Instance Specialization – a Pattern for Multi-level Meta Modelling

Matthias Jahn, Bastian Roth and Stefan Jablonski

Chair for Applied Computer Science IV: Databases and Information Systems
University of Bayreuth, Universitätsstraße 30, 95447 Bayreuth, Germany
{Matthias.Jahn, Bastian.Roth, Stefan.Jablonski}@uni-bayreuth.de

Abstract. Conciseness is one major quality aspect for meta models. To keep them concise, language patterns like inheritance or powertypes can be used in an appropriated way. With instance specialization we present a further language pattern that rests on the idea of prototypal inheritance (e.g., known from Python or ECMAScript). Generally, it allows for a concept to specialize the instance facet of a particular instance and reuse its configuration. Thereby, all assignments of the latter are inherited by a specializing instance, which can be overwritten in different ways within this instance. Beyond describing the instance specialization pattern, we also introduce a semi-automatic, user-supporting mechanism for applying this pattern to existing meta models.

Keywords: meta modelling, meta model evolution, instance specialization, inheritance, prototypal inheritance

1 Motivation

In the field of software engineering meta models are often utilized to define the abstract and concrete syntax of domain specific modelling languages (DSMLs). Hence, the quality of such a DSML depends highly on the quality of those meta models describing it. The quality of a meta model is influenced by various aspects like conciseness, simplicity and extensibility [4]. For improving these aspects, in recent years several meta modelling patterns like clabjects [1, 2] or inheritance were discovered.

In general, multilevel meta modelling aims in contrast to common programming languages at defining more than two meta levels leveraging the modeller to create a higher degree of abstraction without manually (re-)implementing an instantiation mechanism [6, 17]. In modelling environments or programming languages supporting two meta levels the elements of the meta level are typically called classes or types whereas the elements of the instance level are called objects or instances. For multilevel environments elements can act as both [14]: as a type for an instance level's element and as an instance of another element of a higher meta level. Containing either an instance and a type facet, such elements are often called Clabjects (CLAss + obJECT) [1, 2] or concepts [12]. Hence, such concepts can define attributes in the type facet and assignments to attributes of the concept's type within the instance facet.

Beside instantiation, inheritance is another frequently used pattern to improve quality of meta models symbolizing the “is a” relationship between two different concepts [8]. Nevertheless, this relationship is limited to the type facet of both involved concepts whereas the instance facet is not affected. In modern programming languages like ECMAScript with prototypal inheritance a different pattern occurs, which expresses a specialization of the instance facet. Furthermore, this pattern can be observed in various domains like process modelling (type-usage) [11], car modelling (chapter 4.3) or graphical frameworks [18]. Bridging the described gap in meta modelling, in this paper we introduce the pattern of instance specialization for meta modelling and show how it can be integrated into a meta modelling platform. Furthermore, we present an operator that allows for applying the presented pattern to an existing meta model with full support of model migration.

2 Related Work

The pattern of instance specialization occurs in various domains. Also many programming languages use the paradigm of prototype-based inheritance, e.g. ECMAScript, Ruby, Python, Logtalk or OpenLaszlo. Beside programming languages the pattern of instance specialization is also used in the modelling domain. Lieberman [13] introduces the prototype pattern for object oriented systems and shows the advantages (flexibility) of delegation in contrast to inheritance. As mentioned in the paper [13], inheritance implements sets whereas delegation implements prototypes. Up to our knowledge, in the field of meta modelling there is only one approach that introduces the pattern of instance specialization. Volz presents the pattern of instance specialization in [20].

Other domains like process modelling (type-usage[11]) or graphical model environments [18] also use this pattern. Nevertheless, those approaches mainly need to implement the aimed behavior of the pattern manually since the according environments do not provide instance specialization support.

Additional to the lacking of out of the box support, none approach exists that supports applying instance specialization to an existing model. In the field of meta model evolution several operators were discovered [9, 10, 22] but none of them provide support for the presented pattern.

3 Inheritance and Instance Specialization

In this section we first take a look at traditional type specialization, which is typically called inheritance. Afterwards, we introduce the instance specialization pattern and explain how it can coexist and interact with inheritance.

3.1 Inheritance

The principle of inheritance is an often used pattern for object oriented software design (e.g. [15]). It is generally represented as an “is a” relationship between two concepts.

However, this is problematic, since instantiation is also used for that relationship [8]. The base class is called generalization and the other class specialization. Inheritance influences the structure of the according specialization concept. On the one hand attributes of the generalization are inherited and on the other hand the substitution principle is applied to the specialization, i.e., if an instance of the generalization is expected an instance of the specialization is valid as well.

In the multilevel meta modelling context this definition has to be more precise since each element has a type facet and an instance facet. According to the common semantic of inheritance, inheritance influences the type facets of both involved concepts whereas the instance facet is not affected. Hence, this relation links the type facet of the specialization to the type facet of the generalization with the effect that attributes declared at the generalization are inherited to the specialization.

3.2 Instance Specialization

As explained above, inheritance is a relationship that merely extends the type facet of a generalized concept. Nevertheless, in various use cases a specialization of the instance facet is needed (e.g. Process Modelling [11], Lieberman [13]). Similar to common programming languages, declaring a prototype (the base concept) of a specific concept (the instance specialization) enables inheriting concrete attribute values (assignments). Accordingly, the instance specialization is a relationship between two concepts linking their instance facets. The difference between inheritance and instance specialization was discussed in the programming language community for years (e.g. [13, 19]). Nevertheless, for multi-level meta modelling instance specialization is not limited to the instance level but can be applied to any concept.

To interact with inheritance the substitution principle needs to be extended for instance specialization. Hence, instance specialization defines the substitution principle for the instance facets of both concepts, i.e., if an instance of a concept is expected as an attribute value, an instance specialization of that instance is also a valid value. For example, if concept `A` declares an attribute `attr` with concept `B` as its attribute type. Furthermore, let `InstB` be an instance of `B` and `Special` an instance specialization of `InstB`. Then, each instance of `A` may assign `Special` (and `InstB`) for `attr`.

Instance specialization is hence a new relation between two concepts that does not exclude an inheritance between those two concepts. Since both relationships use different concept facets they can interact harmless with each other (on a conceptual point of view).

Overwrite Behavior.
The core idea of this pattern is an inheritance of assignments that are defined at the prototype of an instance specialization concept. However, these assignments may be overwritten by the instance specialization if needed. Since this behavior is not always suitable, the prototype can define whether and how an assignment can be overwritten by an instance specialization.

To configure this, each prototype can declare the overwrite behavior for every assignment. The possible strategies are:

- **Type 0 (forbidden):** An instance specialization is not allowed to overwrite the value of its prototype. The assigned value is always inherited from the prototype.
- **Type 1 (normal, default type):** The prototype's value for the specific attribute can be overwritten in any way by an instance specialization. If no type is specified explicitly type 1 is applied for the particular assignment. This type is also supported by languages like ECMAScript [7] that provide prototypal inheritance as an idiom.
- **Type 2 (limited):** The assignment at the prototype defines the domain of all values that are assigned at an instance specialization, i.e., the values of the instance specialization are a subset of the values defined at the prototype. A similar type is shown by Pirotte et.al. [16] with the difference that the type is not declared at the assignment but at the attribute and thus acts for all according assignments. This type is restricted to assignments which base upon a multi-valued attribute.
- **Type 3 (append):** The value of an instance specialization is appended to the value of the prototype for getting the concrete attribute's value.
- **Type 4 (prepend).** Similar to type 3 the concrete value of the instance specialization is a result of the assigned value together with the prototype's value. Instead, the instance specialization's value is prepended to the prototype's value.

The two types 3 and 4 are only applicable to assignments whose attribute is multi-valued or a string attribute. For strings, assignments within an instance specialization results in a concatenation. For collections, however, it leads to appending or prepending the value(s) of the specialization concept to the prototype's values. Apparently, the both types are equal if they are not ordered within the according collection (e.g., a set). A similar declaration (with some differences) of such types was introduced by Volz [20] but he limits the overwrite behavior types 3 and 4 to strings. The information about the overwrite behavior is stored within the linguistic meta model [21], which is an implementation of the orthogonal classification.

Example.
A typical scenario for instance specialization could be a model for cars. Often manufacturers offer their cars in a base series that can be specialized in various ways. In **Fig. 1** we give a possible example. At level `M1` we have modeled the concept `Car`, which is a representation of the real world counterpart and declares the attributes `typeName`, `manufacturer` and `releaseDate`. Each car may have some equipment (concept `Equipment` with a relation to `Car`).

At level `M0` an instance model is shown. Therein, a car `Ibiza` is modeled that has the name `"Ibiza"` (according assignment to `typeName`), produced by `Seat` (assignment to `manufacturer`), was released on the 1st of January 2009 (assignment to `releaseDate`) and may be equipped with the packages `ABS` and `ESC` (assignment to `equipment`). Each of these attributes defines a specific overwrite behavior. Since every instance specialization of the `Ibiza` base series will be produced by the same manufacturer (`Seat`), the attribute `manufacturer` declares the overwrite type 0. In

contrast to that `releaseDate` can be overwritten in any way. Owing to the fact that a new special car series may only have a subset of all possible equipment packages, the overwrite behavior of `equipment` is set to type 2 (limited). At last, `typeName` can be extended by any instance specialization. That is why the according assignment of `Ibiza` has the overwrite behavior type 3. Additionally, an instance specialization `IbizaReference` is available, which concretely uses the according prototype `Car`. The relationship between these two concepts is equipped with an arrow labeled with `<<concreteUseOf>>` to designate the instance specialization.

The instance specialization `IbizaReference` is a special series of Ibiza that has a special type name ("Ibiza Reference"), a different release date and the `ABS` equipment package as standard equipment. Because of that, `IbizaReference` overwrites `releaseDate` with the value `"2010-04-01"`, sets `typeName` to `" Reference"`, which implicitly results in `"Ibiza Reference"`, and finally chooses `ABS` for `equipment`. The attribute manufacturer cannot be overwritten and is hence inherited from the prototype `Ibiza`.

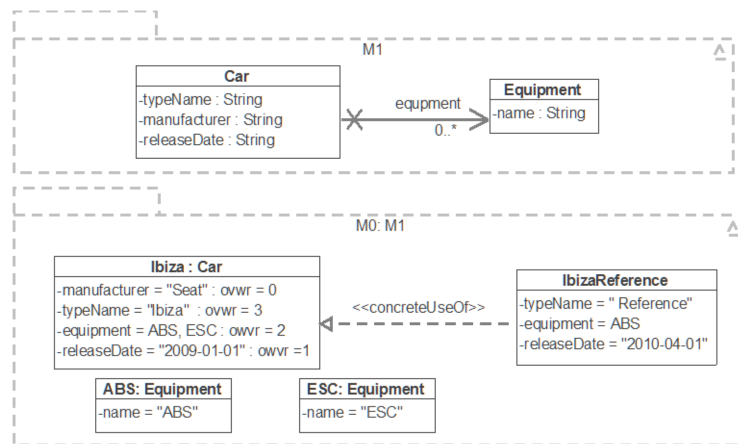

Fig. 1. Car model example

4 Extract Prototype

In this section we present a way for (semi-)automatically introducing an instance specialization into an existing model. This mechanism is supplied by a specific operator, which extracts a prototype out of instances that are similar. It can be seen as a counterpart of the extract super class refactoring method that is provided by many IDE or modelling systems [10, 22].

4.1 Overview

The *Extract Prototype* Operator creates a prototype out of similar instances of one type. Thereby, the operator sets the assignments at the prototype according to the chosen overwrite behavior and updates each instance specialization assignment if necessary.

4.2 Operator process

The operator process is shown by **Fig. 2**. Therein all steps or decisions that need user interaction are highlighted in black. At the beginning the operator is invoked on a concept `Base`, which instantiates another concept `Type`. In the first step this instantiated type is ascertained together with all instances of it. Out of this set a subset of all future instance specializations (including `Base` by default) is chosen. In the following we call this subset `Instances`. Afterwards, the operator fetches all attributes declared at `Type` that can be set at `Base`.
In the next step, a subset (`AttrsToSet`) out of these attributes have to be chosen, which will be set on the prototype. Of course, all attributes of `Type` that are mandatory (multiplicity `1` or `1..*`) have to be part of this subset. After that, for each attribute of `AttrsToSet` the according assignments that were declared at an element of `Instances` are ascertained since they influence the attributes value at the prototype. Subsequently, the prototype is created. Thereby, the prototype's name is defined and an instantiation to `Type` is created. After that, an assignment for each attribute of `AttrsToSet` is created at the new prototype and together with it, the overwrite behavior is defined by the user. In the last activity of the operator the value of each assignment is determined depending on the chosen overwrite behavior:

- **Type 0 (forbidden)**: If a change of the assignment's value is forbidden at an instance specialization, a specific value that was assigned at an element of `Instances` has to be chosen, which acts as new value for the prototype. Since the assignments of all elements of `Instances` are not valid anymore, they will be deleted afterwards.
- **Type 1 (normal)**: If this type is selected, each instance specialization may overwrite the attributes value in any way. Hence, just a selection of the new value out of those made at the elements of `Instances` for the prototype is needed. Then, all assignments that are equal to the chosen value and those that should be deleted (user selection) are removed from the according elements of `Instances`.
- **Type 2 (limited)**: For this type all assignment values of `Instances` are inserted into a set that acts as the resulting value for the prototype. Here, no further adaption is needed since all instance specializations values lie in the created domain.
- **Type 3 (append) and Type 4 (prepend)**: In this case a base value for the prototype has to be chosen. This value may consist of some values or a substring that was assigned at an element of `Instances`.

In the last step the relationship for the instance specialization is created. Thereby, all instantiations of `Instances` are deleted since an implicit instantiation exists via the instance specialization. Furthermore, all assignments of new instance specializations

(`Instances`) are deleted if overwriting is forbidden (type 0) or are adapted (respectively new chosen) if a base value was selected for the prototype assignment (type 3 and 4).

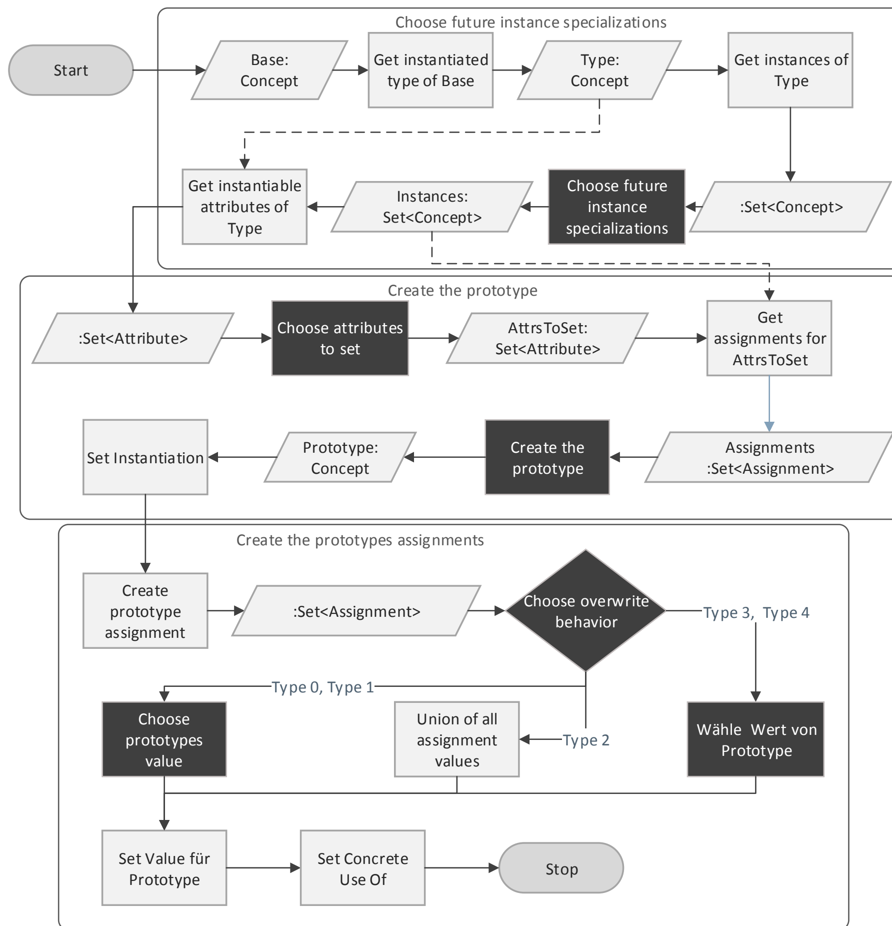

Fig. 2. Process of the Extract Prototype Operator

4.3 Example

In **Fig. 2** a model is shown on which we will demonstrate how the operator works. Therein a DSML for describing cars is presented. Hence, at the top level `M1` a concept `Car` is modeled representing the according real world element with a `manufacturer` attribute, a type (`typeName`), a release date (`releaseDate`) and a relationship to `Equipment` (attribute is called `equipment`). In general, a car may have various equipment parts.

One level below (M0) an instance model is given containing two instances of Car (IbizaStyle, IbizaReference). Both cars are produced by the manufacturer "Seat" and their type is almost equal to their concept's name. Additional to the both Car instances, the M0 level contains two instances of Equipment (ABS, ESC) representing the anti-blocking system and the electronic stability control of a car. According to the modeled scenario (not the real life), IbizaReference provides only ABS whereas the IbizaStyle also has an ESC.

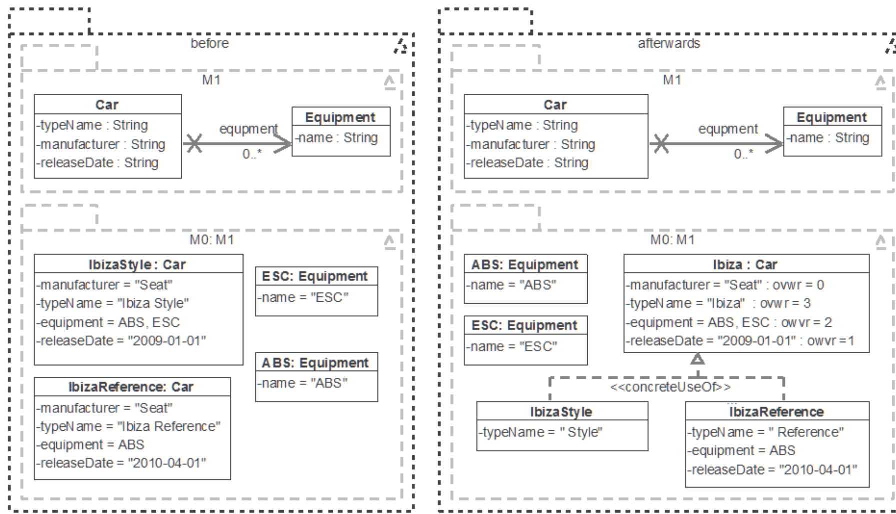

Fig. 3. Application of the operator to the car model example

Since both instances of Car can be seen as two different instance specializations of the car Ibiza we now invoke the *Extract Prototype* operator to create this prototype. That is why we select IbizaReference and call the operator on it. Afterwards, all instances of Car are gathered by the operator and we decide to add IbizaStyle to the set of future instance specializations. In the next step, all attributes of Car are calculated (manufacturer, typeName and equipment) that can be instantiated at IbizaReference. In our example we decide to set all of these attributes at the prototype and hence, all assignments relating to these three attributes of IbizaStyle and IbizaReference are collected. Subsequently, the new prototype can be created, which is called Ibiza and which becomes an instance of Car. Next, all assignments are created at the prototype with the following overwrite behaviors:

- manufacturer should not be overwritten by instance specializations and gets thus type 0 (forbidden)
- typeName can be extended by an instance specialization with any further string value and consequently gets type 3 (append)
- equipment gets type 2 (limited) because the prototype should declare all possible equipment parts

- `releaseDate` gets type 1 since the date can differ in each car series.

Owing to those overwrite behaviors, the assignments of `IbizaStyle` and `IbizaReference` for `manufacturer` are deleted and for `typeName` and `releaseDate` a value for `Ibiza` is selected (`"Ibiza"` and `"2009-01-01"`). According to that, the assignments of the two instance specializations for `typeName` are adapted to the new values `" Style"` or `" Reference"` respectively and thus the original value is retained virtually. The assignment for `releaseDate` of `IbizaReference` is not affected whereas the assignment of `IbizaStyle` is deleted because the value is equal to the prototype's value. For `equipment` the resulting value for `Ibiza` is the union of all values of the future instance specializations and hence `{ABS, ESC}`. The other assignments need not be adapted here since they are valid anymore. In the last step the instantiation of `IbizaStyle` and `IbizaReference` to `Car` is deleted and the instance specialization to `Ibiza` is created. Finally, the operator terminates. The resulting model is free of any redundant attribute values, which is a great benefit especially in case of models.

5 Conclusion and Outlook

Instance specialization as described in this paper is a language pattern that solely impacts on the instance facet of concepts. It enables users to easily define a default configuration regarding a certain use case, which then can be adapted through instance specialization for specific scenarios. We greatly utilize this patterns for the definition and usage of concrete syntaxes for DSMLs (similar to [18]). Thereby, a concrete syntax is determined as an instance of a given meta model designed for this particular purpose. Later on, the concepts of this syntax can be instance-specialized to shape the visual parts of concrete diagrams or documents. As a result, only one meta model is required to formulate diagrams and documents as well as the concrete syntax they base upon. Above all, the common model base extremely reduces the implementation effort for building a dedicated processing module, which can handle both, concrete syntax and all associated instance specializations, in an analogous manner. A suchlike DSML tool as well as the *Extract Prototype* operator introduced in section 4 is implemented on top of the Model Workbench [5], a web-based modelling platform. Since each atomic and complex model manipulation action has to be encapsulated by an operator, they constitute the core of this platform. Besides, the Model Workbench provides support for further multilevel meta modelling patterns (e.g., deep instantiation [3] and materialization [16]). For the future, we plan to offer user-guided support for the introduction of these patterns by means of suitable operators.

References

1. Atkinson, C.: Meta-modelling for distributed object environments. Proceedings of the 1st International Conference on Enterprise Distributed Object Computing (EDOC '97). pp. 90–101 IEEE (1997).

2. Atkinson, C., Kühne, T.: Meta-level independent modelling. Int. Work. Model Eng. 14th Eur. Conf. Object-Oriented Program. 12, 16 (2000).
3. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. «UML» 2001—The Unified Model. Lang. Model. Lang. Concepts, Tools, Lect. Notes Comput. Sci. 2185, 19–33 (2001).
4. Bertoa, M., Vallecillo, A.: Quality attributes for software metamodels. Proceedings of the In 13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2010) (2010). (2010).
5. Chair of Applied Computer Science IV - University of Bayreuth: Model Workbench, http://www.ai4.uni-bayreuth.de/de/research/projects/003_ModelWorkbench/index.html.
6. Demuth, A.: Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking. Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. 452–455 (2011).
7. Flanagan, D.: JavaScript: The Definitive Guide (Definitive Guides). O'Reilly Media, Inc, Sebastopol, CA (2011).
8. Frank, U.: Thoughts on classification/instantiation and generalisation/specialisation, http://www.econstor.eu/handle/10419/68462, (2012).
9. Herrmannsdoerfer, M. et al.: An extensive catalog of operators for the coupled evolution of metamodels and models. Softw. Lang. Eng. Lect. Notes Comput. Sci. 6563, 163–182 (2011).
10. Herrmannsdoerfer, M.: Evolutionary Metamodeling. PhD Thesis, Fakultät für Informatik, Technische Universität München (2011).
11. Jablonski, S., Bussler, C.: Workflow management: modeling concepts, architecture and implementation. International Thomson Computer Press (1996).
12. Jahn, M. et al.: Remodeling to Powertype Pattern. Proceedings of the Fifth International Conferences on Pervasive Patterns and Applications (PATTERNS 2013). pp. 59– 65 (2013).
13. Lieberman, H.: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Conference proceedings on Object-oriented programming systems, languages and applications (OOPLSA '86). pp. 214–223 (1986).
14. Odell, J.: Power types. J. Object-Oriented Program. 7, 2, 8–12 (1994).
15. OMG: Unified Modeling Language (OMG UML)-Infrastructure. Available http//www. omg. org/spec/UML/2.4.1. August, (2011).
16. Pirotte, A. et al.: Materialization : a powerful and ubiquitous pattern abstraction. Proc. 20th Int. Conf. Very Large Data Bases (VLDB '94). 630–641 (1994).
17. Roth, B. et al.: IT-as-a-Service for Building Virtual Research Environments. Proceedings of the 2rd International Conference on Cloud Computing and Service Science. , Porto, Portugal (2012).
18. Roth, B.: Konzeption und Implementierung eines generischen Modellierungswerkzeugs zur Unterstützung der domänenspezifischen Prozessmodellierung. (2010).
19. Stein, L.A.: Delegation Is Inheritance. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA). pp. 138–146 , Orlando, Florida, USA (1987).
20. Volz, B.: Werkzeugunterstützung für methodenneutrale Metamodellierung. PhD Thesis, Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth (2011).
21. Volz, B., Jablonski, S.: Towards an open meta modeling environment. Proceedings of the 10th Workshop on Domain-Specific Modeling - DSM '10. p. 1 ACM Press, New York, New York, USA (2010).
22. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. ECOOP 2007 – Object-Oriented Program. Lect. Notes Comput. Sci. 4609, 600–624 (2007).