Exploring Multi-Level Modeling Relations Using Variability Mechanisms

Iris Reinhartz-Berger[1], Arnon Sturm[2], and Tony Clark[3]

[1]Department of Information Systems, University of Haifa, Israel
[2]Department of Information System Engineering, Ben-Gurion University of the Negev, Israel
[3] Department of Computing, Sheffield-Hallam University, UK

iris@mis.hevra.haifa.ac.il, sturm@bgumail.bgu.ac.il,
t.clark@shu.ac.uk

Abstract. Over the last decade multi-level modeling (MLM) approaches have been addressing the need for relaxing the strict constraints on intra- and inter-layer type-instance relationships that are imposed by traditional approaches to meta-modeling. In this paper we explore MLM approaches in the context of Software Product Line Engineering (SPLE), propose a meta-language, and show how it can represent three commonly used variability mechanisms - configuration, parameterization, and template instantiation - within the context of MLM. By this we contribute to simplifying the representation of complex relationships in current MLM approaches and to the formal definition of SPLE variability mechanisms utilizing MLM concepts.

1 Introduction

Promoting models as the primary artifacts in software development, early approaches to Model-Driven Engineering (MDE) refer to four layers of abstraction: data (M0), model (M1), meta-model (M2), and meta-meta model (M3) where elements of Mn are instances of types defined at level Mn+1. Of those layers, three are practically used by MDE practitioners: M2 to build metamodels for general purpose modeling languages (*e.g.*, UML) or Domain Specific Modeling Languages (DSML), M1 to instantiate those metamodels in the form of models, and M0 to process instances of the models. Together these three levels are called a 'golden-braid' [11] and traditional approaches to modeling have used two occurrences (M0-M2 and M1-M3) whilst maintaining strict separation between elements of different levels. Furthermore, most traditional approaches follow a similar definition of the type-instance relationship between levels whereby, for example, an object at M0 structurally conforms to a class at M1 when all the object's properties have names and values corresponding to the names and types defined by the class attributes.

Recently, multi-level modelling (MLM) approaches aim to extend this architecture by enabling modelling at an arbitrary number of levels [12], and propose variations and liberalizations of the traditional type-instance relationship between levels. Current MLM approaches relax these limitations so that multiple golden-braid occurrences

can co-exist within the same model to support an arbitrary number of levels. Type-instance relationships in these cases can be expressed via concepts, such as clabjects [2], power types [1521], and deep meta-modeling [24].

Software Product Line Engineering (SPLE) [10, 22] can be considered as dealing with multi-levels. SPLE uses families of products and subsequent adaptation of each family to produce a single model describing a particular product in the line. Families represent multiple variations in a single definition and adaptation involves the selection of choices amongst the variations and potentially modification and extension of these choices. The different levels handled in SPLE can be classified into domain and application engineering [19]. The *domain engineering* layer includes activities and tasks to create and handle core assets, *i.e.*, artifacts that are intended to be (re)used by more than one product in the family. These assets include both common and variable parts. The *application engineering* layer consists of activities and tasks to adapt and tailor core assets in order to satisfy particular requirements of the products at hand. The reuse between the layers is done systematically applying mechanisms, commonly termed variability or reuse mechanisms [5, 17]. Examples of such mechanisms are configuration, parameterization, and template instantiation.

In this paper, we aim to explore the potential relations between the two worlds – SPLE and MLM. Our hypothesis is that the elements in the SPLE families correspond to *types* and the elements in the particular products correspond to *instances* of types. The relationships derived from the variability mechanisms may refine the type-instance relationship that is variegated and liberalized by current MLM approaches. Thus, the contribution of the paper has two aspects. To MLM, SPLE variability mechanisms can be used to simplify the representation of complex relationships in current MLM approaches, and to SPLE – MLM concepts can be used to more precisely define SPLE variability mechanisms.

In the sequel, Section 2 reviews variations of and extensions to the type-instance relationship in existing MLM approaches. Section 3 discusses commonly used SPLE variability mechanisms – configuration, parameterization, and template instantiation – demonstrating their use. In Section 4 we discuss the ability to identify the relations derived from the variability mechanisms as being special cases of the type-instance relationship. Finally, in Section 5 we conclude and set the ground for future research.

2 Related Work

Recent modeling language research has addressed the limitations of traditional strict-modelling approaches that impose boundaries between elements from different levels of the golden-braid. Several researchers have begun to explore variations on the type-instance relation that is intrinsic to most modeling languages and to relax traditional strictness with the aim of providing a richer notion of 'type' within a model. This section reviews the current advances, broadly categorized as 'multi-level modeling' (MLM), towards this aim.

MLM approaches have been suggested for a variety of uses, including designing models for the use of non-modelers [14] and developing modeling tools [16]. In many cases there is a requirement to allow types and their instances to co-exist, for example

enactment [16] of business processes requires the type of a process and its active instance to be co-represented.

Traditional approaches to the type-instance relationship often focus on classes and their instances. UML, for example, defines that an instance is related to a type by 'instance-of' when the class structurally and behaviorally defines all features of the instance. The relationship may hold between a class and an object or a meta-class and a class. The semantics of a language is usually defined in terms of the 'instance-of' relationship both by intrinsic rules that hold and by (at least in UML) constraints that are expressed by the modeler and attached to the type. Therefore, researchers addressing issues related to model-based language engineering provide mechanisms that affect the intra- and inter-level type-instance relationships.

To address the dual role that a certain element plays in MLM approaches (as an instance of the higher level and a type of the lower level), Atkinson [2] coined the term *clabjects* to simultaneously refer to classes and objects. He further suggested a potency-based multi-level approach to support deep meta-modeling [3]. Following that approach, each element is assigned with a potency number, which indicates the number of levels in which the element can be instantiated. A special case of potency is the intrinsic features, suggested by Frank [13], in which the potency number is one.

Observing that an element may be an instance of two elements residing in different levels, Atkinson and Kuhne [34] suggest the notion of orthogonal classification architecture (OCA). In OCA, elements can be instantiated along the linguistic dimension and orthogonally along the ontological dimension. The linguistic dimension refers to instantiation across levels whereas the ontological dimension refers to instantiation within levels.

Another approach to MLM that does not involve mixing elements from different type levels is to use a pattern to encode the relationship between a meta-class and the class that acts as its instance. Gonzalez-Perez and Henderson-Sellers [15] utilize the notion of power types [21] by introducing the concept of a power type pattern which exists at a single level but represents elements from different levels. The power type pattern is defined as "a pair of classes in which one of them (the power type) partitions the other (the partitioned type) by having the instances of the former be subtypes of the latter." This allows the modeler to capture the semantics of the type-instance within a strict-modelling framework.

To support information integration in heterogeneous information systems, Jordan *et al.* [18] suggested modelling primitives that extend standard specialization and instantiation mechanisms. Particularly, they distinguish between specialization by extension (that supports adding attributes, associations, or behavior) and specialization by refinement (that supports adding granularity to the description), as well as between standard instantiation (in which all attributes must be assigned a value from their domain) and instantiation with extension (which enables adding attributes, behavior, and so on). They further introduce the subset by specification relation for representing "the existence of a class of specification construct that identifies particular subtypes of another type", the membership relation, and the specification by enumeration relation that describes how the extensions of sets of entities are related.

In summary, the need for flexible modeling technologies has led a collection of researchers to seek ways to relax traditional strict-modelling and to open up the 'type-instance' semantics to the modelers. Most recent advances have sought to mix types and instances and have allowed structural features to be annotated in order to influence their instantiation semantics. We see that there is a similarity with the aims of SPLE variability mechanisms, as they require mixed type-levels and offer control of instantiation through variability.

3 Exploring Alternatives for Relationships in MLM

As noted, frameworks of SPLE commonly distinguish between domain and application engineering (*e.g.*, [19], [23]). In both layers, modeling plays a central role to analyze and design artifacts. As an example, consider the model depicted in Fig. 1. The left part of the model describes a library management system (LMS) or a family of such systems. Respectively, this level can belong to the application or domain engineering layers. The model refers to books (titles) written by authors, as well as to the actual book copies that can be checked out and have (physical) location. A book may have up to n copies.

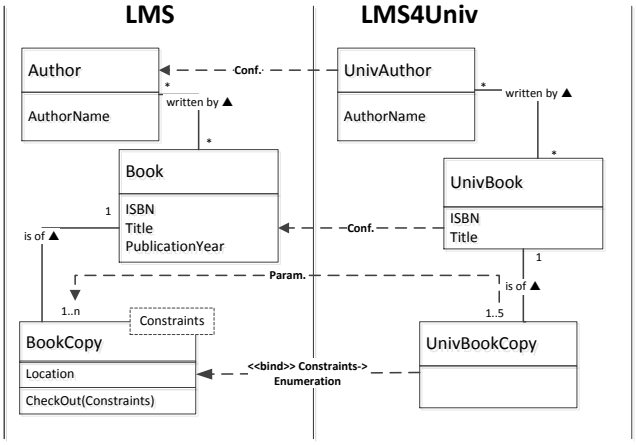

Fig. 1. The LMS example

The second level in our example, partially described in the right part of Fig. 1, refers to library management system(s) in universities (LMS4Univ for short). Here again, the level can be interpreted as specifying a (narrower) domain or an application, and as such it can be considered as an adaptation of the LMS model. Checking-out book copies is constrained according to the (enumerated) user type, *e.g.*, student *vs*. staff, and a book may have up to 5 copies (for economic reasons). In LMS4Univ books have no publication year information.

Note that the two levels depicted in Fig. 1 could be part of a larger chain of levels that includes, for example, a wider domain of Check-In Check-Out applications, or sub-classes of LMS4Univ applications.

For traversing from a higher level to a lower level in SPLE, different variability mechanisms are commonly utilized. These are actually techniques applied to adapt core assets developed in the domain engineering layer to the context of particular products (*i.e.*, artifacts in the application engineering layer). However, they can be used for adapting domains or applications as well. Over the years, different variability mechanisms have been suggested for different development stages, *e.g.*, [1] for implementation, [5], [17] for architecture design, and [6], [7] for reference modeling. We mention here only three common variability mechanisms (see Table 1 for definitions and demonstration of their use through the LMS example):

- In *configuration*, elements of the higher level are selected to be included in the lower level. Partial selections are possible, as in the case of UnivBook which selects only 2 out of the 3 attributes of Book: ISBN and Title.
- *Parameterization* supports assigning values to parameters defined in a higher level. The assignment is done in a lower level. In our example, the maximal number of copies of a certain book is assigned to 5 in LMS4Univ.
- *Template instantiation*, which, in contrast to parameterization that deals with value assignment, deals with type adaptation, is exemplified by constraining book copy check out with enumeration type (which represents user type, *e.g.*, student vs. staff).

Table 1. Common variability mechanisms, their definitions, and use

Variability Mechanism	**Definition**	**Example**
Configuration	Choice between alternative functions and implementations [17]; Modifying selected elements of a core asset based on predefined rules that refer to specific requirements or situations [6, 7].	UnivBook in LMS4Univ (with respect to Book in LMS).
Parameterization	Variation points for features [17]; Data items serving as arguments for distinguished software behavior [5].	Up to 5 UnivBookCopies in LMS4Univ (with respect to up to n BookCopy's in LMS).
Template Instantiation	Type adaptation or selecting alternative pieces of code [17]; Enables filling in product-specific parts in a generic body [5].	UnivBookCopy in LMS4Univ (with respect to BookCopy in LMS).

Next, we explore to what extent the aforementioned mechanisms can be represented by the type-instance relationship and its variations and by this – contribute to simplifying the representation of complex relationships.

4 Variability Mechanisms and Type-Instance Relationships

Our claim is that configuration, parameterization, and template instantiation can be viewed as special cases of the type-instance relationships in the context of MLM. To expand this claim, we first provide a core meta-language that supports our MLM approach.

Consider the simple example shown in Fig. **Error! Reference source not found.**. It uses a MLM approach to model both parametric (as manifested by parameterization or template instantiation) and configurable classes. This is done via the relationship 'of' that works consistently in all cases that are shown. A class defines constraints that must be satisfied by its instances. The relationship 'of' is a declarative statement that the object at its source satisfies the constraints of the class at its target. In the case of BookCopy, ParametricClass is used to model the type parameter used for the method CheckOut. In the case of the UnivBook, the class Book is a *family* specifying that publication year is an optional attribute of book. Therefore, our approach is based on a use of the type-instance relationship which is supported through the consistent use of constraints, the implication of which is a uniform representation for all model-elements. To achieve this *everything is an object* [16].

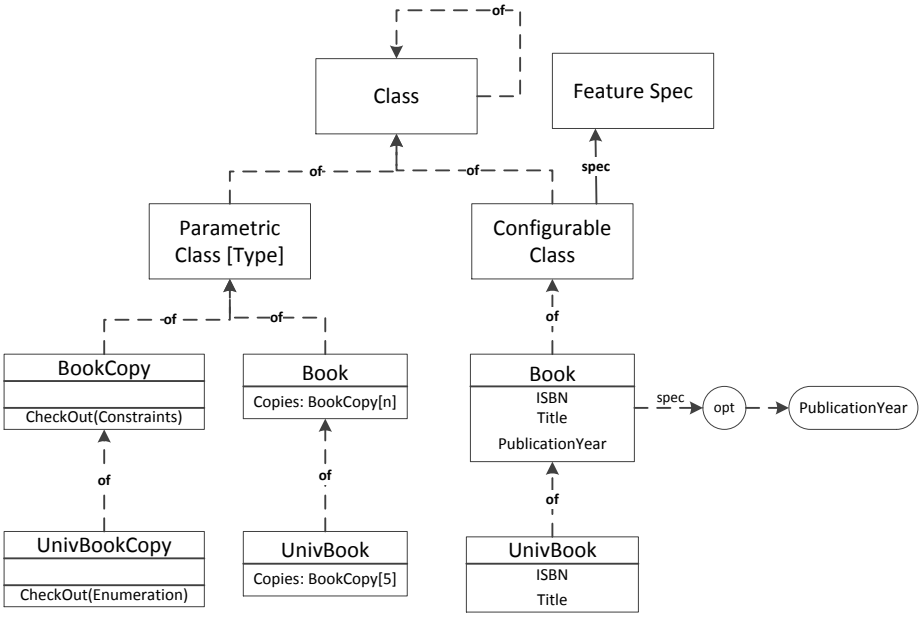

Fig. 2. Configuration, parameterization, and template instantiation expressed via type-instance relationship

Fig. 2 shows a model that contains a kernel meta-language and its extension to support configuration, parametrization, and template instantiation (the last two referred as parametric). Since the basis of our approach is a self-describing meta-language in which everything is an object, the root class is Object. Furthermore, alt-

hough only Class is shown as explicitly inheriting from Object, all classes in the language inherit from the root. To simplify the representation we consider only classes with attributes, hence the use of directed relationships between classes.

Since everything is an object it is possible to access the internal data of any model element through reflection via the attribute named ‘slots’ defined by Object. Inheritance is supported though the ‘parents’ attribute defined by Class and the implication is that all attributes are inherited. An important feature of the approach is defined using constraints on classes. A constraint is a predicate whose ‘check’ operation is supplied with a candidate object.

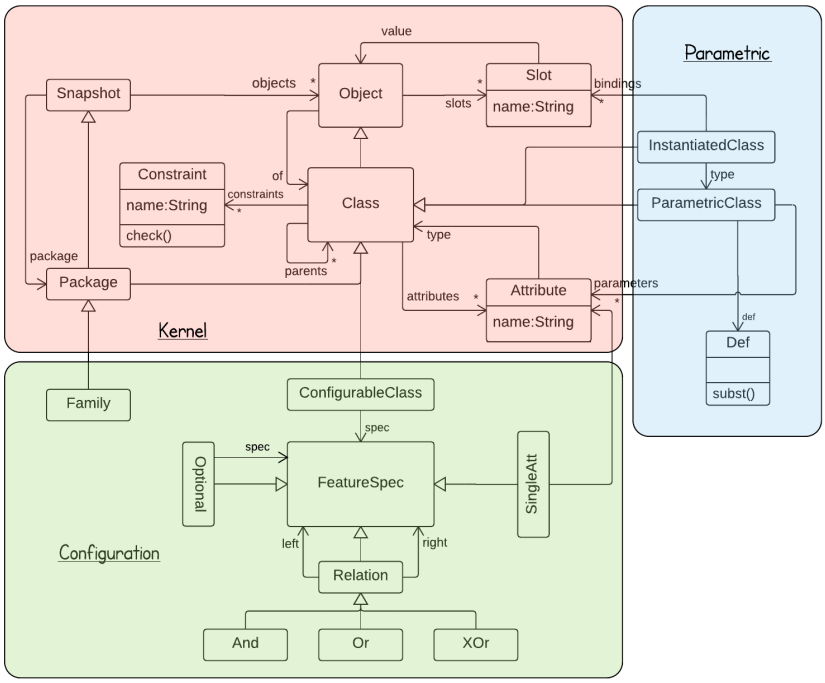

Fig. 2. A Kernel Language

A language definition relies heavily on constraints to specify the ‘of’ relationship that holds between a class and its instances. A key meta-circular constraint that is defined by Class can be paraphrased: ‘An object c is a valid class if it enforces all of its constraints c.constraints when checking o:c for any object o’. Clearly, this constraint allows Class to classify itself and, because of the universal object representation, the object ‘o’ could be a ground instance, a class, or a meta-class.

A snapshot is a container of objects; it can be used as the basis of a wide range of containers where specializations of Snapshot introduce constraints that must hold for the contents. A package is a container of classes (with the associated implied constraint on Package) and a snapshot links to a package that contains the classes that type the contained snapshot-instances. Again, this meta-circularity helps us to ensure that models are objects and that MLM principles work over all levels.

The ‘of’ relationship between a class and an object is defined by the constraints on the class. Since a class may be an extension of the base class Class, it is possible that the constraints used to define a particular occurrence of the relationship ‘of’ might use slot-value information other than those slots defined by Class. Therefore, the ‘of’ relationship can be *overloaded* by the language designer as described below.

Configuration can be expressed via a configurable class that specifies options such that an instance of the class has chosen consistently amongst the available options. Therefore an ‘of’ relationship can hold between a configurable class and its instances if the constraints on the class require the features of each instance to be consistent with the options of the class. Recall that ‘of’ may hold between a ground instance and a class, or a class and a meta-class. Therefore, we may define configuration at any level.

Fig. 2 also shows that configuration is supported via Family – an extension of Package with the implied constraint that a Family is a container of configurable classes. A configurable class defines feature-specifications that are Boolean combinations of attributes. The constraints on a configurable class require that any candidate instance be a class and that the features of the class be consistent with the specification. Therefore, configuration is modeled as a form of type-instance, where the constraints match attributes in the class against feature-specifications in the meta-class.

Parametric model elements define formal parameters or templates that range over element-definitions. When supplied with model elements as actual parameter values or types the formals are consistently replaced within the body of the definitions in order to produce new model elements. Note the term ‘consistently’: the new model elements view the parametric version as a type whose constraints must be satisfied.

Table 2. A MLM Representation for Variability Mechanisms

Variability Mechanism	MLM Instantiation Constraints and Examples
Configuration	*Checks that the structure of the instance is consistent with the variability specified by the type.* Fig. 1 shows LMS as a family with Book as an instance of ConfigurableClass with PublicationYear as an optional attribute. The class UnivBook is a type-consistent instance, and therefore a configuration of Book.
Parameterization	*Assigns a value to a type* ***to create an instance of that type***. Fig. 1 shows that LMS4Univ assigns the value 5 to the parameter n, appearing in LMS and specifying the maximal number of BookCopy associated to a single Book.
Template Instantiation	*Assigns a value to a type* ***to create another type***. The class BookCopy is parametric with respect to the parameter Constraints and the binding of Constraints to Enumeration is shown to produce the instantiated class UnivBookCopy (details of the definition are omitted).

Fig. 2 shows an extension of Kernel with features for parameterization and template instantiation. A parametric class has a collection of typed parameters and a definition. The definition ranges over all model elements and supports an operation ‘subst’ that is supplied with some bindings for the parameters and produces a collection of model elements via consistent substitution. An instantiated class is a normal class that is associated with some parameter bindings. A new constraint on an instan-

tiated class requires that the attributes of the class are consistent with the definition of its class after the bindings have been substituted. Therefore, parameterization and template instantiation are modeled as a form of type-instance, where the constraints match bindings against formal parameters and substitution into a body.

In Table 2 we demonstrate the use of the Kernel Language for applying the variability mechanisms.

5 Conclusions and Future Work

MLM approaches have been proposed in order to relax the traditional strictness requirements on inter- and intra-level type-instance relationships. However, while these proposals are formal, they address the representation of complex relationships to a limited extent. SPLE, on the other hand, distinguishes between different relationships, introducing a variety of variability mechanisms that are more intuitive to the modelers but are less formal in their definitions. The current paper addresses this tradeoff by expressing three key mechanisms to SPLE reuse – configuration, parameterization, and template instantiation – within a type-instance framework. We defined a simple MLM-based kernel-language to show that those mechanisms can be implemented within that framework and can co-exist with other meta-modeling techniques including potency, deep modeling, and power-types (see [8] for an example). This provides a feature-rich, integrated and consistent approach to model-based language engineering. The language used is a much-simplified version of the kernel for the XModeler toolkit [9]. The language is based on a uniform representation for model elements and can support a wide variety of languages that are both general-purpose and domain-specific. We plan to further develop the kernel language and test it in the context of SPLE and to use it as the basis of mixing different MLM approaches with SPLE variability mechanisms.

References

1. Anastasopoules, M. and Gacek, C. (2001). Implementing Product Line Variabilities. Proceedings of the 2001 Symposium on Software Reusability (SSR'01), pp. 109-117
2. Atkinson, C. (1997). Meta-Modeling for Distributed Object Environments. EDOC, pp. 90-101
3. Atkinson, C. and Kühne, T. (2001). The Essence of Multilevel Metamodeling. UML 2001, pp. 19-33
4. Atkinson, C. and Kuhne T. (2003). Model-driven development: A metamodeling foundation. IEEE Software, 20 (5), pp. 36–41
5. Bass, L., Clements, P., and Kazman, R. Software Architecture in Practice. SEI Series in Software Engineering, 3rd Edition, 2012.
6. Becker, J., Delfmann, P., and Knackstedt, R. (2006). Adaptive Reference Modeling: Integrating Configurative and Generic Adaptation Techniques for Information Models. Reference Modeling – Efficient Information Systems Design through Reuse of Information Models, edited by Jörg Becker and Patrick Delfmann, Physica-Verlag HD, pp. 27-58.

7. Brocke, J. (2007). Design Principles for Reference Modelling - Reusing Information Models by Means of Aggregation, Specialisation, Instantiation, and Analogy. Reference Modeling for Business Systems Analysis, edited by P. Fettke and P. Loos, Idea Group Publishing, Hershey, pp. 47-75.
8. Clark, T., Gonzalez-Perez, C., & Henderson-Sellers, B. (2014). A foundation for multi-level modelling. In MULTI 2014–Multi-Level Modelling Workshop Proceedings (p. 43).
9. Clark, T. and Willans, J. (2012). Software language engineering with XMF and Xmodeler. Formal and Practical Aspects of Domain Specific Languages: Recent Developments. IGI Global, USA.
10. Clements, P. and Northrop, L. (2001). Software Product Lines: Practices and Patterns. Addison-Wesley.
11. Cointe, P. (1987). Metaclasses are first class: the objvlisp model. ACM SIGPLAN Notices 22 (12), pp. 156-162.
12. De Lara, J., Guerra, E., and Cuadrado, J. S. (2014). When and How to Use Multi-Level Modelling. ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2), Article no. 12.
13. Frank, U. (2002). Multi-perspective enterprise modeling (MEMO): conceptual framework and modeling languages, Proceedings of the 35th Annual Hawaii International Conference on System Sciences, HICSS), IEEE Computer Society Washington, pp. 72–82
14. Frank, U. (2014). Multilevel Modeling - Toward a New Paradigm of Conceptual Modeling and Information Systems Design. Business & Information Systems Engineering (BISE) 6(6):319-337.
15. Gonzalez-Perez, C. and Henderson-Sellers, B. (2006). A powertype-based metamodelling framework. Software & Systems Modeling 5 (1), pp. 72-90.
16. Henderson-Sellers, B., Clark, T., and Gonzalez-Perez, C. (2013). On the Search for a Level-Agnostic Modelling Language. CAiSE 2013, pp. 240-255.
17. Jacobson, I., Griss, M.L., Jonsson, P. (1997). Software reuse - architecture, process and organization for business. Addison-Wesley-Longman.
18. Jordan, A., Mayer, W., and Stumptner, M. (2014). Multilevel modelling for interoperability. MULTI@MoDELS 2014, pp. 93-102
19. Käkölä, T. (2010). Standards Initiatives for Software Product Line Engineering and Management within the International Organization for Standardization. 43rd Hawaii International Conference on Systems Science (HICSS-43), pp. 1-10.
20. Neumayr, B., Schrefl, M., and Thalheim, B. (2011). Modeling techniques for multi-level abstraction. R. Kaschek, L. Delcambre (Eds.), The Evolution of Conceptual Modeling, pp. 68–92
21. Odell, J. J. (1994). Power types. Journal of Object-Oriented Programming, 2 (2), pp. 8-12
22. Pohl, K., Böckle, G., van der Linden, F. (2005) Software Product-line Engineering: Foundations, Principles, and Techniques, Springer.
23. Reinhartz-Berger, I. and Sturm, A. (2009). Utilizing domain models for application design and validation. Information & Software Technology 51(8), pp. 1275-1289.
24. Rossini, A., de Lara, J., and Guerra, E., Rutle, A., and Wolter, U. (2014). A formalization of deep metamodeling. Formal Aspects of Computing 26 (6), pp. 1115-1152.