithfully reflects the separation of concerns proposed by the textual descriptions*. We briefly instructed Y in the usage of the tool based on the example shown in Fig. 3 and 4 and encouraged him to make use of post-processing.

## 6.2 Results

**Efficiency.** To approach (RQ1), we define *efficient* as *requiring a minimal amount of time to create an accurate result*. Positing the hand-tailored split as perfectly accurate, we measured accuracy of the tool-supported split in terms of average F-measure, considering both precision and recall. Accuracy was determined before and after post-processing: During review of the initial splitting suggestion *S1*, Y reassigned five model elements to create the final suggestion *S2*. From S1 to S2, precision increased from 82% to 86% and recall from 84% to 88%, determining a rise in F-measure from 83% to 87%. It took Y five minutes to create *S1*. The reviewing and post-processing that brought the 4% gain took further 55 minutes. Consequently, in terms of extrapolated overall amount of time, tool-supported splitting outperformed manual splitting.

**Usability.** To approach (RQ2), we conducted an informal interview. Y perceived the user process as comprehensible, the description editor as easy to use and the color-coding as useful. An activity found crucial during post-processing was examining the direct neighbours of a model element. Y perceived this task as cumbersome: He often had to navigate for edge targets outside the visible scope. For future work, we aim at dedicated support for this activity: On selection, neighbourhood information should be instantly available in a tool-tip displaying the names of adjacent elements. One further suggestion by Y, the color-coding of edges, directly made it into the current version. Y also invested considerable time in layouting, i.e., aligning the color-coded model elements into groups – an activity outside of the scope of this work. It is an interesting challenge to devise a layouting algorithm that aligns the sub-models of a model as clusters. Inspection of the false positives and negatives in S2 revealed that 50% of them concerned enumerations, the other 50% concerning classes. Y pointed out that enumerations were hard to relate to classes visually as they are not connected by edges. We consider representing enumerated attributes as edges rather than class members in future work.

## 6.3 Validity

Threats to external validity – or generalizability – are the size of the input model and the size of the test group. It remains a question left to future work whether our tool scales for meta-models of significantly more elements. However, an analysis of publicly available meta-models[1] indicates the input model size to be typical for large meta-models demanding an adequate decomposition. The test group size indeed precludes claims for generality, but allows to provide tentative evidence for critical design weaknesses and benefits. A potential threat to internal validity – or freeness from systematic error – is the flow of information from the control group to the test group. To mitigate this threat, we ascertained in consultation with X that the textual descriptions in vagueness and level of detail represented the intuitions for splitting *before* the manual split was executed.

# 7. RELATED WORK

In this section, we discuss related tooling. A survey of work related to the underlying *approach* is provided in [11].

[1]http://www.emn.fr/z-info/atlanmod/index.php/Ecore

In the Democles model composer [1], the user can iterate the lattice of all permitted decompositions by unfolding entries in a tree-like wizard. Graphical presentation of a split is provided by an add-on graph visualization library. However, this visualization is read-only and not integrated with a modeling editor, ruling out the re-assigning of model elements for post-processing as supported by the new tool.

The splitting tool proposed in [12] makes classic clustering algorithms available for EMF models. It provides a wizard for the selection and customization of algorithms. However, except for numerical input parameters, the user cannot influence the generated results. The tool provides a tree-based editor for the reassigning of model elements to target submodels, but does not present any visual feedback.

# 8. CONCLUSION

In this paper, we present a tool for the splitting of large meta-models. The tool provides a textual editor that allows defining the desired target sub-models by means of textual descriptions. It generates a splitting suggestion that can be reviewed and post-processed in a visual editor. Based on the splitting suggestion, the input model can be automatically split either into multiple resources or packages within one resource. The tool is open source and can be found, along with the models mentioned in this paper, at `https://www.uni-marburg.de/fb12/swt/forschung/software`. In the future, we plan to apply the technique on other models than class models, deeming it necessary to account for constraints.

# 9. REFERENCES


[1] Democles. http://democles.lassy.uni.lu/, May 2011.

[2] Ecoretools 2.0. http://www.eclipse.org/ecoretools/, May 2014.

[3] Joomla. http://www.joomla.org/, May 2014.

[4] Sirius. http://www.eclipse.org/sirius/, May 2014.

[5] Xtext. http://www.eclipse.org/xtext/, May 2014.

[6] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[7] P. Kelsen, Q. Ma, and C. Glodt. Models within models: Taming model complexity using the sub-model lattice. *Fundamental Approaches to Software Engineering*, pages 171–185, 2011.

[8] T. K. Landauer, P. W. Foltz, and D. Laham. An Introduction to Latent Semantic Analysis. *Discourse Processes*, (25):259–284, 1998.

[9] M. P. Robillard. Automatic Generation of Suggestions for Program Investigation. In *Proc. of ESEC/FSE-13*, pages 11–20, 2005.

[10] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[11] D. Strüber, J. Rubin, G. Taentzer, and M. Chechik. Splitting models using information retrieval and model crawling techniques. *Fundamental Approaches to Software Engineering*, pages 47–62, 2014.

[12] D. Strüber, M. Selter, and G. Taentzer. Tool support for clustering large meta-models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 7. ACM, 2013.

# Automated Analysis, Validation and Suboptimal Code Detection in Model Management Programs

Ran Wei
University of York
Deramore Lane, Heslington
York, United Kingdom
ran.wei@york.ac.uk

Dimitrios S. Kolovos
University of York
Deramore Lane, Heslington
York, United Kingdom
dimitris.kolovos@york.ac.uk

## ABSTRACT

As MDE is increasingly applied to larger and more complex systems, the models that MDE platforms need to manage can grow significantly in size. Additionally, model management programs that interact with such models become larger and more complicated, which introduces further challenges in ensuring their correctness and maintainability. This paper presents an automated static analysis and validation framework for languages of the Epsilon platform. By performing static analysis on model management programs written in the Epsilon languages, this framework aims to improve program correctness and development efficiency in MDE development processes. In addition, by applying analysis on model management programs, sub-optimal performance patterns can be detected early in the development process and feedback can be provided to the developers to enable efficient management of large models.

## 1. INTRODUCTION

Model Driven Engineering (MDE) aims at raising the level of abstraction at which software is developed, by promoting models into first-class artefacts of the development process. For MDE to be applicable in the large and complex systems, a set of model management tools and languages are needed to enable developers to manage their models in an automated and efficient manner. Typically, model management tasks include validation, transformation, comparison, merging and text generation [7].

The Epsilon platform [7] is a platform which provides a broad range of model management languages built on top of an extensible common imperative model query and modification language called the Epsilon Object Language (EOL). EOL is an interpreted language that supports optional typing of variables, operations, and operation parameters. This provides a high degree of flexibility (for example, it enables duck typing [6]) and eliminates the need for explicit type casts, as typing-related information is currently only considered at runtime. On the other hand, the absence of a static analyser for EOL and the languages that build atop it, implies that a wide range of genuine errors that could be detected at design time are only detected at runtime.

Beyond detecting type-related errors, a static analyser could also be used to support advanced analysis capabilities, such as code detection, to improve the performance of model querying and transformation programs, particularly in the context of processing large models.

The remainder of the paper is organised as follows. In Section 2 we briefly motivate the need for static analysis capabilities in model management languages, in particular the Epsilon platform. In Section 3 we present the architecture of a static analysis framework for the Epsilon platform and a static analyser for the Epsilon Object Language. In Section 4 we present a sub-optimal code detection facility developed atop the static analysis framework. In Section 5 we discuss preliminary evaluation of our work. In Section 6 we discuss related work and in Section 7 we conclude and provide directions for further work.

## 2. BACKGROUND

MDE allows developers to construct models which abstract away from technical details, using concepts closely related to the domain of interest to reduce accidental complexity. The constructed models are then transformed into part (or all) of the target system under development. Whilst Model Transformation is considered to be the heart and soul of Model Driven Engineering [12], other model management operations are equally important. Typically, such operations include model validation, model merging, model comparison, etc.

### 2.1 Epsilon

Epsilon is a mature and well-established family of interoperable languages for model management. Languages in Epsilon can be used to manage models of diverse metamodels and technologies. The architecture of the Epsilon platform is depicted in Figure 1. At the core of Epsilon is the Epsilon Object Language (EOL) [9]. EOL is an imperative language which reuses a part of the Object Constraint Language OCL) but provides additional features such as multiple model access, statement sequencing and groupings, uniformity of function invocation, model modification, debugging and error reporting. Although EOL can be used as a general purpose model management language, its primary aim is to be reused in task-specific languages. Thus,


*BigMDE* '14, July 24, 2014 York, UK

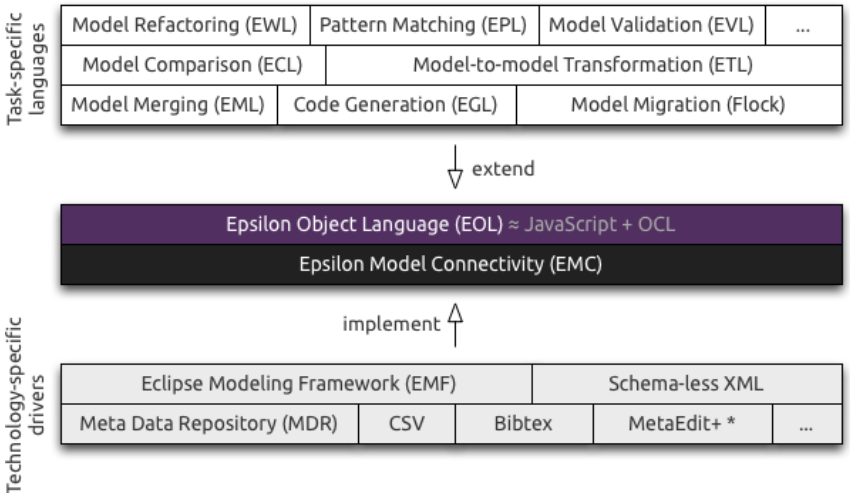



**Figure 1: The basic structure of the Epsilon platform.**

by extending EOL, a number of task-specific model management languages have been implemented, including those for model transformation (ETL), model comparison (ECL), model merging (EML), model validation (EVL), model refactoring (EWL) and model-to-text transformation (EGL).

Epsilon is metamodeling-technology agnostic [7], models written in different modelling languages can be managed by Epsilon model management languages via the Epsilon Model Connectivity (EMC) layer, which offers a uniform interface for interacting with models of different modelling technologies. Currently, EMC drivers have been implemented to support EMF [13], MDR, Z specifications in LaTeX using CZT and plain XML. Epsilon is an Eclipse project and is widely used both in the academia and the industry[1].

## 2.2 Epsilon and static analysis

EOL supports optional typing of variables, operations and operation parameters. The listing below demonstrates the flexibility that this delivers using as an example, a program that operates on an Ecore model. In line 1, an operation called *getName()* is defined, its *context type* is Any, which means that the operation can be invoked on objects/model elements of any type. In line 2, the keyword *self* refers to the object on which the operation is called, for example, in the statement *o.getName()*, *self* refers to *o*. Thus in line 2, the operation checks if the object is an instance of ENamedElement, if it is, it will return *self.name*. A typical static analyser would complain in line 3 that "self.name" does not exist because the type of *self* is declared as *Any*, not *Ecore!ENameElement*. However, this will never be a problem at run-time due to the *if* condition in line 2.

```
1  operation Any getName() {
2    if(self.isTypeOf(Ecore!ENamedElement))
3      return self.name;
4    else return "Unnamed";
5  }
```

[1]http://eclipse.org/epsilon/users/

On the other hand, some genuine errors can also remain hidden until runtime without the help of static analysis. The listing below demonstrates an example of a genuine error. In line 1, a variable named *o* is defined to be of type *Ecore*!*EClass*, and in line 2, *o* is assigned the value of a random EClass in the Ecore model. In line 3, the program tries to print the *value* property of *o* which does not exist according to the Ecore metamodel. As such, an error will be thrown at runtime.

```
1  var o : Ecore!EClass;
2  o = Ecore!EClass.all.first();
3  o.value.println();
```

As the size of such programs grow, locating genuine errors becomes an increasingly difficult task. For example, the EOL program that underpins the widely-used EuGENia tool [10] consists of 1212 lines of code, that query and modify 4 models that conform to 4 different metamodels concurrently. Performing code review on this code for genuine error detection is a time consuming process. Additionally, performing changes is also difficult, as developers have to manually identify and manage dependencies and relations between building blocks (*operations* for example) of such programs. Such tasks require effort and time [15]. For instance, to delete an operation named *F1* it is necessary to know if it is invoked by any other operations or code blocks. Manually performing this task is error-prone.

Since model management programs interact with models, performing static analysis on model management programs can also help identify sub-optimal performance patterns in the process of accessing models. Analysis can also help comprehend the model management programs. For example, coverage analysis can be performed to find out which parts of the metamodel of particular models are accessed, and test cases can be generated based on this comprehension to better test a model management program. With the potential benefits mentioned above, we propose and implement a static analysis framework for Epsilon.

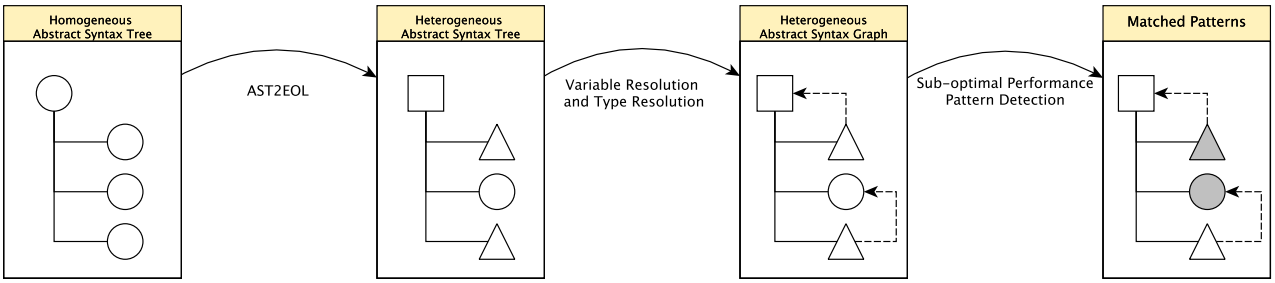



**Figure 2: Detecting sub-optimal performance patterns from Abstract Syntax Trees.**

# 3. TOWARDS A STATIC ANALYSIS FRAMEWORK FOR EPSILON

In this section we discuss the proposed static analysis framework in detail. The general idea of our approach can be illustrated by Figure 2. We first transform the Homogeneous Abstract Syntax Tree of an EOL program into a Heterogeneous Abstract Syntax Tree, we then apply resolution algorithms (including variable resolution and type resolution) to derive a Heterogeneous Abstract Syntax Graph, with its elements connected to each other. We then perform pattern detection to detect sub-optimal code. The aim of the framework is to provide a generic static analysis facility for all languages of the Epsilon platform. Since the core language of the Epsilon platform is EOL, we first develop a static analyser for EOL.

## 3.1 Deriving Heterogeneous Abstract Syntax Trees

Currently, Epsilon provides an ANTLR [7] based parser for EOL. The ANTLR parser produces homogeneous Abstract Syntax Trees (AST) with each node containing a text and an id, which the EOL execution engine consumes. To facilitate static analysis at a high level of abstraction, our first step was to define an EMF-based metamodel for EOL and to transform these homogeneous ASTs to models (heterogeneous trees) that conform to the EOL metamodel.

As EOL is a reasonably complex language, we only introduce the basic and novel parts of the EOL metamodel and the EOL standard library. Figure 3 lists a number of basic building blocks that constitute an EOL program. The fundamental element of the EOL metamodel is the *EolElement* metaclass, as all other metaclasses in the EOL metamodel directly or indirectly extend it, and contains information related to the line and column numbers of the respective text in an EOL program for traceability purposes. A *Program* contains a number of *Import*(s), which are used to import other programs. A *Program* also contains a number of *OperationDefinition*(s) which define additional operations/-functions on existing types. A *Program* contains a *Block*, which contains a number of *Statement*(s). *Expression* is also a fundamental element which is generally contained in *Statement*(s) and other *EolElement*(s). For each of the *Expression*, there is a *Type* associated to it. The type of the *Expression* is generally unknown at the time the source code of a program is parsed into an EOL model, but is resolved later in the type resolution process. In order to run an EOL program that involves processing models, Epsilon currently requires the user to select the required models/metamodels via a user interface at runtime. To facilitate accessing models at design-time for static analysis, we introduce the *ModelDeclarationStatement* to declare references to models in the source code. The syntax of a model declaration statement is as follows.

```
1  model library driver EMF {
2    nsuri = "http://library/1.0"
3  };
```

Like Epsilon, the static analysis framework is also technology-agnostic. As such, beyond the model's local name, a model declaration statement defines the type of the model in question (in this case EMF), as well as a set of model-type-specific key-value parameters (in this case nsuri = *http* : *//library*/1.0) that the framework can use to obtain the model's metamodel. We have currently implemented facilities to support models defined using EMF and plain XML. In the future we plan to extend the static analysis framework with support for other types of models supported by Epsilon, such as models defined in MDR, spreadsheets, etc.

With the metamodel defined, we developed a facility which transforms the ANTLR based ASTs into models that conform to the EOL metamodel. It should be noted that at the stage of AST to EOL model transformation, declared models are not inspected. So at this stage, for the statement

```
1  var book : Book
```

it is only known that variable *book* is of type *ModelElement-Type* whose *elementName* is *Book*. Later in the type resolution process, this information is used against declared models so that the *Book* type can be resolved.

Comparing with our current approach, an alternative approach would have been to redefine EOL's grammar using a toolkit such as Xtext or EMFText which can automate the source-code to model transformation process but we have opted for an intermediate transformation instead in order to reuse Epsilon's robust and proven parsers.

## 3.2 Deriving Heterogeneous Abstract Syntax Graphs

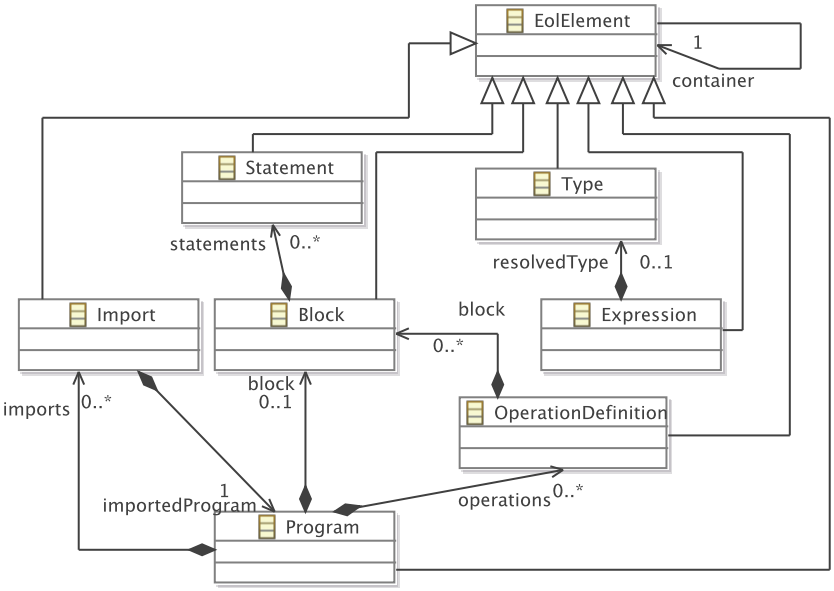



**Figure 3: The basic structure the EOL metamodel.**

With the EOL metamodel and the AST to EOL transformation defined, the next step of the process involves linking elements of the EOL model (heterogeneous tree) constructed in the previous phase to derive an abstract syntax graph. We have created several facilities to achieve this.

### 3.2.1 EOL Visitor

To facilitate the traversal of different elements in the EOL model and to support extensibility, we developed a facility which generates Java visitor classes from Ecore metamodels. We then generated visitor classes from the EOL metamodel which provide a general mechanism to traverse all elements in an EOL model. The EOL Visitor serves as the infrastructure of the static analysis framework, as all other facilities in the static analysis framework extend the EOL visitors to implement functionalities. The EOL visitor also provides high extensibility as new analysis mechanisms can be implemented simply by extending the EOL visitor.

### 3.2.2 Variable Resolver

The first phase of the static analysis on an EOL program involves resolving identifiers to their definitions. Context-free identifiers in EOL can refer to 1) declared variables/operation parameters and 2) undeclared variables provided by the execution engine at run-time. For declared variables, the variable resolver establishes links between the variable declaration and its references. For example, in line 1 of the listing provided below, a variable named $a$ is declared. In line 2, $a$ is referenced. The variable resolver will establish a link between the reference in line 2 and the declaration in line 1.

```
1  var a : Integer;
2  a = 10;
```

Variable resolution also applies to parameters in operation definitions. In the following listing, the variable resolver will establish a link between the reference of the parameter *toPrint* in line 2 and its declaration in line 1.

```
1  operation definedPrint(toPrint: String) : String {
2    toPrint.println();
3  }
```

There are also some implicit variables which are not declared by the developer but are rather provided by the execution engine. For example, the keyword $self$ in an operation definition refers to the object/model element on which the operation is invoked. The following listing demonstrates how $self$ is used. The variable resolver will establish a link between $self$ and the object on which $printSelf$ is invoked.

```
1  operation Any printSelf() {
2    self.println();
3  }
```

It is important to note that at the stage of variable resolution, model element types are not resolved.

### 3.2.3 Type Resolver

In EOL there are several built-in primitive types (*Boolean*, *Integer*, *Real*, *String*) and collection types (*Set*, *Bag*, *Sequence* and *OrderedSet*). There is also a built-in *Map* type and the *Any* type. These types are all subclasses of *Type* in the EOL metamodel. The resolution of the above types is performed during the heterogeneous abstract syntax tree derivation. There is also a subclass of $Type$ in the EOL metamodel called $ModelElementType$ which includes typing information regarding models defined using different technologies. Such typing information should be determined by

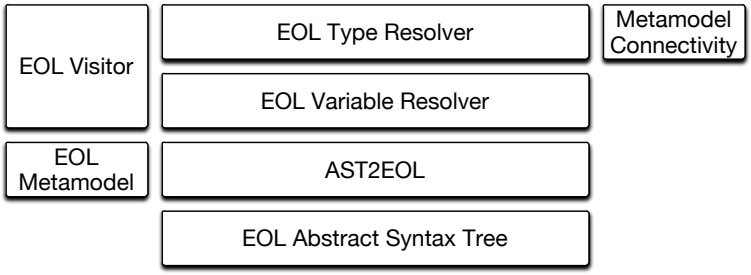



**Figure 4: The architecture of the static analysis framework**

accessing the corresponding models.

To support accessing metamodels at design-time, we introduced *ModelDeclarationStatement*s, which are flexible to support models defined in different modelling technologies. A model declaration defines the model's name, aliases, the modelling technology (driver) that the model conforms to, and a set of driver-specific parameters. The listing below is an example of *ModelDeclarationStatement*; it declares a model named *library* with alias *l* and its driver to be *EMF*, it then specifies the EMF-specific namespace URI (*nsuri*) of the model so that the analyser knows how to locate and access its metamodel.

```
1  model library alias l driver EMF {nsuri = "http://
       library/1.0"};
```

To facilitate uniform analysis of the structural information of models of diverse modelling technologies, the static analysis framework needs to convert type-related information from different modelling technologies into a common representation. Instead of inventing a new representation, we have decided to use EMF's Ecore. As such, the static analysis framework provides a modular architecture where pluggable components are responsible for transforming different types of model declarations into Ecore EPackages. For different modelling technologies:

- For EMF models, return the registered EPackage by looking up the metamodel nsURI property of the model declaration.
- For plain XML, construct an EPackage by analysing the contents of a sample model file specified by respective the model declaration parameter.

We have developed drivers for EMF models and plain XML files, and a common interface which allows new drivers for different modelling technologies to be integrated with the static analysis framework. By accessing models/metamodels, the type resolver is able to resolve types with regards to models/metamodels.

The variable resolver and type resolver constitute the infrastructure of the static analysis framework for the Epsilon languages. The infrastructure is depicted in Figure 4. The EOL Abstract Syntax Tree layer is provided by the EOL engine, the AST2EOL layer uses the AST and the EOL metamodel to translate the AST to an EOL model. The EOL Variable Resolver and the EOL Type Resolver, both make use of the EOL Visitor and the Metamodel Connectivity layer (which is used to read metamodels) to establish a fully type-resolved EOL model.

The static analysis infrastructure can be easily extended. As proof of concept, we have also implemented all of the aforementioned facilities for the Epsilon Transformation Language. We extended the EOL metamodel to create an ETL metamodel, with the ETL metamodel, we created the ETL visitor facility; we extended the AST2EOL to create a AST2ETL facility; we extended the EOL variable resolver and type resolver to create ETL variable and type resolvers. The EOL and ETL static analysers can be found under the Epsilon Labs open-source project [1].

# 4. SUBOPTIMAL CODE DETECTION

Rule-based model transformation languages usually rely on query or navigation languages for traversing the source models to feed transformation rules with the required model elements. In [11], the authors suggest that in complex transformation definitions, a significant part of the transformation logic is devoted to model navigation. In the context of large-scale MDE processes, models can contain several millions of elements. Thus, it is important to retrieve desired elements in an efficient way. On top of the static analysis framework, we have built a facility which is able to detect sub-optimal performance patterns when navigating and retrieving model elements. This facility performs pattern matching to detect potential computationally heavy code in EOL (and potentially all Epsilon languages). It does so by matching patterns defined in the Epsilon Pattern Language (EPL) [8] against fully resolved EOL abstract syntax graphs.

The structure of this facility is depicted in Figure 5. The *SubOptimalDetector* has a *EOLModel* as input to perform the detection; it makes use of the *EPLEngine* of the Epsilon platform to derive Abstract Syntax Trees, it has a set of defined *EPLPattern*s (.epl scripts) using EPL, and a logging facility (*LogBook*) to keep the warning messages it generates for pattern matches.

In this section, we present the sub-optimal detection facility. We provide several examples that illustrate potential sub-optimal performance patterns in the context of large scale model manipulation. We then present and explain a sub-

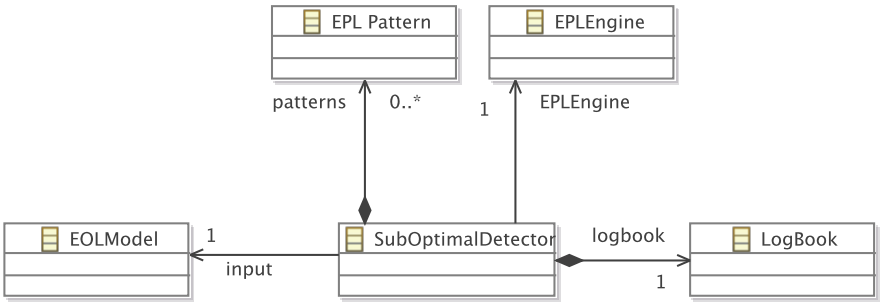



**Figure 5: The structure of the sub-optimal performance detection facility**

optimal performance pattern defined in EPL. It should be noted that this facility targets EOL programs, however, it can be easily extended to cope with programs written in other Epsilon languages as discussed earlier.

The examples we present are all based on a simple *Library* metamodel illustrated in Figure 6. The Library metamodel contains two simple metaclasses, *Author* and *Book*. An *Author* has a *first_name*, a *surname* and a number of published *Book*s where a *Book* has a *name* and an *Author*. The association between *Author* and *Book* is bidirectional, they are *books* and *authors* respectively.

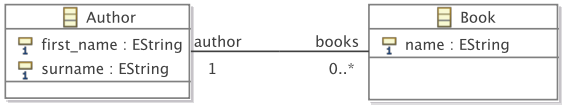



**Figure 6: The Library metamodel**

## 4.1 Inverse navigation

A frequent operation in EOL is to retrieve all model elements of a specific type by using the *.all* property call which can be a computationally heavy operation to perform as models grow in size. By analysing the metamodel of the model under question, bidirectional relationships between model elements can be used to avoid such heavy computations.

```
1  var a = Author.all.first;
2  var books = Book.all.select(b|b.author = a);
3  var aBook = Book.all.selectOne(b|b.author = a);
```

The listing above demonstrates a potential performance bottleneck. In line 1, an *Author* is retrieved from the model. In line 2, all instances of type *Book* are retrieved and then a conditional *select* is performed to find the books that are written by *Author* 'a'. However, since the relationship between *Author* and *Book* is bidirectional, this can be replaced by the (more efficient) statement:

```
1  var books = a.books;
```

Thus the complexity of the operation *all* is reduced from $n$ to 1 given that $n$ is the number of *Book*s in the model under question. It is also the case for the *selectOne* operation in line 3, which can be rewritten as:

```
1  var aBook = a.books.first();
```

## 4.2 Compound select operations

Another computationally-heavy pattern is the presence of compound *select* operations on the same collection.

```
1  var authors = Author.all.select(a|a.first_name =
2    'William').select(a|a.surname = 'Shakespeare');
```

**Listing 1: a potential performance overhead using compound select operations**

Listing 1 demonstrates such operations. In line 1, all of the *Author*s are retrieved first, then a *select* operation is performed to select all *Author*s whose first_names is *William*, then another *select* operation is performed to select all *Author*s whose surname is *Shakespeare*. The complexity of this operation is $n^2$ given that $n$ is the number of *Author*s in the model under question. However, the condition of both the *select* operations can be put together to form a single *select* operation. And the statement above can be written as

```
1  var authors = Author.all.select(a|a.first_name =
2    'William' and a.surname = 'Shakespeare');
```

the complexity of this operation is therefore $n$ as the collection of the *Author*s is only traversed once.

## 4.3 Select operation on unique collections

Performing *select* operations on unique collections (sets) can sometimes be inefficient depending on the *condition* of the *select* operation.

```
1  var authorOne = Author.all.first;
2  var authorTwo = Author.all.last;
3  var bookOne = authorOne.books.first;
4  var bookSet : Set(Book);
5  bookSet.addAll(authorTwo.books);
6  bookSet.select(b|b = bookOne);
```

**Listing 2: Select operation on unique collection**

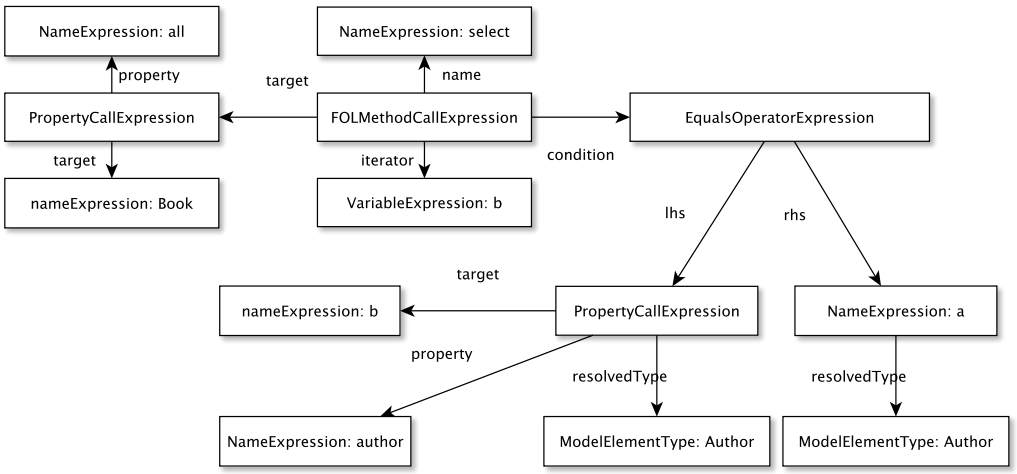



**Figure 7: The model representation for** $Book.all.select(b|b.name = a)$

Listing 2 demonstrates an inefficient and computationally expensive *select* operation. In Line 1 and 2, two *Author*s are retrieved from the model; in line 3, a *Book* is retrieved from *authorOne*'s publications; in line 4, a *Set* called *bookSet* is created and in line 5, all of the *Book*s that *authorTwo* published are added to *bookSet*. In line 6 the *select* operation iterates through all of the books in the *bookSet* and find the ones that match the *bookOne*. However, the *bookSet* is a unique collection, which means that all of the elements in it only appear once. Therefore, it is not necessary to perform a *select* operation but rather a *selectOne* operation, as the *select* operation would return at most one result eventually. The complexity of the *select* operation is $n$ given that $n$ is the number of books that *authorTwo* published; If the *select* operation is replaced with *selectOne*, the complexity of it would be 1 for the best case scenario and $n$ for the worst case scenario ($n/2$ for the average case).

## 4.4 Collection element existence

In some cases, checking existence of an element inside a collection can be written in inefficient ways.

```
1  if(Book.all.select(b|b.name = "EpsilonBook")
2    .size() > 0) {
3    "There is a book called EpsilonBook".println();
4  }
```

**Listing 3: Collection element existence**

Listing 3 demonstrates such a scenario. In line 1, the condition of the *if* statement retrieves all instances of *Book*, then selects the ones with the name *EpsilonBook*, and calculates the size of it then evaluates if the size is greater than 0. This operation eventually checks for the existence of a book named *EpsilonBook*. Thus, this operation can be more efficiently re-written as:

```
1  Book.all.exists(b|b.name = "EpsilonBook")
```

## 4.5 Select the first element in a collection

Listing 4 demonstrates another example of sub-optimal EOL code.

```
1  var anEpsilonBook = Book.all.select(b|b.name =
2    "EpsilonBook").first();
```

**Listing 4: Select an element in a collection**

In line 1, a *select* operation is performed on all of the instances of *Book* to filter out the books with the name 'EpsilonBook', then a $first$ operation is performed to select the first one of the collection returned by *select*. This can be more efficiently re-written as:

```
1  var anEpsilonBook = Book.all.selectOne(b|b.name =
2    "EpsilonBook");
```

to avoid traversing all of the instances of *Book*.

## 4.6 A sub-optimal performance pattern

In this section we present a sub-optimal performance pattern which is written in the Epsilon Pattern Language (EPL). To understand how this pattern works, it is first important to understand what is contained in an EOL model for a certain EOL program.

### *4.6.1 Understanding an EOL model*

Figure 7 illustrates a fragment of an EOL model which represents the statement below.

```
1  Book.all.select(b|b.author = a);
```

Firstly, invocations of the *select*() operation in the EOL metamodel are represented by the $FOLMethodCallExpression$ (FirstOrderLogic method call) metaclass; it has a *name* (an instance of $NameExpression$) and an *iterator* (an instance of $VariableExpression$). In this case, the *name* is 'select' and the *iterator* is 'b'.

The *select* operation has a *condition*, in this case, it is an instance of *EqualsOperatorExpression*. The lhs (left hand side) of it is an instance of *PropertyCallExpression*, whose *target* (an instance of NameExpression) is 'b' and *property* (an instance of NameExpression) is 'author'; the rhs (right hand side) of it is 'a' (an instance of NameExpression). Both the lhs and rhs of the *EqualsOperatorExpression* have resolvedTypes, in this case, they are both *Author* (instances of *ModelElementType*s).

The *target* of the *FOLMethodCallExpression* is an instance of *PropertyCallExpression* with its *target* as *Book* (an instance of NameExpression) and its *property* as *all* (an instance of NameExpression). The types of these expressions, altogether with some irrelevant details are omitted for the purpose of the discussion.

### 4.6.2 The EPL pattern

In Listing 5, we define an EPL pattern to match occurences of the pattern described above. In lines 2-6, a *guard* is defined to look for a *FOLMethodCallExpression* the name of which is either 'select' or 'selectOne'; the type of the condition should be *EqualsOperatorExpression*; its target should be an instance of *PropertyCallExpression*; and the property of the *PropertyCallExpression* should be 'all'.

In lines 8-10, a *guard* is defined to look for an instance of *EqualsOperatorExpression* in the condition of the FOLMethodCallExpression found previously, the *lhs* of which should be an instance of *PropertyCallExpression*.

Lines 12-14 specify that the *resolvedType* of the *lhs* should be an instance of *ModelElementType*. In lines 16-18, it specifies that the *resolvedType* of the *rhs* should be an instance of *ModelElementType*. In lines 20-24, it specifies that the type of the *lhs* and the *rhs* should be the same.

Lines 26-37 perform the match of the pattern. This part firstly fetches the *EReference* from the *lhs* of the condition (in this case, 'b.author', it is an *EReference* because as previously discussed, all metamodels are converted to EMF metamodels for uniformity). The EReference is then inspected; if it is not null and it has an eOpposite reference, the pattern continues to check if the type of the eOpposite of the reference is the type of the *rhs* of the condition (in this case, 'author').

In lines 39-47, a helper method is defined to help look for an *EReference* given an *EClass* and a name; its implementation is straightforward.

```
1  pattern InverseNavigation
2    folcall : FOLMethodCallExpression
3    guard: (folcall.method.name = 'select' or folcall.
         method.name = 'selectOne')
4    and folcall.conditions.isTypeOf(
         EqualsOperatorExpression)
5    and folcall.target.isTypeOf(PropertyCallExpression)
6    and folcall.target.property.name = 'all',
7
8    condition : EqualsOperatorExpression
9    from: folcall.condition
10   guard: condition.lhs.isTypeOf(
         PropertyCallExpression)
11
12   lhs : PropertyCallExpression
13   from: condition.lhs
14   guard: lhs.resolvedType.isTypeOf(ModelElementType),
15
16   rhs : NameExpression
17   from: condition.rhs
18   guard: rhs.resolvedType.isTypeOf(ModelElementType),
19
20   lhsType : ModelElementType
21   from: lhs.resolvedType,
22   rhsType : ModelElementType
23   from: rhs.resolvedType
24   guard: lhsType.ecoreType = rhsType.ecoreType
25   {
26     match {
27       var r = getReference(lhs.target.resolvedType.
            ecoreType, lhs.property.name);
28       if(r.upperBound = 1 and r.eOpposite <> null and
             r <> null)
29       {
30         if(r.eOpposite.eType = lhs.target.resolvedType
               .ecoreType)
31         {
32           return true;
33         }
34       }
35       return false;
36     }
37   }
38
39 operation getReference(class: Any, name:String)
40 {
41   for(r in class.eReferences)
42   {
43     if(r.name = name)
44       return r;
45   }
46   return null;
47 }
```

**Listing 5: EPL pattern for inverse navigation**

### 4.6.3 The Java pattern

Our original attempt to construct the sub-optimal performance detection facility was to define patterns using Java, we defined a method in Java to achieve the same function described above. The equivalent Java implementation is 76 lines of code with a long chain of If statements which makes it very difficult to comprehend. With the EPL approach, the patterns are more comprehensible. Developers can contribute to the pattern pool by defining their own EPL patterns and registering them with the framework through appropriate extension points.

## 5. TOOL SUPPORT

The static analyser proposed in this paper is a pragmatic static analyser. The flexibility of EOL allows its users to write code with optional typings that always work at runtime. Reporting errors on such cases are not desirable for EOL, especially on legacy EOL code that is proven to work with extensive testing. Thus the design decision was to allow such behaviour and delegate the resolution to the EOL execution engine. We applied the static analyser on a large EOL program (Ecore2GMF.eol) that underpins the Eugenia tool [10] for evaluation. We allow optimistic typing - when Any type is encountered in assignments, operation calls and property calls, we provide a warning message to

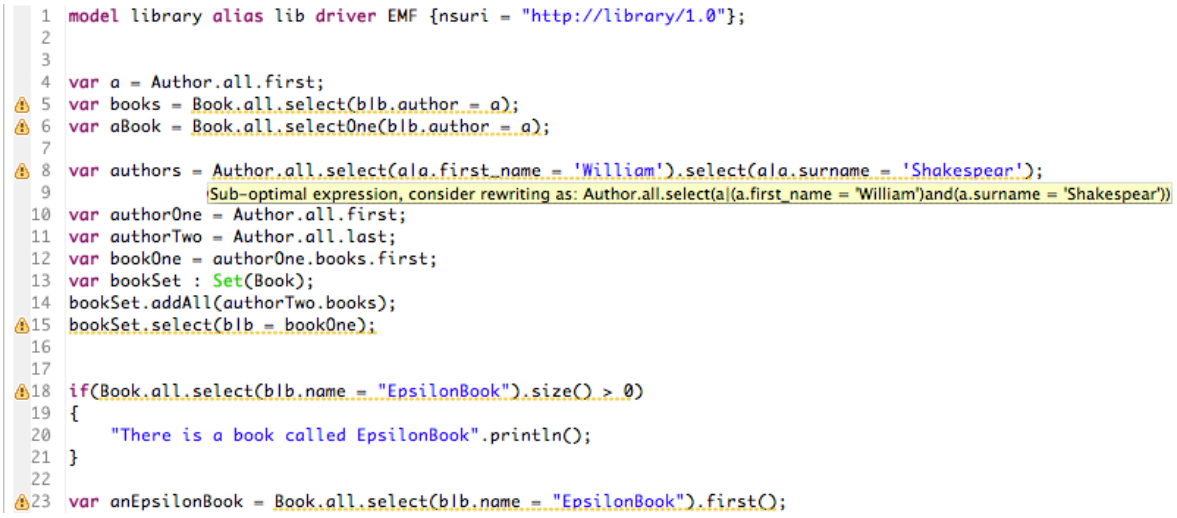



**Figure 8: EOL Editor Screen Shot**

notify the user that there might be potential type incompatibility issues at run-time. With this configuration, analysis on Ecore2GMF.eol which consists of 1212 lines of code generates 126 warning messages. This result shows that the static analyser supports plausible legacy code. At the same time, it provides reasonable warning messages when optional typing is used.

After evaluating the static analyser, we evaluate the sub-optimal performance detection facility. Figure 8 provides a screenshot of the editor we implemented by extending the existing EOL Eclipse-based editor. The lines of code with warnings represent matches of the patterns discussed above. The implementation of the editor is able to extract and display the warning messages generated by the detection facility. The sub-optimal performance detection facility is not only able to detect patterns that incur performance overheads, but also provide suggestions on how to rewrite the code. An example warning message is shown in line 8, the warning message suggests to rewrite the operation as:

```
1 Author.all.select(a|a.first_name = "William" and a.
      surname = "Shakespeare")
```

The rest of the patterns function as expected.

## 6. RELATED WORK

There are several automated analysis and validation tools for model management programs. In [14] the authors propose a generic static analysis framework for model transformations specified in VIATRA2 Textual Command Language (VTCL [2]). The latest static analysis framework detects common errors and type related errors regarding models. However, the VIATRA2 framework provides limited support for other metamodelling technologies as it uses its own modelling language (VTML) and store the metamodels in the model space. Additionally, VTCL is not as flexible as EOL; it does not provide optional typing mechanisms as EOL does.

Acceleo [4] provides static analysis mechanisms for syntax highlighting, content assistant, and model related error detection. However, to the best of our knowledge, it does not support modelling technologies other than EMF.

Xtend [3] also provide static analysis facilities which are used to detect syntax and built-in type-related errors, model related type information validations are not included.

In [15], a static analysis tool is proposed to detect errors in model transformations written in the Atlas Transformation Language (ATL), the tool presented is used to convert an ATL program into a model, but no validation algorithms are implemented on this tool to our best knowledge.

The latest release of ATL IDE [5] provides a static analysis facility, it resolves the types of variables including built-in ATL types and types related to metamodels. The ATL IDE also provides code-completion of operation calls and metamodel element navigations. However, the static analysis is not responsible to provide any errors on type incompatibilities as it adopts an optimistic and flexible approach. The ATL platform also provides limited support for multiple modelling technologies other than EMF.

In [11], ways of deriving optimisation patterns from benchmarking OCL operations for model querying and navigation are proposed and several optimisation patterns are identified, including short-circuit boolean expressions, opposite relationship navigation, operations on collections, etc.

## 7. CONCLUSIONS AND FUTURE WORK

In this paper we have reported on our work on designing and developing a static analysis framework for the core language of Epsilon (EOL). The focus of our report is mostly on the sub-optimal detection facility of the static analysis framework. However, it is to be noted that the static analysis framework is able to detect various type-related errors that may occur using EOL. The static analysis framework follows a pragmatic approach so as not to compromise the flexibility

of the Epsilon languages. As a result, it can generate false negatives (problems that exist but cannot be detected by the static analyser). To minimise the number of false negatives, a more strict coding style is encouraged - to avoid the use of *Any* type as much as possible, so that the static analyser can perform more accurate analysis. This is clearly a trade-off to make; to obtain better error reporting, developers need to write more boilerplate code with explicit type casting, while to obtain better flexibility, developers need to bare with the fact that the analyser may produce false negatives that emerge at run-time.

It should be noted that the sub-optimal performance detection facility is only one application of the static analysis framework for Epsilon. In the future, we plan to look into facilities such as program comprehension, metamodel coverage analysis, impact analysis, etc. We will also look into the possibility of pre-loading models and look for more fine-grained performance patterns for EOL programs.

# 8. REFERENCES


[1] Epsilon labs. `https://epsilonlabs.googlecode.com/svn/trunk/StaticAnalysis/`.

[2] A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287. ACM, 2006.

[3] P. Friese and B. Kolb. Validating Ecore models using oAW Workflow and OCL. *Eclipse Summit Europe*, 2007.

[4] J.-M. Gauthier, F. Bouquet, A. Hammad, F. Peureux, et al. Verification and Validation of Meta-Model Based Transformation from SysML to VHDL-AMS. In *MODELSWARD 2013, 1st Int. Conf. on Model-Driven Engineering and Software Development*, pages 123–128, 2013.

[5] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[6] A. Koenig and B. Moo. Templates and duck typing. *Dr. Dobbs, June*, 2005.

[7] D. Kolovos. *An extensible platform for specification of integrated languages for model management.* PhD thesis, University of York, 2008.

[8] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez. The Epsilon Book. *Structure*, 178, 2010.

[9] D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer, 2006.

[10] L. M. Rose, D. S. Kolovos, and R. F. Paige. Eugenia live: a flexible graphical modelling tool. In *Proceedings of the 2012 Extreme Modeling Workshop*, pages 15–20. ACM, 2012.

[11] J. Sánchez Cuadrado, F. Jouault, J. García-Molina, and J. Bézivin. Deriving ocl optimization patterns from benchmarks. *Electronic Communications of the EASST*, 15, 2008.

[12] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[13] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[14] Z. Ujhelyi, A. Horváth, and D. Varró. A generic static analysis framework for model transformation programs. Technical report, Technical report, Budapest University of Technology and Economics, 2009.

[15] A. Vieira and F. Ramalho. A static analyzer for model transformations. In *3rd International Workshop on Model Transformation with ATL, Zurich, Switzerland*, 2011.