Design Decisions for UML and MOF based Domain-specific Language Models: Some Lessons Learned*

Bernhard Hoisl[1,2], Stefan Sobernig[1], Sigrid Schefer-Wenzl[1,2], Mark Strembeck[1,2], and Anne Baumgrass[1,2]

[1] Institute for Information Systems, New Media Lab,
Vienna University of Economics and Business (WU Vienna)
[2] Secure Business Austria Research (SBA Research)
{firstname.lastname}@wu.ac.at

Abstract. In recent years, the development of domain-specific modeling languages (DSMLs) that are based on the MOF and/or UML has become a popular option in the model-driven development context. As a result, the model-driven software engineering community collected many design and implementation experiences. However, most research contributions on this topic do not aim at supporting the DSML development process as a repetitive decision-making process. In this paper, we document some of our experiences gathered from developing ten MOF/UML-based DSMLs and present our experiences in a reusable manner via decision templates. In particular, this paper focuses on design decisions for the initial phase of the DSML development process, i.e. the definition of the DSML's core language model.

Keywords: Domain-specific modeling, Domain-specific languages, Design decisions, UML, Model-driven development

1 Introduction

In model-driven development (MDD), a domain-specific modeling language (DSML) is a specialized modeling language tailored for a particular application domain (e.g., access control, backup policies, or system auditing) (see, e.g., [1,2,3,4]). Thus, a DSML's abstraction level, its expressiveness, and concrete syntax are customized for software developers and for experts in the DSML's application domain. Often DSMLs are developed based on the Unified Modeling Language (UML) [5]. The UML can leverage industry-grade tool support, scientific evaluations of its semantic foundations, and standardized modeling extensions (e.g., SoaML for service-oriented systems [6]). The UML benefits

*This work has partly been funded by the Austrian Research Promotion Agency (FFG) of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) through the Competence Centers for Excellent Technologies (COMET K1) initiative and the FIT-IT program.

from its organizational maintenance through the Object Management Group (OMG) and builds upon a standardized metamodel: the Meta Object Facility (MOF) [7]. With this, the MOF and the UML provide a rich DSML development toolkit.

In recent years, a number of contributions discussed the development of domain-specific languages (DSLs). Examples include empirical research evidence (e.g., case study research [8,9,10]), DSL development processes [1], development guidelines and patterns [2,3,4,11], or selected facets of UML-based DSMLs [12,13]. Despite the availability of such sources of design knowledge, most contributions fall short in one or several respects: Many experiences lack empirically gathered evidence (e.g., an explicitly documented research design). Many are not specifically tailored toward DSMLs in general, or MOF/UML-based DSMLs in particular, but rather toward textual DSLs. Others reflect design knowledge which is specific to a particular toolkit (e.g., the Eclipse Modeling Framework, EMF). Our work complements the experiences mentioned above by providing reusable design knowledge for designing the core language model of MOF/UML-based DSMLs; i.e. specific options, consequences, and dependencies of decisions in this particular phase of DSML development.

The purpose of this paper is to present our experiences, lessons learned, and some of the challenges we faced while developing ten MOF/UML-based DSMLs over the last years. For an overview of these projects see Table 1 (P1–P10). From these experiences, we extracted two decision points with corresponding decision options for the initial DSML development phase of constructing the *core language model*. The core language model captures all relevant domain abstractions and specifies the relations between these abstractions. Accordingly, we defined a core language model for each of our DSMLs. We document the design decisions in a reusable manner by adopting decision templates inspired by related work on documenting architectural design decisions (see, e.g., [3]). The basic phases of DSML development are adopted from [1].

The remainder of the paper is structured as follows: In Section 2, we introduce the process model of DSML development according to [1]. In Section 3, we describe the relations between the decisions and the respective decision options in a structured manner. Limitations of our contribution are discussed in Section 4. Section 5 provides an overview of related work and Section 6 concludes the paper.

2 Background: DSML Development Phases

Before we outline the lessons learned from our DSML projects (see Table 1), we give an overview of the DSML development process applied in our projects (for a detailed discussion see [1]). The following steps were performed iteratively to build the DSMLs:

Define DSML core language model One first defines an initial core language model and the corresponding language model constraints for the target domain. By following a domain analysis method, such as domain-driven design

#	Objectives	Domain
P1	An approach to model interdependent concern behavior using extended UML activity models [14].	Separation of concerns
P2	An integrated approach for modeling processes and process-related RBAC models (roles, hierarchies, statically and dynamically mutual exclusive tasks etc.) [15].	Business processes, role-based access control (RBAC)
P3	A UML extension for an integrated modeling of business processes and process-related duties; particularly the modeling of duties and associated tasks in business process models [16].	Business processes, process-related duties
P4	An approach to provide modeling support for the delegation of roles, tasks, and duties in the context of process-related RBAC models [17].	Business processes, delegation of roles, tasks, and duties
P5	A UML extension to model confidentiality and integrity of object flows in activity models [18].	Data confidentiality and integrity
P6	UML modeling support for the notion of mutual exclusion and binding constraints for duties in process-related RBAC models [19].	RBAC (consistency checks for duties)
P7	Incorporation of data integrity and confidentiality into the model-driven development of process-driven service-oriented architectures [20].	Integrity and confidentiality for service invocations
P8	Integration of context constraints with process-related RBAC models and thereby supporting context-dependent task execution [21].	Business processes, RBAC, context constraints
P9	A generic UML extension for the definition of audit requirements and specification of audit rules at the modeling-level [22].	Audit rules
P10	An approach based on model transformations between the valid structural and behavioral runtime states that a system can have [23].	Model transformation

Table 1. Overview of conducted DSML development projects.

(see, e.g., [24]), domain abstractions are identified and form the language model of a DSML. Because the language model often cannot capture all restrictions and/or semantic properties of the DSML elements, language model constraints are added, if necessary. This phase results in the *DSML core language model* and a catalog of *DSML language model constraints*.

Define DSML concrete syntax In this phase, graphical or textual notation symbols as well as composition and production rules are defined. The DSML core language model and the DSML language model constraints serve as input to produce the *DSML concrete syntax specification*.

Define DSML behavior The behavior specification of a DSML determines how the DSML elements interact to produce the behavior intended by the DSML designer. Syntax and behavior of a DSML are usually defined in parallel. The *DSML behavior specification* (e.g., control flow models, formal textual specifications) is the output of this phase.

DSML platform integration All artifacts defined for a DSML are mapped to the features of a selected platform, either by extending an existing platform or by developing a new tool set. Platform integration is achieved by defining model transformations (see, e.g., [25]) to convert a model into another platform-specific model (model-to-model transformation, M2M) or into machine-readable software artifacts (model-to-text transformation, M2T).

# Decision/Option	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
D1 Language model formalization										
O1.1 M1 class model										
O1.2 Profile definition	×		×				×		×	×
O1.3 Metamodel extension	×	×	×	×	×	×	×	×	×	×
O1.4 Metamodel modification										
O1.5 Combination of options[1]	⊂		⊂				≍		⊂	⊂
D2 Language model constraints										
O2.1 Explicit constraint expressions	×	×	×	×	×	×	×	×	×	×
O2.2 Code annotations										
O2.3 Constraining M2T transformations							×		×	
O2.4 Textual annotations		×			×			×		×
O2.5 Combination of options[1]		≍			⊂		≍	⊂	⊂	⊂
O2.6 None										

Table 2. Overview of design decision points and options.

3 Collected Decisions on the Core Language Model

Most of our DSMLs (see Table 1) provide modeling support for different types of security aspects in a business process context. P10 [23] is an exception and aims at describing program transformations in dynamic programming environments. Each of the ten DSML projects adopted the development process sketched in Section 2. However, due to differing requirements, we did not always perform all DSML development phases. For example, we do not provide platform integration for P10. Thus, to document our experiences from the ten projects, we focus on a single phase and on two specific decisions that appeared in each of the ten projects. In particular, this paper reports on the core language model definition and on the respective design decisions.

For each case presented in Table 1, we identified different decision options. The development of a DSML core language model requires two important decisions: DSML language model formalization (Section 3.1) and defining language model constraints (Section 3.2). Table 2 summarizes these options for both design decision points and lists the options adopted for each of the ten cases. Fig. 1 depicts an overview of the two decisions, the corresponding options, as well as the interdependencies between the decisions and their options. These relations are then discussed for each decision in the respective *Consequences* sub-section (see below).

3.1 D1 Language Model Formalization

Decision *In which way should the domain concepts be formalized?*
Context Domain abstractions are identified and form the language model of a DSML (i.e., the abstract syntax). This language model definition can be expressed, for instance, in a narrative text form, with mathematical expressions (e.g., set algebra), or via a modeling language (e.g., the UML). The language model definition serves as input for the phase of formalizing the domain constructs into the core language model expressed via the UML.

[1] ⊂ options complementary; ≍ options equivalent

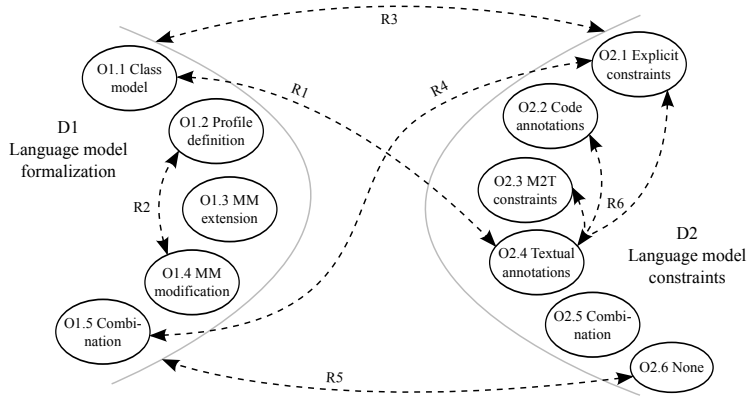

Fig. 1. Relations (R1–R6) between decision options.

Options For UML-based DSMLs, the language model can be defined within the boundaries of the modeling language via dedicated language extension constructs (such as UML profiles) or by extending the modeling language to provide the required semantics (see, e.g., [5,26]).

O1.1 M1 class model: UML class models are an ad-hoc instrument to formalize domain abstractions. Domain concepts can be expressed as classes and relationships as associations.

O1.2 Profile definition: Profiles are a language extension option to tailor the UML for different purposes. A profile consists of a set of stereotypes which define how an *existing* UML metaclass may be extended.

O1.3 Metamodel extension: A metamodel extension introduces, for instance, new metaclasses and/or new associations between metaclasses (MOF-based extension [7]).

O1.4 Metamodel modification: In contrast to a metamodel extension, existing metaclasses of the UML metamodel are modified; e.g., by changing the type of a class property or by deleting existing associations (MOF-based extension [7]).

O1.5 Combination of options: A combination may include the definition of a metamodel extension as well as an equivalent profile definition (e.g., P7). Similarly, stereotype definitions can be provided to accompany a metamodel modification (e.g., P9).

Drivers

Domain space: The degree of overlap between the domain space of the DSML concepts and the general purpose language constructs (i.e., the UML specification) has a direct impact on whether a profile definition is sufficient or on whether a metamodel extension/modification is needed (O1.2–O1.4). In general, a UML extension is reusable if it is compliant with the UML standard.

DSML expressiveness: For instance, a UML profile (O1.2) can only specialize the UML metamodel in such a way that the profile semantics do not conflict with the semantics of the referenced metamodel. Therefore, profile constraints may

only define well-formed rules that are more constraining (but consistent with) those specified by the metamodel [5]. In contrast, a metamodel extension/modification (O1.3 and O1.4) is only limited by the constraints imposed by the MOF metamodel.

Portability and evolution: A metamodel extension/modification (O1.3 and O1.4) creates a fork of a certain version of the UML specification. The metamodel does not inherit revisions coming from newly released OMG specifications and can deviate from the UML or MOF standard.

DSML integration: Available DSMLs, software systems, and tool support have a direct impact on the design process of a DSML in terms of integration possibilities. For instance, the UML specification defines a standardized way to use icons and display options for profiles (O1.2). Tool support for authoring UML class models and profiles (O1.1 and O1.2) is widely available.

Consequences (see Fig. 1)

R1 Constraint limitations for class models: A class model defines a language model at the UML instance level (i.e. at the M1 level, see [7]). This means, no metamodel is defined to reflect the domain space and, thus, domain concepts can neither be instantiated nor explicitly constrained for their usage as modeling constructs. Thus, restrictions can only be defined in terms of text annotations attached to the language model.

R2 Profile dependency: Dependencies can occur from combined language model formalizations. For instance, profiles are dependent on the UML metamodel. If a profile is combined with a metamodel modification, changes to the metamodel can lead to implicit and unwanted changes affecting the defined stereotypes (e.g., if a stereotype-extended metaclass is modified).

Examples In all DSML projects, we formalized the language models as metamodel extensions (O1.3). Additionally, profiles (O1.2) were employed in P1, P3, P7, P9, and P10. Therefore, we effectively adopted combined strategies (O1.5). In order to be compliant with the OMG specifications, we did not consider modifying the UML metamodel (O1.4). As an example, Fig. 2 depicts an excerpt from a UML extension (taken from P7). On the left hand side, it shows a UML package definition called `SecureObjectFlows::Services` as an example of a metamodel extension, on the right hand side, it shows a UML profile specification named `SOF::Services`. Mappings between these two language-model representations are provided as M2M transformations. Both UML customizations provide the same modeling capabilities for using one of our UML security extensions (for details see [18,20]) with the SoaML specification [6].

3.2 D2 Language Model Constraints

Decision *Do we have to define constraints over the core language model(s)? If so, how should these constraints be expressed?*

Context A core language model has been formalized in the UML, using either a UML metamodel extension/modification, a UML profile, or a UML class model (see Section 3.1). The resulting language model describes the domain-specific language in terms of its language elements and their interrelations. The definition

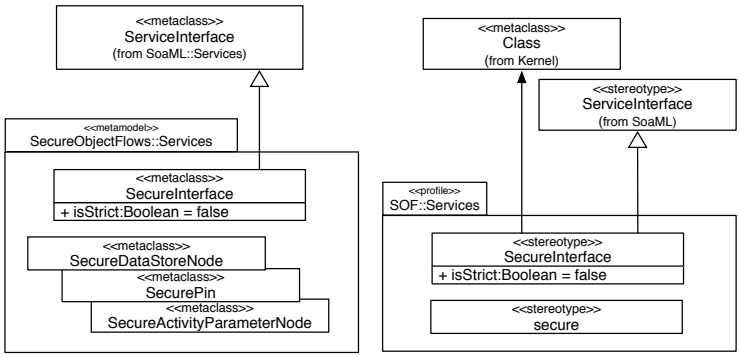

Fig. 2. Exemplary UML metamodel extension and profile definition [20].

of these interrelations is limited through the expressiveness of the MOF and the UML (e.g., part-of relations). A structural UML model, however, cannot capture certain categories of constraints over domain concepts that are relevant for the description of the target domain. Examples are invariants for domain concepts, pre-conditions and post-conditions, as well as guards (referred to as *static* constraints, hereafter). As a result, the language model formalization could be incomplete or ambiguous.

If the language model has been realized by creating multiple formalizations (e.g., multiple profiles), there is an additional risk of introducing inconsistencies provided that the DSML can be used in different configurations (e.g., different profile compositions). Consider, for example, profiles which provide a bridge between two UML extensions.

Options

O2.1 Constraint-language expressions: One can make language model constraints explicit using a constraint-expression language, for instance, via the Object Constraint Language (OCL) or via the Epsilon Validation Language (EVL) in Eclipse.

O2.2 Code annotations: The language model and its elements are enriched through annotations which contain expressions in the host language (or a language embedded within the host language). For example, this can be realized by using model annotations and UML's `OpaqueExpression` [5].

O2.3 Constraining M2T transformations: The constraints over the language model are expressed at the level of transformation templates. That is, template expressions contain checks (e.g., conditional statements based on model navigation expressions) which test model instances for the implicit fit with corresponding domain constraints; e.g., conditional Epsilon Transformation Language (ETL) statements based on Epsilon Object Language (EOL) expressions.

O2.4 Textual annotations: Certain constraints (e.g., temporal bindings) elicited from the target domain cannot be captured sufficiently via evaluable expressions (i.e., constraint language expressions, code annotations) and/or the constraints serve a documentary purpose (to the domain expert). In such cases,

unstructured text annotations may capture constraint descriptions meant for the human reader only (e.g., via UML comments).

O2.5 Combination of options: For instance, textual annotations are used as an addition to constraint-language expressions.

O2.6 None: Static constraints over the language model are not made explicit in (or along with) the language model.

Drivers

Constraint formalization: In early iterations (e.g., DSML prototyping), constraints might not be expressed via well-formed, syntactically valid constraint-language expressions, but rather as pseudo-expressions or unstructured text. With the language model maturing during subsequent iterations these annotations can be transformed into evaluable expressions.

Automated language model checking: Depending on whether tool integration for model checking is a requirement, the options O2.1–O2.3 are candidates. A driver toward either option is the intended model-checking time. Relevant points in time follow from the model formalization option adopted (e.g., class model vs. metamodel-based) and the platform-support (model-level or instance-level checks). Language-model checking based on template expressions (O2.3) realizes the latest possible checking point. Therefore, this option does not offer any constraint-based feedback during model development.

Native language model constraints: Constraint-language expressions are developed with the purpose of integrating (i.e., navigating and checking) with the (meta-)model representations. Examples are standard-compliant and vendor-specific OCL expressions for the UML, as well as EVL expressions and Java-coded constraints over secondary Ecore representations of UML models (Eclipse EValidator framework).

Maintainability: Explicitly stating model constraints (O2.1 through O2.3) creates structured text artifacts which must be maintained along with the model artifacts (e.g., the XMI representation). Toolkits and their model representations offer different strategies for this purpose, for instance, embedding constraints into model elements (i.e., model annotations, such as UML comments), maintaining constraint collections as external resources (e.g., separate text files), or editor integration. Each strategy affects the artifact complexity and the effort needed to keep the constraints and the models synchronized.

Portability: If the portability of constraints between different MDD toolkits (e.g., Eclipse MDT, Rational Software Architect, MagicDraw, Dresden OCL) is a mandatory requirement, the platform-dependent options O2.2 and O2.3 can be excluded. However, due to the version incompatibilities and the different vendor-specific constraint-language dialects (e.g., Eclipse MDT OCL), even O2.1 does not guarantee portability for the underspecified sections of the OCL/UML specifications (e.g., navigating stereotypes in model instances or for transitive quantifiers such as `closure` [27]).

Consequences (see Fig. 1)

R3 Conformance between language model and constraints: Constraints on the language model can be defined separately from the referencing metamodel (e.g.,

using code annotations; O2.2) or at a later stage (e.g., for M2T transformations; O2.3). It must be ensured that language model constraints do not contradict their language model formalization and vice versa. Moreover, constraints may need to be adapted when the corresponding metamodel changes (e.g., OCL navigation expressions).

R4 Constraint inconsistencies: A combination of different language model formalizations (e.g., a UML profile and a metamodel extension; O1.5) may require the duplication and modification of explicit constraint definitions.

R5 Unambiguous language model: If no further constraints to the language model are specified, the language model must be fully and unambiguously defined using the chosen formalization option and their implicitly enforced restrictions (e.g., by using profiles and, thus, inheriting all semantics from the UML metamodel; O1.2).

R6 Impossible constraint evaluation: Some constraints cannot be captured by the means of constraint languages and the underlying language models, code annotations, or model transformation templates (see, e.g., [5]; O2.1–O2.3). Such constraints have to be provided as text annotations in a natural language (O2.4). These constraints either have a documentation purpose only, or they serve for porting the constraints to another environment as they are not bound to a concrete expression form.

Examples In our DSMLs, we encountered all options but code annotations (O2.2) and entirely unconstrained language models (O2.6). So far, we provide constraint-language expressions (O2.1) in the OCL for all of our cases. This is because precise execution semantics were to be expressed in terms of the foundations of UML activities (token flows, e.g., in P1) and of the UML state machines (state/transition; in P10). In eight out of ten DSMLs (P2–P9), these semantics are described by a generic and MOF-compliant metamodel, as well as corresponding metamodel extensions. The generic constraints were then mapped to a UML-based language formalization (i.e. the actual language model and the respective OCL expressions). Code annotations (O2.2) were not considered because the additional model constraints should not be specific to a particular platform (e.g., model representation APIs, generator language). For two DSMLs (P7, P9), we additionally incorporated constraining M2T transformations (O2.3). Textual annotations (O2.4) are either used to complement OCL constraints (P5, P8, P10) or as full substitutes (P2) for otherwise formally expressed constraints.

Constraint 1: The operands specified in a `ContextCondition` are either `ContextAttributes` or `ConstantValues`.

```
context ContextCondition inv:
  self.expression.operand.oclAsType(OperandType)->forAll(o |
    o.oclIsKindOf(ContextAttribute) or
    o.oclIsKindOf(ConstantValue))
```

Constraint 5: The $\text{fulfilled}_{\text{CD}}$ Operations must evaluate to true to fulfill the corresponding ContextCondition.

As an example for these two different purposes, consider the above excerpt from P8: For an activity, each action can be guarded by a constraint whose conditions refer to a set of operands and checking operations. At the instance-level (M0), the operations are called to evaluate whether an action should be entered, depending upon some contextual state. Constraint 1 shows a complementary textual annotation. Constraint 5 exemplifies a constraint expressed in natural language due to a model-level mismatch: While the constraint is captured at the language-model level (M2), the operation calls (whose boolean return values are combined to yield the runtime evaluation of the guard) become manifest at the occurrence level of an activity instance (M0) only.

4 Limitations

The most important limitations of the work presented in this paper are that 1) our lessons learned result only from a collective experience and that 2) the underlying decisions were taken by the same group of researchers who developed the ten DSMLs. We reported decisions being characteristic for a single phase (i.e. defining the DSML core language model) and their interdependencies. Documenting the remaining phases (see Section 2) is future work. Moreover, there is the risk of a technology bias given that the ten DSML projects were all performed in a specific technology context (e.g., MOF/UML, OCL, Eclipse modeling tools).

Methodically, this paper presents the results of a narrative synthesis [28] of our DSML development experiences. Therefore, by emphasizing a preselected process model and one of its phases [1], we may have neglected design decisions beyond the scope of this approach. Other risks are the disagreement among the authors during the synthesis process and the dependence of the synthesis results on the review performance of each author (time constraints, level of experience). To mitigate these, we conducted multiple refining iterations over the decision templates and the decision relations, under shifting roles of data checker and data extractor.

5 Related Work

Related work on DSL development [1,2,3,4,8,9,10,11,12,13] was already outlined in Section 1. Below, we review the work relevant for our methodical approach.

For reflecting and synthesizing the decision-related findings from our DSML-development projects, we adapted the guidelines on conducting narrative syntheses proposed by [28]. That is, we selected a process model and its phases as the implicit "theory" underlying our DSML projects. We then collected meta-data about the primary works (e.g., participants, setting, outcomes, target domain, MDD technologies). Based on the selected "theory" (i.e., phases and development artifacts), we then characterized the decisions taken in each development project. In particular, we adopted previously defined decision templates.

The practice of documenting design decisions in a template-based or model-based manner has been proposed for architectural design decisions (see, e.g.,

[29]). In our work, we share the primary motivation of documenting *reusable* design decisions, i.e., decisions and options which are characteristic for every decision-making process in a given technical domain.

6 Concluding Remarks

In this paper, we presented lessons learned from ten DSML development projects in the form of a narrative synthesis. We documented MOF/UML-based decision options and relations between them for the phase of defining the core language model for a DSML in a structured and reusable form. By doing so, we provide decision support for future decision-making processes, facilitate decision documentation, and offer scaffolding for making decisions under incomplete or changing requirements (i.e., in early stages of developing or prototyping). Although we especially focus on design decisions for MOF/UML-based DSMLs, certain decision options do also apply to other modeling languages used in MDD processes. In our future work, we will document additional decision points to cover the remaining phases of the DSML development process.

References

1. Strembeck, M., Zdun, U.: An Approach for the Systematic Development of Domain-Specific Languages. Software: Practice and Experience (SP&E) **39**(15) (2009) 1253–1292
2. Mernik, M., Heering, J., Sloane, A.: When and How to Develop Domain-specific Languages. ACM Computing Surveys (CSUR) **37**(4) (2005) 316–344
3. Zdun, U., Strembeck, M.: Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Projects. In: Proc. of the 14th European Conference on Pattern Languages of Programs (EuroPLoP). (2009)
4. Spinellis, D.: Notable Design Patterns for Domain-specific Languages. Journal of Systems and Software **56**(1) (2001) 91–99
5. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure – Version 2.4.1. Available at: `http://www.omg.org/spec/UML` (2011)
6. Object Management Group: Service oriented architecture Modeling Language (SoaML) – Version 1.0. Available at: `http://www.omg.org/spec/SoaML` (2012)
7. Object Management Group: OMG Meta Object Facility (MOF) Core Specification – Version 2.4.1. Available at: `http://www.omg.org/spec/MOF` (2011)
8. Zdun, U.: A DSL Toolkit for Deferring Architectural Decisions in DSL-based Software Design. Information and Software Technology **52**(9) (2010) 733–748
9. Wile, D.: Lessons Learned from Real DSL Experiments. Science of Computer Programming **51**(3) (2003) 265–290
10. Kelly, S., Pohjonen, R.: Worst Practices for Domain-Specific Modeling. IEEE Software **26**(4) (2009) 22–29
11. Karsai, G., Krahn, H., Pinkernell, C. et al.: Design Guidelines for Domain Specific Languages. In: Proc. of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM). (2009)
12. Selic, B.: A Systematic Approach to Domain-Specific Language Design Using UML. In: Proc. of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), IEEE (2007)

13. Robert, S., Gérard, S., Terrier, F. et al.: A Lightweight Approach for Domain-Specific Modeling Languages Design. In: Proc. of the 35th Euromicro Conference on Software Engineering and Advanced Applications, IEEE (2009)
14. Strembeck, M., Zdun, U.: Modeling Interdependent Concern Behavior using Extended Activity Models. Journal of Object Technology **7**(6) (2008) 143–166
15. Strembeck, M., Mendling, J.: Modeling Process-related RBAC Models with Extended UML Activity Models. Information and Software Technology **53**(5) (2010)
16. Schefer, S., Strembeck, M.: Modeling Process-Related Duties with Extended UML Activity and Interaction Diagrams. In: Proc. of the International Workshop on Flexible Workflows in Distributed Systems. (2011)
17. Schefer, S., Strembeck, M.: Modeling Support for Delegating Roles, Tasks, and Duties in a Process-Related RBAC Context. In: Proc. of the International Workshop on Information Systems Security Engineering (WISSE), Springer, LNBIP (2011)
18. Hoisl, B., Strembeck, M.: Modeling Support for Confidentiality and Integrity of Object Flows in Activity Models. In: Proc. of the 14th International Conference on Business Information Systems (BIS), Springer, LNBIP (2011)
19. Schefer, S.: Consistency Checks for Duties in Extended UML2 Activity Models. In: Proc. of the International Workshop on Security Aspects of Process-aware Information Systems (SAPAIS), IEEE (2011)
20. Hoisl, B., Sobernig, S.: Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models. In: Proc. of the International Workshop on Security Aspects of Process-aware Information Systems (SAPAIS), IEEE (2011)
21. Schefer-Wenzl, S., Strembeck, M.: Modeling Context-Aware RBAC Models for Business Processes in Ubiquitous Computing Environments. In: Proc. of the 3rd International Conference on Mobile, Ubiquitous and Intelligent Computing. (2012)
22. Hoisl, B., Strembeck, M.: A UML Extension for the Model-driven Specification of Audit Rules. In: Proc. of the 2nd International Workshop on Information Systems Security Engineering (WISSE'12), Springer, LNBIP (2012)
23. Zdun, U., Strembeck, M.: Modeling Composition in Dynamic Programming Environments with Model Transformations. In: Proc. of the 5th International Symposium on Software Composition, LNCS, Vol. 4089, Springer (2006)
24. Evans, E.: Domain-driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2004)
25. Mens, T., Gorp, P.v.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science **152** (2006) 125–142
26. Bruck, J., Hussey, K.: Customizing UML: Which Technique is Right for You? Available at: `http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html` (2008)
27. Object Management Group: OMG Object Constraint Language (OCL) – Version 2.3.1. Available at: `http://www.omg.org/spec/OCL` (2012)
28. Cruzes, D., Dybå, T.: Synthesizing Evidence in Software Engineering Research. In: Proc. of the International Symposium on Empirical Software Engineering and Measurement (ESEM). ACM (2010)
29. Obbink, H., Kruchten, P., Kozaczynski, W. et al.: Software Architecture Review and Assessment (SARA) Report, Version 1.0. Available at: `http://kruchten.com/philippe/architecture/SARAv1.pdf` (2002)