JSMF: a flexible JavaScript Modelling Framework

Jean-Sébastien Sottet and Nicolas Biri

Luxembourg Institute of Science and Technology,
5 avenue des Hauts Fourneaux, Esch/Alzette, Luxembourg
{jean-sebastien.sottet,nicolas.biri}@list.lu

Abstract. Model-Driven Engineering (MDE) technologies are more and more used outside the software engineering field such as the typical cases of code generation and formal validation and verification. In fact, MDE is applied to many different modelling situations such as regulation, law compliance, data analytics, etc. In these domains, we often face incomplete and/or evolving requirements, which implies the need for specific modelling facilities. In particular, we must be able to relax standard metamodel definition in order to express uncertainty and unforeseen modelling constructs.
However metamodels are important for dealing with computer-based manipulation of models: from the flexible models a (new) metamodel should emerge. It will freeze the modelling language features that permit: model transformation, comparison, evolution, etc.
In this article we propose a framework called JSMF, based on JavaScript. It allows for defining a flexible metamodel, to freely define models and provide controls for checking the level of conformance. We also show how to use JSMF for the incremental definition of a frozen metamodel, starting from a flexible one.

Keywords: Flexible modelling, Conformance relaxation, Natural modelling

1 Introduction

Model-Driven Engineering (MDE) is increasingly applied to modelling cases where the traditional approach of building models that conform to a precise and pre-defined metamodel is not sufficient (e.g., enterprise modelling). Indeed, models are built by domain experts that are not using traditional modelling tools. Surveys regarding enterprise models [8] show that domains experts are more eager to use home-grown and semi-structured languages than using a traditional modelling language, defined by a strict metamodel. These modelling situations (e.g., an enterprise architect trying to build organizational view of the enterprise) often require a more flexible modelling tool support [9]. Many reasons could explain this need for freedom: exploration of a new domain, unclear border of model purpose, imprecise/incomplete knowledge about the system under study, etc.

If a flexible modelling environment (i.e., not enforcing strict conformance) allows for better handling of models for specific situations, it lacks the MDE key capabilities, e.g., model transformation, computation, comparison, etc. In order to allow computer based manipulations of such models, the flexible (meta)model should be progressively frozen [5] ensuring the modelling continuum [12]. Once frozen, the metamodel becomes a rigid metamodel as opposed to a flexible metamodel.

Many approaches exist for bridging sketching tools/free modelling and the MDE approach trying to reconcile inconsistencies between flexible models and their underlying metamodel. In this article we will not investigate the alignment of drawing tools elements and metamodel concepts nor try to infer types for model elements. We rather propose a framework that allows for 1/ defining level of flexibility regarding the conformance relation, 2/ consequently being able to define models that not strictly conform to a metamodel (e.g.: by adding properties), 3/ making new metamodel emerging for raw elements.

We will first introduce the JSMF framework as a general overview focusing on its flexibility feature. In a second time, we depict our vision of model flexibility based on (meta)models features. Finally, we propose a reconciliation mechanism for the conformance relation.

2 JavaScript Modelling Framework - JSMF

The JavaScript Modelling Framework (JSMF) has been designed for providing a flexible modelling environment that could support the requirements of natural modelling [12]. It is a JavaScript-embedded Domain Specific Language (DSL) inspired by the Eclipse Modelling Framework (EMF) in its basic functions but that relies on JavaScript dynamic typing and on a relative independence between a metamodel and a model. JSMF also comes with a set of tools (not detailed in this article) to manipulate and compute properties about models: JSTL for model-to-model transformation, model checking and querying facilities.

Notion of model and metamodel. In JSMF, the notion of model is seen as a container: it contains model elements which can be shared amongst models. A model can contain metamodel or model elements independently. An explicit reference to a (meta)model (i.e., reference model) can be added to a model but is not mandatory.

Model elements. JSMF actually differs from existing EMF translation/adaptation in JavaScript like, for instance, EMFJSON [1] or a lightweight EMF implementation in JavaScript JSMF [2]: it does not copy the instantiation mechanism of EMF/Java using the JavaScript prototype. On the contrary a model element conforms to a metamodel element only using instantiation mechanism. But it does not prevent from adding any properties (i.e., like any *standard* JavaScript

[1] see: https://github.com/emfjson/emfjson-jackson
[2] see: https://github.com/dslmeinte/jsmf

object). Rules for setting (and getting) attributes and references are dynamically created at object creation but can be adapted afterwards. Moreover the metamodel and model can evolve independently, the model is just keeping a reference to the metamodel that has been used for its creation (which can be inconsistent).

In JSMF, a metamodel can be defined using different syntaxes as shown in Listing 1. The classical syntax is inspired by the EMF API. In the example, a `Family` is a new instance of class `Class`. This `Family` class has an attribute `lastname` of type `String` - here we use the basic type of JavaScript. One can also use `jsmf.String` (as shown below as a type element). In JSMF, types are simply functions that returns true if the given value in a model (i.e., during instantiation) is valid for this type. When we provide `String`, it's implicitly translated in the `jsmf.String` function that returns true for any string. The `Family` class has also a relation named `members` that related it to a class `Person` with a cardinality (0..*) as defined by `JSMF.Cardinality.any`.

The compact syntax is largely inspired by the one used in JavaScript for defining raw objects. Here, we present a class `Person` that has no super-type (empty array of types `[]`) and has two attributes `firstname` and `age` of respective type `String` and `jsmf.Positive`.

```
//Compact (JavaScript Object) Notation
const Person = Class.newInstance('Person', [],
  { firstname: String , age: jsmf.Positive})

//Classical Syntax - EMF like -
const Family = Class.newInstance('Family')
Family.addAttribute('lastname', String)
Family.addReference('members', Person, jsmf.Cardinality.Some)
```

Listing 1. Definition of Person and Family in JSMF using two possible syntaxes

As a basic feature of JSMF we allow to define relationships that target any class of any type, see Listing 2. This is equivalent to referencing the EMF EObject Class. This is a first form of flexibility: it means that any class can be targeted by such relation (i.e., target type of this reference is not discerned). For example, the previous `Family` declaration would be the following if we want to accept any class instance as a member of a family:

```
const Family = Class.newInstance('Family')
Family.addAttribute('lastname', String)
Family.addReference('members', jsmf.JSMFAny,
  jsmf.Cardinality.Some)
```

Listing 2. Definition of annotation

As a summary, the features and the conceptual structure of the JSMF (meta)modelling environment is defined in the Figure 1. In JSMF a class can inherit from multiple classes, combining all the properties together. However, if a model element is overridden into multiple inherited classes (e.g., the same attributes defined many times), only the last class is taken as the only definition [3].

[3] alike first versions (e.g., 2.3) of Python programming language.

Alike many object inspired modelling frameworks, a JSMF Class has many attributes and references. Attributes are classically defined (name, type and mandatory attributes). References are partially classically defined: reference point at a *TargetElement*. Target elements can be a precise class or any JSMF element: *JSMFAny*.

One specificity of JSMF is that we can define an *associated* class which could quality the reference/relation, i.e., providing additional information such as weight, probability, etc. Through references one can directly define a relationship: proposing an opposite reference and an opposite cardinality. Cardinalities are defined as follow by predefined set of min/max: `[0,1]`, `[1,1]`, `[0,*]`, `[1..*]`.

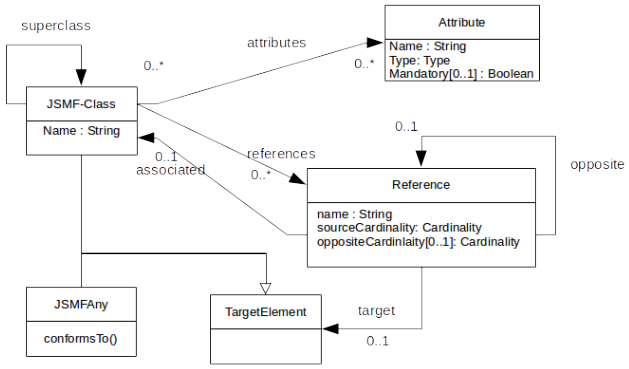

Fig. 1. JSMF Conceptual Model

For building modelling elements that conform to metamodel elements two notations also exists: a classical object (EMF-like) construction (create object first, then set attributes) or the compact JavaScript one.

```
// Expanded notation & new
const john = new Person()
john.firstname = 'John'
john.age = 46

// Compact notation & newInstance
const kennedy = Family.newInstance({name: 'Kennedy',
  members: [john]})
```

Listing 3. Definition of annotation

3 Flexibility

As presented in the previous section, JSMF looks like a classical framework that takes benefits from JavaScript dynamic typing. In this section we show how

JSMF could that offers more flexibility than traditional approaches. In order to build a flexible modelling support, several rules needed to be relaxed as well. Relaxing the conformance relation allow models to be adapted to various situations. For instance, for adapting (meta)model to the domain being discovered or to overcome language limitation regarding a peculiar purpose.

3.1 Defining Flexible (Meta)Models

Within JSMF, relaxing the conformance can apply to an entire model or just to some elements. As a result we allow each (meta)model element to be configured separately. A metamodel which has some flexible modelling elements can be referred to as a proto-metamodel. Like proto-languages [12], proto-metamodel defines a "partial and flexible structure" that a model could conform to. Some elements may not be defined, used differently amongst domain engineers and more importantly the language and models can still evolve.

We have identified some important properties for setting the flexibility level.

Attributes optionals/mandatory: By default in JSMF, attributes are optional but they can be set as mandatory thanks to an option in the attribute definition.
Attributes types: The types can be checked or not, same applies to type-imposed constraints (e.g., a Number between 0 and 20).
References cardinalities: The cardinality can be checked or not. Not checking the cardinalities means setting a 0..* cardinality.
References targeted type: As shown before, targeted types can be loosely defined using JSMFAny. There is a slight difference between using JSMFAny and not checking type. For the latter the declared type can be a hint.
References opposite: The opposite relation can be checked or not (including the opposite cardinality).

The level of flexibility can change all along the life-cycle of a metamodel.

3.2 Flexibility management in JSMF

JSMF provides facilities to define the flexibility of the models class by class and even at the attributes and references level. By default, a JSMF (meta)model comes with the following flexibility rules:

- Attributes types are checked when we assign a value to an attribute slot. If the attribute is mandatory, JSMF check that the value is undefined. Assigning an invalid value to an attribute slot will raise an exception.
- Reference targeted types are checked. An error is raised when an attempt is made to assign the wrong type to a reference.
- References cardinalities are not dynamically checked.
- Additional-attributes and references can be defined on an existing object.

JSMF allows a fine management of the two first items of the list above (attribute types and mandatory checking, and reference type checking). We can either modify the definition of the flexibility of an attribute or reference at its declaration or later on for a whole class. Here, in Listing 4 we revisit the example of Listing 1 but we replace the `Family` class by a definition with a flexible reference `members` and we set the whole `Person` class as flexible.

```
// Set the whole Person class as Flexible
Person.setFlexible(true)

// And thus is a (stupid but) valid instance declaration
const john = Person.newInstance({firstname: 42, age: 'John'})

// errorCallback define the flexible behaviour of members
const Family = Class.newInstance('Family', [],
  { lastname: {type: String, mandatory: true}},
  { members: { target: Person
             , errorCallback: jsmf.onError.silent}})

const smith = Family.newInstance('Smith')
// We can assign something else than a Person to members
smith.members = [smith]
```

Listing 4. Flexibility declaration example

As written in Listing 4, the flexibility of a whole class can be set thanks to the `setFlexible` method. For a single attribute or reference, one can use `errorCallback` during the declaration to define the behaviour of JSMF when a type error is encountered. Several default methods are provided to handle these type errors. Here, we use `onError.silent`, which discards the type error and performs the assignment. More advanced methods can be defined, allowing for example to correct some errors before the assignment.

3.3 Checking conformance

Whilst model flexibility can be desirable in an exploration phase, it may not fit with all production purposes. We need tools to assess how a model differs from its metamodel and then to reconcile them. JSMF provides a checking tool that will check basic construction rules (attributes and references types, cardinalities, extra attributes or references) as well as metamodel-specific constraints (*a* la OCL). The checker returns a JavaScript object containing all the constraints violations of a model. These violations are grouped by constraint type and come with a detailed context that allows precise identification of the issue location in the model. In Figure 5, we use the flexible `Person` and `Family` to provide a first sketch of a model instance with some differences its reference model (metamodel).

```
const DomesticPet = jsmf.Class.newInstance('DomesticPet', [],
  { name: String, race: String})
```

```
Person.setFlexible(true)

const FamilyModel = new Model('FamilyModel', {},
  [Family, Person])

const kennedy = Family.newInstance({name: 'Kennedy'})
const john = new Person({firstname: 'John',
  birthdate: new Date('1917-05-29')})
const jackie = Person.newInstance({firstname: 'Jacqueline',
  nickname: 'Jackie', birthdate: new Date('1929-07-28')})
const charlie = DomesticPet.newInstance({name:'Charlie',
  race: 'Welsh Terrier'})
kennedy.members = [john, jackie, charlie]

const Families = new Model('Families', FamilyModel,
  [kennedy, jackie, john, charlie])
```

Listing 5. Flexible metamodel and model instance

If we check the `Families` model (typing `Families.check()`), we obtain as a result a JavaScript object that gather all the errors, grouped by error type. The conformance check of the example of Listing 5 catches the following error:

- The family object Kennedy has a field lastname which is mandatory but undefined.
- The Kennedy Family also has a member that is not a Person, Charlie.
- Several elements ave additional attributes: Jacqueline has a birthdate and a nickname, and John has a birthdate.

To give an insight of the object representation, Listing 6 provides an excerpt of the conformance check object. Note that this is a raw output of a JavaScript object, not intended to be used as-is for human comprehension. The excerpt shows the reference type violation entries. It is composed of a unique error. The error is defined as a heterogeneous list, where the first element is the model element that contains the error (the Kennedy family), the second element is the reference name that contains the error (members) and the third element is the wrongly typed element (the dog Charlie). The advantage of the output as a JavaScript object is that we have a direct access to the model elements that do not conform to the reference model. Thus we can access and modify them directly.

```
{ // ...
  referencesTypeRule:
   [ [ { lastname: undefined, members: [Object], name: '
       Kennedy' },
       'members',
       { name: 'Charlie', race: 'Welsh Terrier' } ] ],
  // ...}
```

Listing 6. An excerpt of `check` result

4 Metamodel and Model reconciliation

The Metamodel and Model reconciliation involves applying different modifications to the model and to the proto-metamodel to obtain, ideally, a model and a metamodel such that the first conforms to the latter. As an embedded DSL language, and thanks to its flexibility, JSMF can take advantage of JavaScript expressiveness to tweak both the model and the metamodel quite easily. A classical reconciliation session consists in applying modifications to the model and metamodels and observing the conformance errors of the resulting (meta)models.

A subtlety is that, as explained in Section 2, evolution of a metamodel element is not immediately replicated on elements that conform to it. We provide two functions, `refreshElement` and `refreshModel` to respectively resynchronize an element or a whole model with an up-to-date metamodel.

Based on the example in Listing 5 we illustrate the two major reconciliation approaches: modification of the model elements to conform to the metamodel and the converse.

Applying change directly to the model. Let's decide that `lastname` is the correct attribute name for a family name. Thus, we must change the `kennedy` object accordingly, which can be done in two JavaScript instructions: `kennedy.lastname = kennedy.name; delete kennedy.name`. Here, the metamodel is unchanged, we have just fixed a model element.

Applying change directly to the metamodel. Changes to the metamodel can be performed manually or through heuristics functions. Manually, we can for example add a `nickname` attribute to the `Person` class quite easily and then we just have to refresh the model elements, as follows:

```
Person.addAttribute('nickname', String)
jsmf.refreshModel(Families)
```

Discovering metamodel structure from a model. Changes can also come from the models used here as an *archetypal* example. Note that we are not addressing the discovery of a full metamodel from a flexible model, but rather refine some parts (i.e. class) of a flexible metamodel. One can decide that a model element is an *archetypal* representation of a class, thus updating the metamodel accordingly. The properties of the object (note that this also works for non-JSMF objects) are examined (as key-value pairs) and the attribute types inferred from the current data, notably using JavaScript's `typeof`. For references, we make the assumption that they target only JSMF elements. For instance, the Jackie Kennedy instance can help in building or updating the `Person` class by invoking the `archetypalDiscovery` function. The `Person` gets 3 attributes, here adapted from log trace:

```
attributes:
  { firstname:[Function: isString], //...
    nickname:[Function: isString], //...
    birthdate:[Function: isDate], //... }
```

5 Related work

In [4] the authors introduce the requirements for a flexible metamodelling environment. They classified the different usage and nature of metamodels to be used in either flexible or rigid modelling scenarios. Models can be defined independently from a metamodel (metamodel is then built from the model).

In the first extreme of flexibility, metamodels are not pre-defined and are inferred from models. The approaches of [2, 11] also support this type of scenario. In a same idea, [13] works on partially rigid metamodels. It involves inference only for the flexible part taking benefits from rigid parts. We did not go deep into inference mechanisms used to create metamodel [6] from example models. However, it should be implemented as an additional module in the JSMF framework. Moreover, most of these approaches [13, 11] come with graphical editor allowing users to draw any shape and then assign structure and semantic to them. This last point is beyond of the scope of the article.

The concept of Muddle [7] offers the same capabilities as JSMF. It allows for the definition of models that follow a "partial" structure defined by a metamodel that can be translated into a EMF (meta)model. In this work, we opt for a direct handling of both flexible and rigid models into our framework.

A posteriori typing [3] offers to separate the instantiation mechanism from typing/classifying. JSMF does not directly support this process but embed tools to support similar situations: relying on metamodel and model reconciliation mechanisms (see Section 4).

Finally, the approach of [10] is close to the current capabilities of JSMF. In JSMF setting a specific level of error is equivalent to *relaxing the conformance* (i.e., the model element partially conforms to its metamodel element).

6 Conclusion

We have presented JSMF[4], a flexible modelling framework which allows to define both relaxed and rigid conformance. JSMF, as we have shown in this article, comes with the possibility to check the conformance (dynamically and statically) comparing the models and metamodels and reconciling them together in both directions: i.e., by enforcing conformance (Metamodel→ Model) or by discovering structure from an example (Model→ Metamodel). With JSMF any part of any modelling element (its attributes and references) can be made flexible by disabling (i.e., silencing) the -dynamic- type checking used when assigning values. However, flexibility without control would not be useful. In order to retain benefits from MDE tools (e.g., model transformations) we need to get back to rigid modelling elements. We have designed a static checker that helps in identifying issues and supporting a progressive translation to a classical MDE approach. As such, JSMF can be seen as a prototyping environment before applying more traditional/industrial MDE solutions. The flexibility of JSMF is ideal during the exploration of a new domain, problematic, etc.

[4] see https://js-mf.github.io/ for implementation and more components.

Future work will focus on investigating inference (e.g., type inference) for ambiguous modelling elements or from any data source. Notably we aim at supporting natural modelling sessions [1] relying on natural interaction (e.g., tangible objects, shapes, etc.).

References

1. Bjeković, M., Sottet, J., Favre, J., Proper, H.A.: A framework for natural enterprise modelling. In: Proceeding of the 15th IEEE Conference on Business Informatics (2013)
2. Cho, H., Gray, J., Syriani, E.: Creating visual domain-specific modeling languages from end-user demonstration. In: Proceedings of the 4th International Workshop on Modeling in Software Engineering. pp. 22–28. IEEE Press (2012)
3. De Lara, J., Guerra, E., Cuadrado, J.S.: A-posteriori typing for model-driven engineering. In: Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on. pp. 156–165. IEEE (2015)
4. Gabrysiak, G., Giese, H., Lüders, A., Seibel, A.: How can metamodels be used flexibly. In: Proceedings of ICSE 2011 workshop on flexible modeling tools, Waikiki/Honolulu. vol. 22 (2011)
5. Hoppenbrouwers, S.: Freezing Language; Conceptualisation processes in ICT supported organisations. Ph.D. thesis, University of Nijmegen, The Netherlands (2003)
6. Javed, F., Mernik, M., Gray, J., Bryant, B.R.: Mars: A metamodel recovery system using grammar inference. Information and Software Technology pp. 948–968 (2008)
7. Kolovos, D.S., Matragkas, N.D., Rodríguez, H.H., Paige, R.F.: Programmatic muddle management. In: XM@ MoDELS. pp. 2–10 (2013)
8. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What Industry Needs from Architectural Languages: A Survey. IEEE Trans. Software Eng. (2013)
9. Ossher, H., Bellamy, R., Amid, D., Anaby-Tavor, A., Callery, M., Desmond, M., Vries, J.d., Fisher, A., Frauenhofer, T., Krasikov, S., Simmonds, I., Swart, C.: Business Insight Toolkit: Flexible Pre-requirements Modeling. In: ICSE Companion. pp. 423–424. IEEE (2009)
10. Salay, R., Chechik, M.: Supporting agility in mde through modeling language relaxation. In: XM@ MoDELS. pp. 20–27 (2013)
11. Sánchez-Cuadrado, J., De Lara, J., Guerra, E.: Bottom-up meta-modelling: An interactive approach. In: International Conference on Model Driven Engineering Languages and Systems. pp. 3–19. Springer (2012)
12. Zarwin, Z., Bjeković, M., Favre, J.M., Sottet, J.S., Proper, H.: Natural modelling. Journal of Object Technology 13(3), 1–36 (2014)
13. Zolotas, A., Matragkas, N., Devlin, S., Kolovos, D.S., Paige, R.F.: Type inference in flexible model-driven engineering. In: European Conference on Modelling Foundations and Applications. pp. 75–91. Springer (2015)