Approach to Define Highly Scalable Metamodels Based on JSON

Markus Gerhart, Julian Bayer, Jan Moritz Höfner, and Marko Boger

University of Applied Sciences Konstanz, ProGraMof,
Brauneggerstraße 55, 78462 Konstanz, Germany
{mgerhart,jubayer,jahoefne,marko.boger}@htwg-konstanz.de
http://www.htwg-konstanz.de

Abstract. Domain-specific modelling is increasingly adopted in the software development industry. While open source metamodels like Ecore have a wide impact, they still have some problems. The independent storage of nodes (classes) and edges (references) is currently only possible with complex, specific solutions. Furthermore the developed models are stored in the extensible markup language (XML) data format, which leads to problems with large models in terms of scaling.
In this paper we describe an approach that solves the problem of independent classes and references in metamodels and we store the models in the JavaScript Object Notation (JSON) data format to support high scalability. First results of our tests show that the developed approach works and classes and references can be defined independently. In addition, our approach reduces the amount of characters per model by a factor of approximately two compared to Ecore. The entire project is made available as open source under the name MoDiGen. This paper focuses on the description of the metamodel definition in terms of scaling.

Keywords: Metamodel definition, JavaScript Object Notation, JSON, Model scalability, Metamodel scalability, Model storage

1 Introduction

Tools for creating **D**omain-**S**pecific **M**odeling **L**anguages (DSML) are becoming more accepted in the software development industry to develop specific solutions for specific problems. These solutions are developed with tools such as Xtext[5], **M**eta **P**rogramming **S**ystem (MPS)[6], MetaEdit+ Modeler[7], Kybele[8], MagicDraw[9] or Eugenia[10]. The underlying metamodel of the tools is of crucial importance, because it serves as basis for all subsequent steps like code generation or programatic manipulation of the model data. Therefore, access to the metamodel has to be very simple and the memory consumption should be as low as possible.

Only if the metamodel is maintained at a high abstraction level, the subsequent programmatic processing can be implemented simple, clean, and clear like it is suggested by the KISS-principle ("Keep it simple, stupid"). However, existing open source tools cover only the needs of specific subject areas such as the software development industry. Requirements of very complex metamodels for instance the statics of buildings can currently only be fulfilled through complex detours. Not considering commercial solutions, because they do not give insight into the structure of their metamodel and storage solution, another common problem is the storage consumption of very large models. Furthermore, a great weakness of existing open source solutions such as Ecore[1] is, that they do not scale well to very large models.

Our suggested approach allows the definition of metamodels in a simple and clear way. We offer the possibility, that nodes (classes) and edges (references) can exist independently and with equal rights. This leads to a variety of possibilities in the creation of metamodels. In addition, the data of metamodels and models is held in the **J**ava**S**cript **O**bject **N**otation (JSON)[17] file format. This allows the smooth scaling of the model data using existing solutions such as CouchDB[14], MongoDB[15], and RavenDB[16].

We demonstrate, that the storage of metamodels and models is possible without problems, even when involving well over 10,000 elements (classes and references).

The paper first reviews related work in the field in section 2, which is mostly other tools and techniques for the definition of meta models. Our general approach for the definition of meta-models based on JSON for high scalability is described in section 3. The core contribution of this publication is the developed metamodel with independent classes and connections and the data structure using JSON to store models for high scalability. Section 4 illustrates the results of our approach from different angles. Finally, we summarise the limitations of our research and draw conclusions in section 5.

2 Related Work

The Ecore metamodel of the **E**clipse **M**odeling **F**ramework (EMF) stores edges as parts of nodes. An EReference actually models one end of an edge [1]. This is also true for the **G**eneric **M**odeling **E**nvironment (GME) [2] and WebGME [3]. The disadvantages of this approach were already discussed in [11]. Storing edges as parts of nodes does not scale well for large numbers of edges. Accessing, loading and saving individual edges requires linear time and may pose a problem in terms of heap memory. As an alternative, [11] proposes storing edges as relations like a relational database, separating the edge from the node. That is the same basic principle we follow. Instead of an SQL-like structure as proposed in [11], we introduce the edge as a first-level-object that is stored in a NoSQL database.

The Ecore metamodel uses an E**x**tensible **M**arkup **L**anguage (XML) based format for model serialization [1]. The need for a model serialization format that is not based on XML was formulated in [12]. While our approach does not satisfy all the criteria for model storage set forth in that paper, we believe that JSON as the basic format for model storage will simplify implementing these criteria. A diagram is stored as a collection of JSON objects which can be addressed through an identifier. Instead of loading large models all at once, objects could be loaded as needed using their identifier. The partial loading of model data based on JSON is covered by databases like CouchDB or MongoDB. An XML-based format would always have to be loaded in full apart from approaches such as partial parsing which significantly increases the runtime.

Approaches such as GEMSjax [4] or EMF-REST [19] provide **Re**presentational **S**tate **T**ransfer (REST) access to Ecore models and translate them into either JSON or XML. Compared to a method that stores JSON data in a document oriented database, these approaches require additional overhead, as the Ecore model has to be serialised into the target format on each call to the REST interface.

3 Approach

In this section we will give a detailed explanation of our approach. First we discuss and reason about the architecture of our metamodel and its components and then present an example.

3.1 Architecture

An overview of the MoDiGen metamodel architecture is given in Figure 1. The design is focused on universality and simplicity. We utilise standard conform JSON to store both model and instance data. This helps us to achieve programming language agnosticism and scalability. In the following paragraphs we will discuss the architecture depicted in Figure 1 and the corresponding metamodel components in detail.

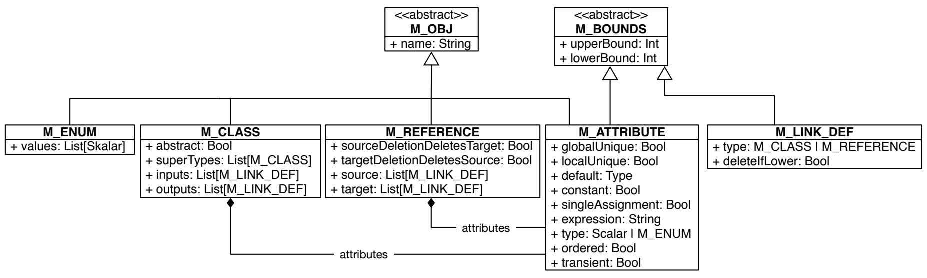

Fig. 1. MoDiGen Metamodel

M_Obj is the abstract base class of most of the metamodel's components. Its use is to provide a name attribute to all components that need to be identifiable. The name attribute is guaranteed to be unique on a model scope.

M_Bounds is abstract and defines *upperBound* and *lowerBound* attributes. Bound values can either be zero, a positive Integer, or -1 for infinity. By defining an upper as well as a lower bound, one can model maxima, minima, as well as ranges. By default lowerBounds are set to 0 and upperBounds to -1.

The modelling of attributes in nodes and edges is done using **M_Attribute**. It extends M_Bounds so it can be defined to either be single-valued or an array with an optional maximum and/or minimum length. To further define its behaviour a number of mandatory attributes exist which will be discussed now. If *uniqueLocal* is set to true the M_Attribute behaves much like a Set data structure, for it is a collection which can not contain any duplicate values. The *uniqueGlobal* flag in contrast, guarantees that the attribute's values are unique on a model scope. This may be useful for modelling attributes like Social Security Numbers. The *default* attribute is a value of *type* and defines the initial value of the M_Attribute. Using *expression* one can define a simple arithmetic formula to derive the value of the attribute. This renders the attribute read only and can be useful in cases where one attribute depends on other attributes. Whether the M_Attribute is a String, Integer, Double, or Enum is defined by setting the *type* attribute. The *ordered* flag defines whether the attribute's values are ordered in some fashion. The attribute *transient* determines whether the attribute is transient. If set to true the attribute's value won't be stored when the model is being saved to a database. This might be useful for attributes which are the result of an expression. Single Assignment behaviour can be modelled by setting the *singleAssignment* flag, this causes the value to be settable exactly once. *Constant* on the other hand means that the value is *default* and may never be changed.

Nodes are modelled using **M_Classes**, which in addition to a number of mandatory attributes may contain an arbitrary number of user defined attributes. The mandatory attributes are defined in the following manner. *Abstract* denotes whether the M_Class is declared as abstract, meaning it may not be instantiated but only be used as a base for other M_Classes. Inheritance between M_Classes is modelled using the *superTypes* attribute. It contains all direct predecessors. By defining *superTypes* as a list we explicitly allow multiple inheritance. The *inputs* attribute is a list of all incoming M_Link_Defs and *outputs* is a list of all outgoing M_Link_Defs. On the other side M_Reference also has a *target* and *source* attribute which behaves like *inputs* and *outputs*.

M_Link_Def is used to define one endpoint of a connection and has a *type* attribute which is either M_Class or M_Reference as well as an upper- and lower-bound. The flag *deleteIfLower* defines whether the M_Class and M_Reference which contains the M_Link_Def should be deleted in case the number of values of the M_Link_Def drops to its lower bound. This means one can model a minimum count of Output/Input or Source/Target values a certain Class or Reference must have of any type.

An **M_Reference** denotes an edge between two M_Classes. By defining references as first-level classes our metamodel gains a couple of powerful modelling possibilities. For example edges may have an arbitrary number of custom attributes and allow n:m relationships. A set of standard attributes also exists which will be defined now. With *sourceDeletionDeletesTarget* set, in case the source of the M_Reference is deleted, the target, and the reference itself is also deleted. Accordingly *targetDeletionDeletesSource* deletes the source and the reference in case the target is deleted. This can be useful to model containments and similar constructs where one end of a reference can not exist without the other end. The *source* attribute is a list of sources and the *target* attribute is a list of targets of the M_Reference. Both have the Type M_Link_Def, therefore M_References can be defined to be valid for a number of different source- and target-classes with dedicated bounds for each class in both directions. For example one can define an edge to be valid from A to B or C and further specify separate bounds for the number of edges from A to B or A to C as well as separate bounds for the number of Bs, Cs, and As involved in those edges.

Finally **M_Enum** is a simple Enum of a scalar type and might be used as a type for attributes.

3.2 Example

To demonstrate the capabilities of our metamodel, we will consider the following (oversimplified) family tree model. Three M_Classes, *Person*, *Male*, and *Female* exist, where Male and Female inherit from Person and Person is abstract. The following relationships exist between these classes: The Relationship *isHusband* has a Male source and a Female target, while the reverse relationship *isWife* has a Female source and a Male target. The Male class also has a relationship *isFather*, directed at Person and the Female class has the corresponding *isMother* relationship, also directed at Person. Figure 2 illustrates the setup.

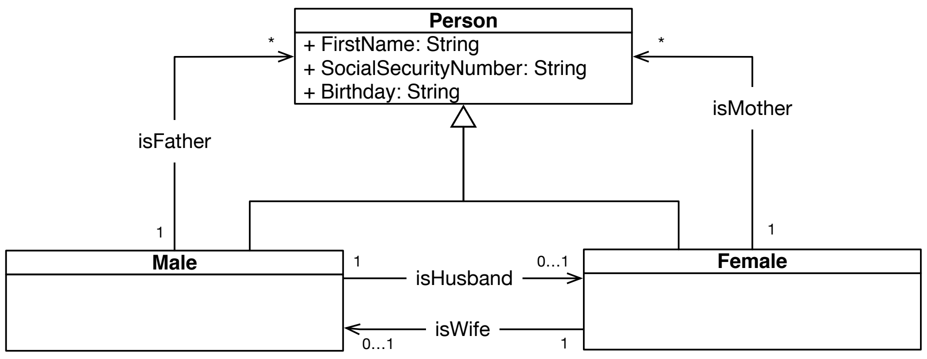

Fig. 2. Family Tree Model

Listing 1.1 is taken from the compressed familytree model and represents the M_Class Male. The *mtype* property is necessary because JSON has no type system and so this property is needed to declare that this is an M_Class. The *inputs* and *outputs* properties are M_Link_Defs linking to the respective M_Reference and indicating that at most one such relationship is allowed for one instance of this M_Class. Because Male inherits all its attributes from Person *mAttributes* is empty.

```
"Male": {
  "mType": "mClass",
  "name": "Male",
  "superTypes": ["Person"],
  "mAttributes": [],
  "inputs": [
    { "type":"isWife",
      "upperBound": 1,
      "lowerBound": 0
    }
  ],
  "outputs": [
    { "type":"isHusband",
      "upperBound": 1,
      "lowerBound": 0
    },
    { "type":"isFather",
      "upperBound": -1,
      "lowerBound": 0
    }
  ]
},
```

Listing 1.1. The Male M_Class taken from the Family Tree example model

An M_Reference is given in listing 1.2. This is the *isHusband* M_Reference linking the Male source to the Female target. The *isHusband* reference links exactly one Male object to one Female object.

```
"isHusband": {
  "mType": "mRef",
  "name": "isHusband",
  "mAttributes": [],
  "source": [
    { "type":"Male",
      "upperBound": 1,
      "lowerBound": 1
    }
  ],
  "target": [
    { "type":"Female",
      "upperBound": 1,
      "lowerBound": 1,
    }
  ]
}
```

Listing 1.2. The isHusband M_Reference taken from the Family Tree example model

We instantiate this model with three persons. A Male instance and a Female instance, who are married to each other (using the isHusband and isWife references) and another Male, who is the child of the other two.

Listing 1.3 shows part of the JSON of an instance of the family tree model. Specifically it shows one instance of the Male class and one of the isHusband Reference. The complete JSON source for both the model and the instance can be found at MoDiGen[13].

```
"846bc8a2-00fc-401f-b626-0b0252516aee": {
  "mClass": "Male",
  "outputs": {
    "isFather": ["8e9b1093-a589-4ae4-8e1e-1b3d63a3f842"],
    "isHusband": ["ee204744-6322-49d4-928e-1442e8bc70c4"]
  },
  "inputs": {
    "isWife": ["666d4de7-e0f2-4620-8c19-d5469b40be1f"]
  },
  "mAttributes": {
    "First_Name": ["Hans"],
    "SocialSecurityNumber": ["12"],
    "Birthday": ["12-02-2015"]
  }
},

"ee204744-6322-49d4-928e-1442e8bc70c4": {
  "mRef": "isHusband",
  "source": {
    "Male": ["846bc8a2-00fc-401f-b626-0b0252516aee"]
  },
  "target": {
    "Female": ["a264a43b-6f97-4257-9243-baddbf745490"]
  }
}
```

Listing 1.3. Family tree instance

4 Evaluation

The presented approach makes it possible to create nodes and edges with equal rights. This results from the revised metamodel definition which is crucial for programmatic processing of the model data. The storage of metamodel and model information is done using JSON. This allows for easy integration and processing by conventional programming languages and web technologies. Furthermore, the number of characters and therefore the storage consumption for metamodel definitions and model instances was reduced, compared to the XML data structure of Ecore, by separating edges and nodes, and by using JSON.

The change of the number of characters based on the metamodel definition of the familytree example is shown in Figure 3. It turns out that the number

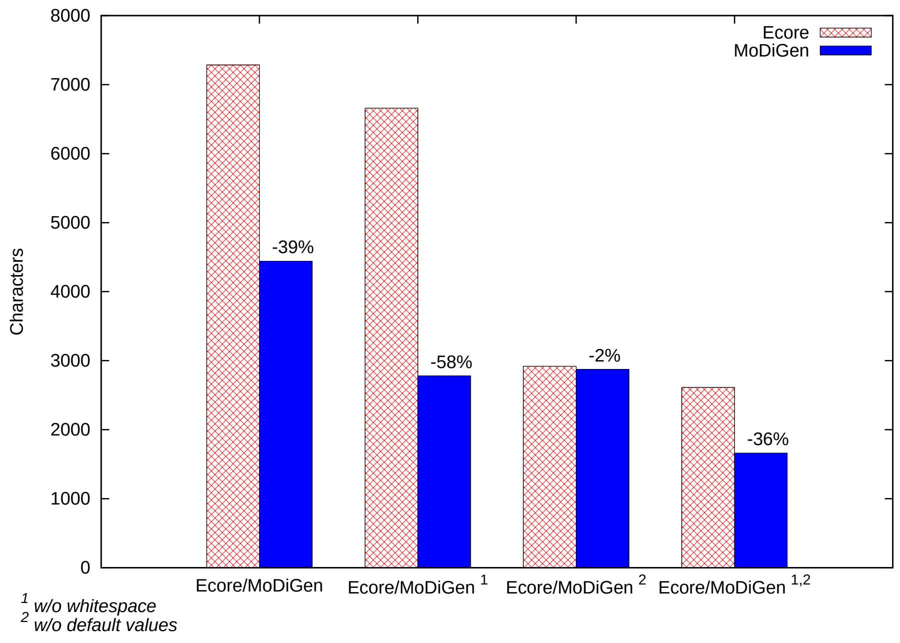

Fig. 3. Number of characters for the Familytree Metamodel

of characters were consistently reduced when compared to Ecore. The biggest difference can be revealed by removing white spaces. This comes at the expense of human readability but is irrelevant for machine processing. By removing default values a further reduction was achieved. Ecore applies these measures by default.

The development of the number of characters based on the model instance, depending on the number of nodes is shown in Figure 4. This is based on the smallest possible model instances (without whitespace and default values) for a model where all nodes are interconnected. It can be seen, that for smaller models the Ecore approach is more appropriate, but for larger models the presented approach has advantages. This is mainly caused by the changed handling of connections between objects. If only few connections are present in a model the advantage of our approach relativizes. We generated model instances with 10,000 interconnected nodes for Ecore as well as MoDiGen and found that the storage consumption of Ecore was 5,58 Gigabyte and the storage consumption of MoDiGen was 3,6 Megabytes.

Our approach has advantages regarding the scalability of big models. This is mainly due to the used data structure implemented in JSON. Data formats like JSON can easily be horizontally scaled using existing database solutions like CouchDB, MongoDB or RavenDB. This is additionally favoured by the lower memory consumption of the developed metamodel. In contrast to XML-based

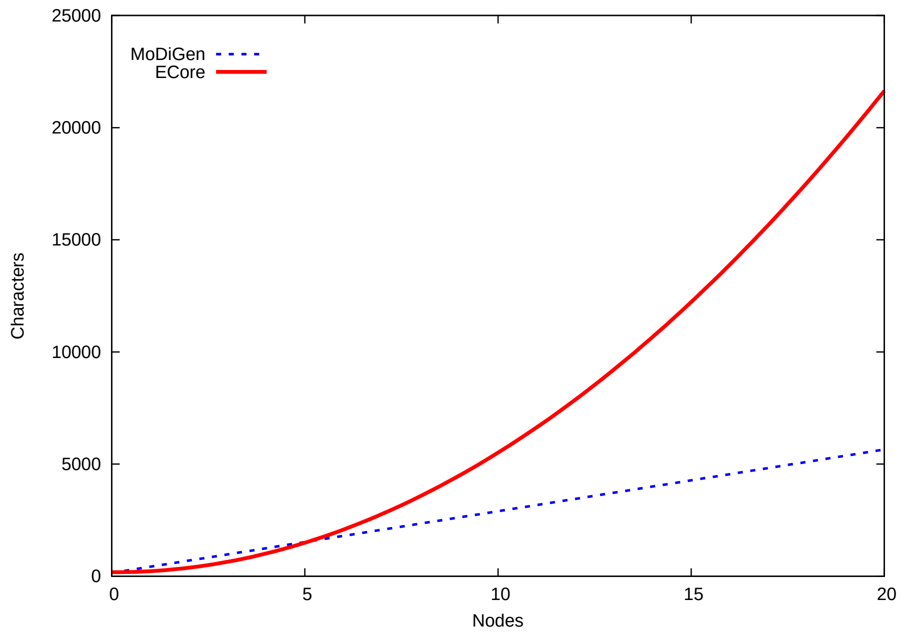

Fig. 4. Development of the number of characters for the familytree model instance according to the created nodes

data structures the JSON based data structure offers the possibility to access just parts of the stored model.

5 Conclusion and future work

We have introduced the MoDiGen metamodel and shown how its approach differs from other metamodels for DSMLs, such as Ecore or GME. Treating edges as first level objects instead of features of nodes allows for easy programmatic access to the edges. The use of JSON yields more compact models than XML, allows for seamless integration into web applications using JavaScript, and opens the door for improvements regarding scalability.

In comparison to Ecore, the MoDiGen metamodel lacks the possibility to define operations. While in the Ecore itself, EOperation is more of a placeholder, it can be given an implementation in the context of the Eclipse Modeling Framework. Edges as first level objects give easier access to references and permit the existence of stand-alone edges. However, this also means that the modeller has to explicitly state whether an edge must be automatically deleted upon the deletion of one of the connected nodes. This is not a problem in Ecore where references are deleted when the containing class is deleted. In its current form, the JSON representation of MoDiGen models still contains code that could be removed.

For example, attributes that have their default value or empty properties are still in the JSON. The JSON representation could be significantly compressed by removing that code.

In the future, we plan to implement a complete modeling framework on the basis of this metamodel and work on improving the JSON representation in terms of size. We will also use the MoDiGen metamodel for code generation projects. Furthermore, we plan on extending the metamodel to allow specification of constraints using the **O**bject **C**onstraint **L**anguage (OCL)[18].

References

1. Steinberg, D., Budinsky, F., Paternostro M., Merks E.: EMF Eclipse Modeling Framework, Second Edition. Addison-Wesley Professional (2008)
2. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. In: Proceedings of WISP'2001, IEEE, Budapest (2001)
3. Maróti, M. Kereskényi, R, Kecskés, T., Völgyesi, P., Lédecyi, Á.: Online Collaborative Environment for Designing Complex Computational Systems. Procedia Computer Science, Volume 29, pp. 2432-2441 (2014)
4. Farwick, M., Agreiter, B., White, J., Forster, S., Lanzanasto, N., Breu R.: A Web-Based Collaborative Metamodeling Environment with Secure Remote Model Access. ICWE 2010, LNCS 6189, pp. 278-291. Springer-Verlag, Heidelberg (2010)
5. Itemis AG, `http://www.eclipse.org/Xtext`
6. MPS Meta Programming System, `https://www.jetbrains.com/mps/`, Accessed 2015-01-22
7. MetaEdit+ Modeler, `http://www.metacase.com/mep/`, Accessed 2015-01-22
8. Kybele GMF Generator a tool for developing GMF editors in a few steps, http://www.kybele.etsii.urjc.es/kyb_kybelegmfgen/, Accessed 2015-01-22.
9. No Magic Inc.: MagicDraw, https://www.magicdraw.com/, Accessed 2015-01-22.
10. Eugenia a tool to automatically generate a GMF editor, http://eclipse.org/epsilon/doc/eugenia/, Accessed 2015-01-22
11. Scheidgen, M.: Reference Representation Techniques for Large Models. In: BigMDE'13, ACM, Budapest (2013)
12. Kolovos, D., Rose, L., Matragkas, N., Paige, R., Guerra E., Cuadrado, J., De Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In: BigMDE'13, ACM, Budapest (2013)
13. MoDiGen, `http://www.modigen.de/publications/`
14. CouchDB, `http://couchdb.apache.org/`
15. MongoDB, `https://www.mongodb.org/`
16. RavenDB, `http://ravendb.net/`
17. Java Script Object Notation, `https://tools.ietf.org/html/rfc7159`
18. OMG Object Management Group: Object Constraint Language Version 2.4, `http://www.omg.org/spec/OCL/2.4/` (2014)
19. EMF-REST, `http://www.emf-rest.com`