Practical Multi-level Modeling on MOF-compliant Modeling Frameworks

Kosaku Kimura, Yoshihide Nomura, Yuka Tanaka,
Hidetoshi Kurihara, and Rieko Yamamoto

Fujitsu Laboratories, Kawasaki, Japan
{kimura.kosaku,y.nomura,tanaka.yuka,kurihara.hide,r.yamamoto}
@jp.fujitsu.com

Abstract. This paper describes practices for multi-level modeling by only using existing modeling frameworks that comply Meta-Object Facility (MOF). We design modeling patterns for achieving the multi-level modeling methodologies on Eclipse Modeling Framework, and implement the dataflow model by applying the patterns. Moreover, we attempt to compare the patterns regarding the facilitation of developing both our tool and plugins. We found Orthogonal Classification Architecture (OCA) pattern is easier to develop our tool than powertypes pattern, but regarding plugins for our tool, powertypes pattern can define model-to-text transformation templates more simply than OCA pattern.

1 Introduction

Model-driven engineering (MDE) gains productivity of software developments providing several powerful tools for designing, developing or verifying software. Especially, model transformation technologies (i.e., model-to-model and model-to-text) are important for facilitating agile software developments. For the model-to-text transformation enables to generate executable source codes from a model, developers can develop complex applications by using graphical editors.

There are various kinds of graphical editing tools for developing and executing applications, e.g., Extract-Transform-Load [2, 5], Business Analytics [3] and Workflow Management [4]. We also have been developing a graphical editing tool on a cloud platform for facilitating developments of big data processing applications [18]. Figure 1 shows the web interface of our tool.

Many of the tools are based on modeling frameworks and provide automatic generation features for executable source codes. However, extending models of the tools tends to be difficult for third-party developers, and therefore, there have been a few plugins published from developer communities. Nowadays, the meta-models of graphical editing tools have to be easily extensible so that developers can develop more plugins [16].

Meta-Object Facility (MOF)[1] is a standard for MDE provided by Object Management Group (OMG), and Eclipse Modeling Framework (EMF)[2] is one

[1] http://www.omg.org/mof/
[2] https://eclipse.org/modeling/emf/

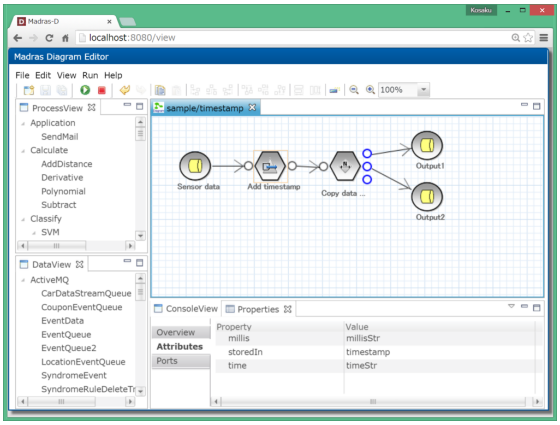


Fig. 1. EMF-based graphical editing tool for developing and executing big data processing.

of mature MOF-compliant modeling frameworks, and there are various toolkits in the EMF community, such as Acceleo[3], Query/View/Transformation (QVT) Operational[4] and ATL Transformation Language[5]. Those toolkits also conform to or follow the OMG's standards. In this paper, we attempt to achieve multi-level modeling on EMF. EMF provides the Ecore metamodel, which is compatible with Essential MOF, and tools for creating models that conform to the Ecore metamodel.

One of the major drawbacks of EMF is that it is hard to define and use a new metamodel located at the same level as the Ecore metamodel, because EMF is basically adequate to create models and objects just based on the Ecore metamodel. If we use our own metamodel, although it is obviously possible to develop a proprietary tool based on it by using the code generation feature of EMF [1, 19], the tool tends to force an unusual manner to developers, and eventually, most of them may feel that "I do not want to use it." This issue is crucial for developing the ecosystem and the community of our tool.

In order to overcome the drawbacks of existing modeling frameworks, various methodologies of multi-level modeling have been proposed such as **Orthogonal Classification Architecture (OCA)** [6, 7, 9, 11], **powertype-based metamodeling** [13, 14] and **deep instantiation** [12, 17]. The methodologies can provide simple solutions to design metamodels along with models and objects. However, there is little consensus in the literature on fundamental multi-level modeling concepts [10], and therefore, it is still difficult to determine to apply them to industries. For now, multi-level models must be defined by only using existing MOF-compliant modeling frameworks, so we have to clarify a workaround for that.

[3] `http://www.eclipse.org/acceleo/`
[4] `http://www.eclipse.org/mmt/?project=qvto`
[5] `https://eclipse.org/atl/`

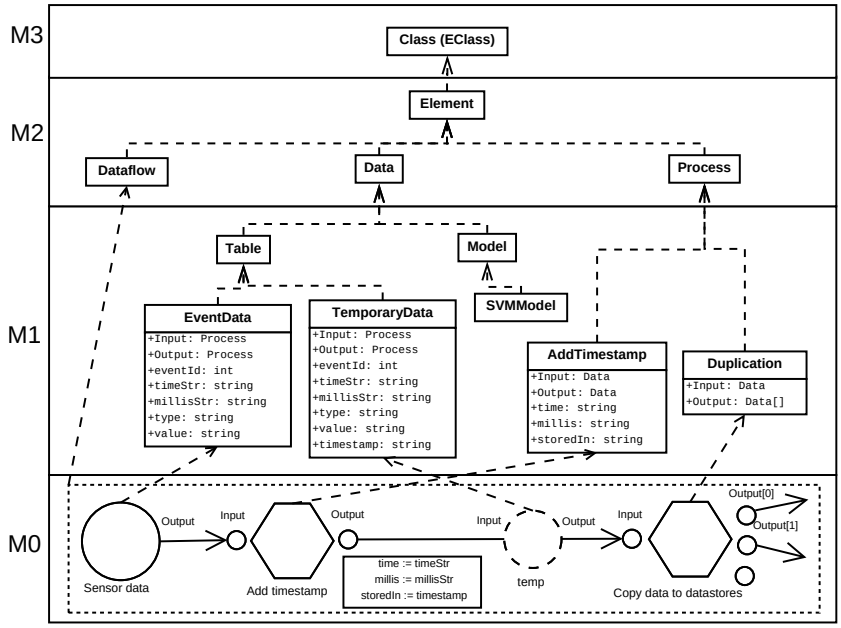


Fig. 2. Hierarchy of dataflow model.

This paper describes practices for achieving multi-level models on EMF. We use a hierarchy of a dataflow model as an example model that is used on graphical editing tools. We design multi-level modeling patterns on EMF, and implement the dataflow model by applying the patterns. Moreover, we attempt to compare the patterns regarding the facilitation of developing both our tool and plugins.

The remainder of this paper is organized as follows. Section 2 describes a model of a graphical editing tool as our motivating example. Section 3 describes patterns for multi-level modeling on EMF. In Sect. 4 we discuss the comparison of the patterns, and our conclusions are presented in Sect. 5.

2 Motivating example: a dataflow model for graphical editing tools

A typical graphical editing tool consists of a palette and a canvas as well as Fig. 1. The palette shows icons representing types of nodes, and the canvas is used to define a diagram by putting a node of the type selected from the palette and drawing an edge between nodes. By using such tool, we can easily develop a data processing application as a flow diagram that consists of nodes and edges representing icons and lines, respectively.

Figure 2 shows the hierarchy of the dataflow model that we want to design. Layer **M3** represents the original Ecore metamodel, and layer **M2** represents the metamodel of the dataflow model. Objects in layer **M2** (i.e., `Dataflow`, `Data` and `Process`) are instances of `Class`. An instance of `Dataflow` composes instances

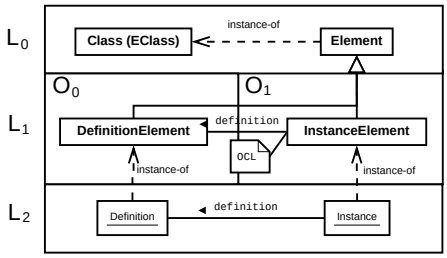


Fig. 3. OCA pattern.

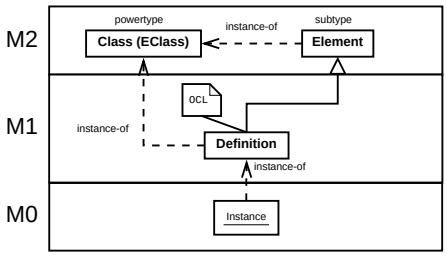


Fig. 4. Powertypes pattern.

of `Data` and `Process`, and represents how data is processed and the order of execution in the processing methodologies as well as the definition in [15].

Layer **M1** represents definitions of types and subtypes of `Data` and `Process` that are displayed on the palette. Classes in layer **M1** are instances of the classes in layer **M2** and have definitions of type names, input ports, output ports and owned properties. In Fig. 2, `Table` and `Model` are instances of `Data`, and `AddTimestamp` and `Duplication` are instances of `Process`. Moreover, `EventData` and `TemporaryData` are subclasses of `Table`, and `SVMModel` is subclasses of `Model`. Those subclasses define their own properties and data schemata for storing databases. A plugin created by a third-party developer defines a new instance of `Process` in layer **M1**, i.e., a new type of nodes in the palette.

Layer **M0** represents an instance of `Dataflow` edited on the canvas in Fig. 1. Objects in layer **M0** are instances of the objects in layer **M1**. Data node `Sensor data` in layer **M0** represents data that is produced and sent by sensors and has the schema defined by `EventData`.

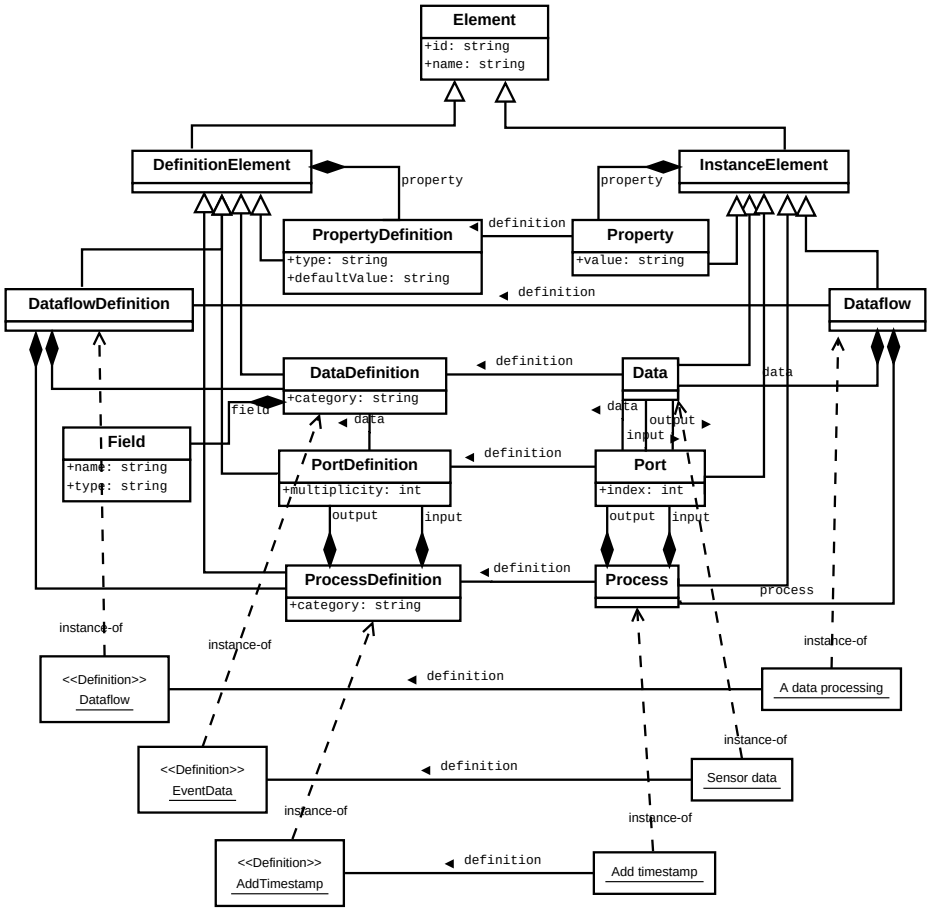
3 Multi-level modeling on EMF

Several multi-level modeling methodologies introduce a new concept of objects. A **clabject** is an object that is both a class and an instance of another class [8]. **Clabjects** sometimes have a **potency** feature that represents the depth to which an attribute can be instantiated [12] and is utilized in **deep instantiation**. In order to achieve multi-level model by only using EMF, we consider that it is difficult to introduce them on EMF, because applying those concepts obviously needs to develop a new modeling editor.

We attempt to implement the dataflow model described in Sect. 2 by applying the following two methodologies: **OCA** and **powertype-based metamodeling**. Figure 3 and 4 show modeling patterns as workarounds for each methodology.

3.1 Model applying OCA pattern

The **OCA** has two dimensions of model layers: *linguistic layers* and *ontological layers*. In Fig. 3, **L** and **O** denotes *linguistic layers* and *ontological layers*, respectively. Layer $\mathbf{L_0}$ contains the Ecore metamodel and class `Element` that is an

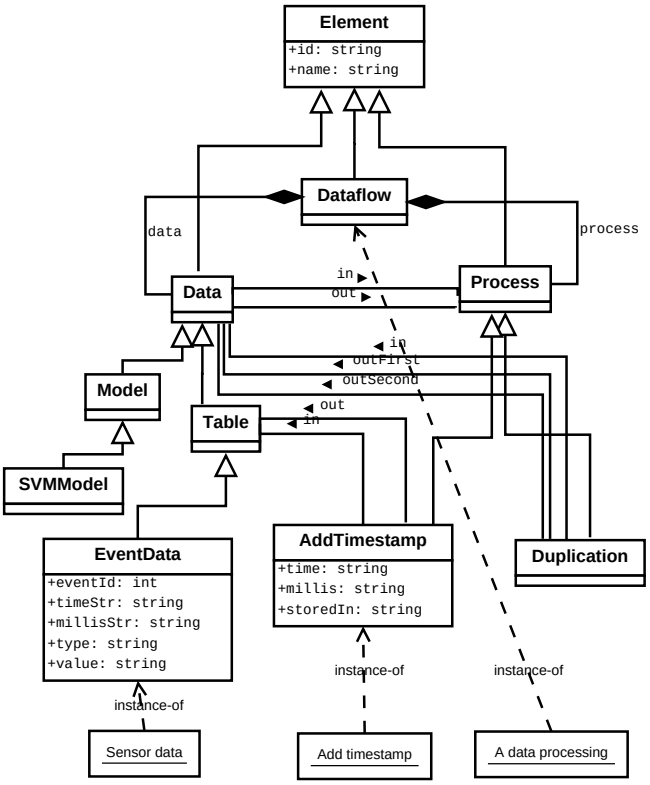



```
-- for instance objects of Data
context Data
  inv DataHasDefinition: definition <> null
  inv DataHasValidProperties:
    definition.property->forAll(i | property->exists(definition = i))
  inv DataHasValidFields:
    definition.field->forAll(name <> null and type <> null)

-- for instance objects of Process
context Process
  inv ProcessHasDefinition: definition <> null
  inv ProcessHasValidProperties:
    definition.property->forAll(i | property->exists(definition = i))

  inv ProcessHasValidInputPorts:
    definition.input->forAll(i | input->exists(definition = i))
  inv ProcessHasValidOutputPorts:
    definition.output->forAll(i | output->exists(definition = i))
...
```

Fig. 5. Dataflow model and excerpt of OCL constraints in OCA pattern.




```
context EClass
  def: isA(typeName : String) : Boolean =
    name = typeName or oclIsKindOf(EClass)
    and oclAsType(EClass).eAllSuperTypes->exists(name = typeName)

-- for subclasses of Data
  inv DataHasNoExtraProcessRefs:
    isA('Data') implies eReferences->forAll(
      eReferenceType.isA('Process') implies name.matches('in|out')
    )

-- for subclasses of Process
  inv ProcessHasValidInputPorts:
    isA('Process') implies eReferences->forAll(
      name.matches('^in.*') implies eReferenceType.isA('Data')
    )
  inv ProcessHasValidOutputPorts:
    isA('Process') implies eReferences->forAll(
      name.matches('^out.*') implies eReferenceType.isA('Data')
    )
...
```

Fig. 6. Dataflow model and excerpt of OCL constraints in powertypes pattern.

instance of class `Class`, for defining elements of ontological layers ($\mathbf{O_0}$ and $\mathbf{O_1}$)in layer $\mathbf{L_1}$. Class `DefinitionElement` in layer $\mathbf{O_0}$ and class `InstanceElement` in layer $\mathbf{O_1}$ defines the type and the instance of elements of the dataflow model, respectively. Layer $\mathbf{L_2}$ contains definition objects and instance objects that are instances of class `DefinitionElement` and `InstanceElement`, respectively.

We represent an *ontological instantiation* relationship by a reference to a definition object. The instance object has a reference to the definition object, and the correctness of the relationship between them is verified by constraints written in Object Constraint Language (OCL).

Figure 5 shows the dataflow model that conforms to the **OCA** pattern. Class `Dataflow`, `Data`, `Process`, `Port` and `Property` are instance classes, which are subclasses of class `InstanceElement`, and all of them respectively have their own definition classes, which are subclasses of class `DefinitionElement`. Object `Dataflow`, `EventData` and `AddTimestamp` are definition objects, i.e., instances of the definition classes. Object `A data processing`, `Sensor data` and `Add timestamp` are instance objects, i.e., instances of the instance classes.

Examples of OCL constraints for instance objects of class `Data` and `Process` are shown in the lower part of Fig. 5.

3.2 Model applying powertypes pattern

Powertype-based metamodeling introduces a powertype that is defined as a type whose instances are types inheriting a subtype [14]. While in the original idea, every object in layer **M1** must be a **clabject** that is both an instance of a powertype and a subclass of a subtype, we define an object in layer **M1** of Fig. 4 just as an instance of a powertype, i.e., class `Class`, and use OCL constraints for defining the relationship between the object and a subtype. We define that the object is regarded as a genuine subclass of the subtype if it satisfies the OCL constraints.

Figure 6 shows the dataflow model that conforms to the **powertypes** pattern. As class `Dataflow`, `Data` and `Process` are subclasses of class `Element`, the hierarchy of all classes are represented as inheritance relationships. Class `EventData`, which is a subclass of class `Table`, has attributes that represent data schema. Class `AddTimestamp`, which is a subclass of class `Process`, has attributes that represent parameters of the process. Class `AddTimestamp` also has an input port and an output port as references to class `Table`, which means that it consumes and produces `Table`-typed data.

Object `A data processing`, `Sensor data` and `Add timestamp` are instances of class `Dataflow`, `EventData` and `AddTimestamp` respectively.

Examples of OCL constraints for subclasses of class `Data` and `Process` are shown in the lower part of Fig. 6.

4 Evaluation

We attempt to compare our modeling patterns, **OCA** and **powertypes**, regarding the facilitation of the following developments: developing our tool by

Table 1. Definition of `AddTimestamp`.

Name	Description
`Input`	a single port that consumes a subclass of `Table`
`Output`	a single port that produces a subclass of `Table`
`time`	a formatted date string, e.g., ‘‘`yyyy-MM-dd hh:mm:ss`’’
`millis`	an integer string of a millisecond value
`storedIn`	a field name to which a timestamp value is assigned

ourselves and developing plugins for our tool by third-party developers. We consider there are a lot of viewpoints regarding the facilitation, but we have not yet completed the comprehensive evaluation from the viewpoints. In this paper, we concentrate the following two viewpoints: model manipulation for our tool and template description for plugins.

4.1 Model manipulation for our tool

Regarding the development of our tool, we focus on how to manipulate the model on the methodology. The **OCA** pattern can utilize the code generation features of EMF, because we do not need to extend metamodels in layer $\mathbf{L_0}$ of Fig. 3. All objects that are added by plugins for new types of data or processes are located in layer $\mathbf{O_0}$, and they can be manipulated by using automatically generated codes. On the other hand, when we apply the **powertypes** pattern, we have to extend the Ecore metamodel dynamically, so it is difficult to utilize the code generation. We have to manipulate objects in layer **M0** by only using the default Ecore APIs that are not intuitive and troublesome to manipulate.

4.2 Template description for plugins

Regarding the development of plugins, we focus on the description of the model-to-text transformation template for process `AddTimestamp` in Fig. 1, 2, 5 and 6. Table 1 shows the definition of process `AddTimestamp`. The process produces a record that is appended a new field named as the string value of `storedIn`. The new field is assigned a string value of a timestamp that is calculated by using `time`, and `millis` of an original record.

Now, we consider a template for producing the following SQL-like processing query.

```
insert into <Output> select <Field of Input>[,<Field of Input> ...],
UDF.timestamp(<time>, <millis>) as <storedIn> from <Input>
```

We use Acceleo, which is an implementation of MOFM2T[6], for generating the query. By applying the **OCA** pattern, the template can be described as follows.

[6] `http://www.omg.org/spec/MOFM2T/`

```
[template public generate(aProcess : Process) overrides generate
? (definition.name='AddTimestamp')]
insert into [output->any(definition.name='out').data.name/]
select [for (input->any(definition.name='in')
.data.definition.field.name) separator(',')][name/][/for]
, UDF.timestamp(
[property->any(definition.name='time').value/],
[property->any(definition.name='millis').value/]
) as
[property->any(definition.name='storedIn').value/]
from [input->any(definition.name='in').data.name/]
[/template]
```

By applying the **powertypes** pattern, the template can be described as follows.

```
[template public generate(aProcess : AddTimestamp) overrides generate]
insert into [out.name/]
select [for (_in.eClass().eAttributes) separator(',')][name/][/for]
, UDF.timestamp([time/],[millis/]) as [storedIn/] from [_in.name/];
[/template]
```

By contrasting those descriptions, the **powertypes** pattern can describe the template more simply than the **OCA** pattern. This is because objects in layer **M1** of Fig. 4 are just models of the Ecore metamodel, so we can directly access the attribute values of their instances. This advantage is valid not only Acceleo but also other EMF-based toolkits, and the fact indicates that the **powertypes** pattern is more usable by following the existing common manners of MOF-compliant modeling frameworks.

5 Conclusion

In this paper, we described the practices for achieving multi-level modeling by only using EMF. We defined modeling patterns of the following two methodologies: **OCA** and **powertype-based metamodeling**. We implemented the dataflow model for graphical editing tools by applying the patterns. By comparing the implementations on the two methodologies, We found the **OCA** pattern is easier to develop our tool than the **powertypes** pattern, but regarding plugins for our tool, the **powertypes** pattern can define model-to-text transformation templates more simply than the **OCA** pattern.

We consider we have to achieve the ease of developing plugins for our tool rather than the ease of developing our tool itself, because increasing the number of plugins can benefit our tool and our communities. Although regarding the simplicity of template descriptions, the **powertypes** pattern is a better choice than the **OCA** pattern, further evaluations from other viewpoints are needed for determining the best pattern.

We hope that the multi-level modeling methodology is standardized adequately following the existing common manners of MOF-compliant modeling frameworks.

References

1. Metamodeling with EMF: Generating concrete, reusable Java snippets, http://www.ibm.com/developerworks/library/os-eclipse-emfmetamodel/
2. Pentaho Data Integration, http://community.pentaho.com/projects/data-integration/
3. RapidMiner, https://rapidminer.com/
4. RunMyProcess, https://www.runmyprocess.com/en/
5. Talend Data Integration, http://www.talend.com/products/data-integration
6. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. Software Engineering, IEEE Transactions on 35(6), 742–755 (Nov 2009)
7. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. Software, IEEE 20(5), 36–41 (Sept 2003)
8. Atkinson, C.: Meta-modelling for distributed object environments. In: Enterprise Distributed Object Computing Workshop [1997]. EDOC'97. Proceedings. First International. pp. 90–101. IEEE (1997)
9. Atkinson, C., Gerbig, R.: Melanie: multi-level modeling and ontology engineering environment. In: Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards. ACM (2012)
10. Atkinson, C., Gerbig, R., Kühne, T.: Comparing multi-level modeling approaches. In: MULTI 2014–Multi-Level Modelling Workshop Proceedings. pp. 53–61 (2014)
11. Atkinson, C., Kennel, B., Goß, B.: The level-agnostic modeling language. In: Software Language Engineering, pp. 266–275. Springer (2011)
12. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: UML 2001The Unified Modeling Language. Modeling Languages, Concepts, and Tools, pp. 19–33. Springer (2001)
13. Henderson-Sellers, B., Gonzalez-Perez, C.: The rationale of powertype-based metamodelling to underpin software development methodologies. In: Proceedings of the 2nd Asia-Pacific Conference on Conceptual Modelling - Volume 43. pp. 7–16. APCCM '05, Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2005)
14. Henderson-Sellers, B., Gonzalez-Perez, C.: On the ease of extending a powertype-based methodology metamodel. Meta-Modelling and . WoMM 2006 pp. 11–25 (2006)
15. Kimura, K., Nomura, Y., Kurihara, H., Yamamoto, K., Yamamoto, R.: Multi-query unification for generating efficient big data processing components from a dfd. In: Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on. pp. 260–268. IEEE (2013)
16. Kimura, K., Nomura, Y., Tanaka, Y., Kurihara, H., Yamamoto, R.: Runtime Composition for Extensible Big Data Processing Platforms. In: 2015 IEEE 8th International Conference on Cloud Computing. pp. 1053–1057 (2015)
17. Neumayr, B., Schrefl, M.: Abstract vs concrete clabjects in dual deep instantiation. In: MULTI 2014–Multi-Level Modelling Workshop Proceedings. pp. 3–12 (2014)
18. Nomura, Y., Kimura, K., Kurihara, H., Yamamoto, R., Yamamoto, K., Tokumoto, S.: Massive event data analysis and processing service development environment using dfd. In: Services (SERVICES), 2012 IEEE Eighth World Congress on. pp. 80–87 (2012)
19. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)