A Catalog of Reusable Design Decisions for Developing UML/MOF-based Domain-specific Modeling Languages*

Bernhard Hoisl[1], Stefan Sobernig[1], and Mark Strembeck[1,2]

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

Abstract

In model-driven development (MDD), domain-specific modeling languages (DSMLs) act as a communication vehicle for aligning the requirements of domain experts with the needs of software engineers. With the rise of the UML as a de facto standard, UML/MOF-based DSMLs are now widely used for MDD. This paper documents design decisions collected from 90 UML/MOF-based DSML projects. These *recurring* design decisions were gained, on the one hand, by performing a systematic literature review (SLR) on the development of UML/MOF-based DSMLs. Via the SLR, we retrieved 80 related DSML projects for review. On the other hand, we collected decisions from developing ten DSML projects by ourselves. The design decisions are presented in the form of reusable *decision records*, with each decision record corresponding to a decision point in DSML development processes. Furthermore, we also report on frequently observed (combinations of) decision options as well as on associations between options which may occur within a single decision point or between two decision points. This collection of decision-record documents targets decision makers in DSML development (e.g., DSML engineers, software architects, domain experts).

1 Introduction

This report presents a collection of reusable design-decision descriptions (decision records) and auxiliary materials (prototype solutions, rationale tables) gained from systematically reviewing 90 domain-specific modeling language (DSML) projects. The variety of these 90 projects allowed us to gain significant insights into the design of DSMLs (e.g., knowledge about frequently observed decision options and combinations of options in and across existing DSML designs). The audience of this document collection are decision makers in DSML

*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.

development processes (e.g., DSML engineers, software architects, domain experts). We consider this report as a valuable source for the decision-making process of design-decision makers by providing, e.g., guidance on when to favor or to discard certain candidate solutions for a DSML design problem. The reviewed DSMLs are based on the Unified Modeling Language 2 (UML; [116]) and the corresponding Meta Object Facility (MOF [115]). As a consequence, the documented design rationale is to some extent specific to the capabilities of the UML 2.x, in general, and the UML 2.x extension techniques, in particular.

This documentation of generic design-rationale documentation has been compiled in several steps, in a long-running research project:

1. Initial design reviews of 10 of our own DSML projects [70]; see also Appendix B.

2. A pilot literature-review study [51].

3. An authoritative, large-scale systematic literature review (SLR) of software-engineering publications published between 2005 and 2012 [141] yielding 80 DSML projects; see also Appendix C).

In step 1, we reviewed ten (out of the final 90) DSML projects which had been developed by ourselves and are summarized in Table 1. The first column of Table 1 states the consecutive DSML project numbering used throughout this paper, the DSML's name, and a reference to the corresponding publication(s). The application domains shown in the third column of Table 1 are encoded according to the 2012 ACM Computing Classification System (CCS).[1]

Projects P2–P9 provide support for modeling various security properties of software-based information systems, such as, role-based access control (RBAC), process-related duties, or data confidentiality and integrity. The DSMLs resulting from P2–P9 are based on a common and generic metamodel defined in P2. The other two DSMLs support the modeling of interdependent concern behavior (P1) and the modeling of composition in dynamic programming environments (P10).

In step 3), we performed a systematic literature review (SLR; see, e.g., [26, 87, 164]) on the development of UML/MOF-based DSMLs (more information is available at [141]). With the SLR, we were able to retrieve 80 related DSML projects and to complement and revise our decision catalog. The revisions to the initial version of the catalog [70, 71] are documented in Appendix A. The SLR helped us to collect evidence for validating the decision options and associations we identified (see Table 3).

To map the domain coverage of the 90 projects, we classified every DSML according to the CCS. Table 15 shows the frequency of categories assigned to the selected DSML projects (due to its size, the table was moved to Appendix D at the end of this paper). In total, we used 63 distinct CCS categories and we assigned 177 category tags, that is a mean of $\sim$2 category assignments per DSML project. The frequency distribution shows that the paper corpus covers a very broad and diverse range of application domains. The most frequently assigned CCS categories see an important number of DSMLs falling into the areas of service-oriented architectures (especially implemented via web services), software security engineering, as well as business process modeling.

[1]Available at `http://www.acm.org/about/class`; last accessed: Feb 2, 2015.

Table 1: Overview of conducted DSML development projects. See also Appendix B.

DSML	Objective	Domains
P1 ConcernActivities [151]	An approach to model interdependent concern behavior using extended UML activity models.	Access control, Software design engineering
P2 BusinessActivities [150]	An integrated approach for modeling processes and process-related RBAC models (roles, hierarchies, statically and dynamically mutual exclusive tasks etc.).	Access control, Business process modeling, Software security engineering
P3 UML-PD [137, 138]	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.	Access control, Business process modeling, Software security engineering
P4 UML-DEL [136, 138]	An approach to provide modeling support for the delegation of roles, tasks, and duties in the context of process-related RBAC models.	Access control, Business process modeling, Software security engineering
P5 SOF [68]	A UML extension to model confidentiality and integrity of object flows in activity models.	Business process modeling, Software security engineering
P6 UML-PD [135]	UML modeling support for the notion of mutual exclusion and binding constraints for duties in process-related RBAC models.	Access control, Business process modeling, Software security engineering
P7 SOFServices [67, 72]	Incorporation of data integrity and confidentiality into the MDD of process-driven SOAs.	Business process modeling, Service-oriented architectures, Software security engineering, Web services
P8 UML-CC [139]	Integration of context constraints with process-related RBAC models and thereby supporting context-dependent task execution.	Access control, Business process modeling, Software security engineering
P9 SecurityAudit [69]	A generic UML extension for the definition of audit requirements and specification of audit rules at the modeling-level.	Publish-subscribe / event-based architectures, Software security engineering
P10 MTD [161]	An approach based on model transformations between the valid structural and behavioral runtime states that a system can have.	Object oriented languages, Software architectures

Additionally, we extracted the UML diagram types tailored by the 90 DSMLs according to Annex A of the UML superstructure [116] (see Table 2). In total, the DSMLs tailor 156 diagram types, that is a mean of ~1.8 diagram types per DSML project. In this calculation, we omit four DSMLs, which target all UML diagram types or are unspecific about the diagram types. Regarding DSMLs' primary modeling instruments, ~65% (101/156) of the tailored diagram types define structure and ~35% (55/156) behavior of a software system. In terms of modeling a system's structure, class diagrams are adopted by 55 DSMLs, followed by component and package diagrams (15 and 14 DSMLs). On the behavioral side, activity (27), state machine (13), and use case diagrams (9) are the three most frequently used ones.

To structure this document collection, we adopted the notion of a tai-

[2]The DSML does not tailor a UML diagram type specifically; for example, a stereotype extension of a UML element applicable in all diagram types, such as, `Element` (see, e.g., [27, 69]) or `Constraint` (see, e.g., [37]).

Table 2: Frequency of DSML-tailored UML diagram types.

Diagram type	Diagram-type kinds	Frequency
Class	Structure	55
Activity	Behavior	27
Component	Structure	15
Package	Structure	14
StateMachine	Behavior	13
UseCase	Behavior	9
CompositeStructure	Structure	8
Object	Structure	6
Sequence	Behavior	5
*[2]	—	4
Deployment	Structure	3
InteractionOverview	Behavior	1

lorable DSML development procedure from [152]. In particular, this procudural model includes the following main tasks: 1) core-language-model definition, 2) concrete-syntax definition, 3) behavior specification, and 4) platform integration (for details see Section 2.1). If these tasks are performed in sequence, an instance of this procedure will result in a *language-model-driven* development approach (steps 2 and 3 are performed in parallel; see also [162]).

The remainder of the paper is structured as follows. First, we elaborate on our choices of representing design rationale for DSMLs (Section 2), including the identified decision points (Section 2.1), a template for decision records (Section 2.2), an excerpt from encoded design decisions and options (Section 2.3), as well as the notational conventions used throughout this document (Section 2.4). Next, Section 3 explains seven prototype option-sets and sketches nine frequently adopted decision options found at corresponding decision points. Subsequently, the complete catalog of collected decision records for designing UML/MOF-based DSMLs is presented in Section 4 (Sections 4.1–4.6 correspond to the identified decision points in Section 2.1). Associations between options of one decision point and between options of different decision points are discussed in Section 5. Following the bibliography, revisions to the initial version of the catalog [70, 71] are explained in Appendix A. The complete list of encoded design decisions for each of the 90 DSMLs is provided in Appendix B. In addition, Appendix C enumerates the application domain(s), the tailored diagram type(s), and the option sets per DSML project in an overview table. At last, the frequency of application domains is reported in Appendix D.

2 Representing Design Rationale

Design rationale ([29, 46]) on DSML development is the reasoning and justification of decisions made when designing, creating, and using the core artifacts of a DSML (e.g., abstract and concrete syntax, behavior specification, metamodeling infrastructure, MDD toolchain integration). Documenting design rationale explicitly aims at helping design-decision makers by providing and explaining past decisions (e.g., in a design-space analysis) and by improving the understanding of a DSML design during development and maintenance (e.g., as a kind of design-process documentation).

2.1 Decision Points

The process of developing a DSML can be divided into 6 different decision points which can be conducted in different sequences depending on the development style used and the intention behind developing the DSML [152]. In general, each decision point corresponds to a decision record in this document (D1–D6 below), with each decision record grouping a number of design-decision options.

In a large-scale empirical investigation [141], most DSMLs were found to involve decisions related to 3 of the 6 phases: D1, D2, D4 (highlighted using rectangle boxes below). Besides, depending on the context (e.g., application domain, usage intention, and development style), some decision points may also be skipped.

D1 Language-model definition (Section 4.1). After a systematic analysis and a structuring of the respective language domain, one identifies the domain abstractions to be represented by a DSML. In this context, one of the main questions is how one describes these domain abstractions to arrive at a comprehensive and comprehensible language model which can be used as a basis for developing the DSML. Corresponding options are the description via a (formal) textual description, via formal or informal (graphical) models, or through a combination of these options.

D2 Language-model formalization (Section 4.2). At this decision point, it is determined how a language model defined informally or defined independently from the UML (see D1) is turned into a *formal UML model*. By formal model, we refer to a realization of the language model using a well-defined metamodeling language such as the UML/MOF metamodeling infrastructure. A metamodeling language is itself based on a well-defined and well-documented language model (i.e., CMOF for the UML metamodel [115]) and provides at least one well-defined and well-documented concrete syntax to define an own language model (e.g., the CMOF diagram syntax to specify a UML metamodel extension). At this decision point, decision options are the definition of an M1 structural model, a UML profile, a UML metamodel extension, a UML metamodel modification, or a combination of these options.

D3 Language-model constraints (Section 4.3). A structural UML model cannot (or only insufficiently) capture certain categories of constraints on domain abstractions, such as invariants for domain abstractions, pre- and post-conditions, as well as guards. As a result, the language-model formalization could be incomplete or ambiguous. To prevent this, one can specify special-purpose language-model constraints, for instance via a constraint-language (such as the OCL [118]), code/textual annotations, model-to-model/model-to-text transformations (M2M/M2T), or a combination of these options.

D4 Concrete-syntax definition (Section 4.4). The concrete syntax of a UML/MOF-based DSML serves as its user interface and can be defined in several ways. One can either use model annotations, reuse or extend a diagrammatic syntax, mix foreign syntaxes with the UML syntax, extend a UML/MOF-based frontend syntax, provide an alternative syntax, or apply a combination of some of these options.

D5 Behavior specification (Section 4.5). The behavior specification of a DSML defines behaviors specific to one or more DSML language element(s). It determines how the language elements of the DSML interact to produce behavior as intended by the DSML engineer. Behavior can be specified via M1 behavior models, formal or informal textual specifications, constraining model executions, or a combination thereof.

D6 Platform integration (Section 4.6). In order to produce platform-specific, executable models (e.g., source code) from DSML models, all DSML artifacts need to be mapped to a software platform. Corresponding decision options at this point are the generation of intermediate models, using code generation templates, employing API-based generators, the direct execution of models, performing M2M transformations, or applying a combination of these options.

2.2 A Template for Decision Records

For structuring and presenting the recurring DSML design decisions, we employ a document template which lays out predefined sections. This template has been derived from prior work on documenting design rationale in software engineering, in particular architectural design decisions [70]. The decision-record template provides a space of solutions to a given DSML design problem. Each decision-record document contains the following sections (see Figure 1):

1. *Problem statement:* Describes the problem that has been repeatedly observed for several DSML design projects, in a specific decision context (see below).

2. *Decision context:* Each decision record captures problem and solution statements specific to a decision context (e.g., using certain metamodeling toolkit, the application domain modeled by a DSML, or a certain decision-making phase [152]).

3. *Decision options:* For each decision problem, several candidate solutions are listed (e.g., formalizing the language model using a UML profile or a MOF metamodel). The options listed by each decision record have been extracted from the selected DSML projects as primary sources (see Table 13) and/or secondary studies on designing MOF/UML-based DSMLs.

4. *Decision drivers:* Describe forces which steer the DSML engineer towards a particular option (e.g. whether the DSML must extend the UML metamodel).

5. *Decision consequences:* The selection of an option (or, a combination of options) affects the solution spaces of subsequent decisions (e.g. another

decision context and the follow-up decisions). DSML designers must be aware of such consequences for an informed decision making in subsequent decision steps (e.g., to avoid conflicting language-model constraint definitions).

6. *Application:* This section documents how different design options were applied in actual DSML projects. The main goal is to document the successful application of corresponding options and to provide references for further investigation by the design-decision maker.

7. *Sketch:* Finally, each decision record gives a concrete example of applying one of the options. This excerpt, while being limited to one option, is meant to improve the comprehensibility of the previously described options which are presented in an abstracted manner.

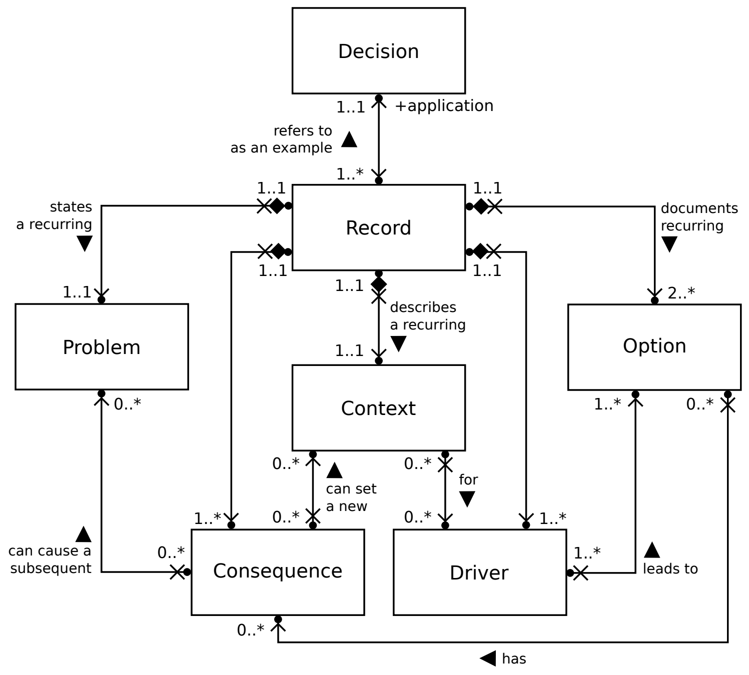


Figure 1: Conceptual overview of key concepts: decision records, decision context, decision problem, decision options, decision drivers, and decision consequences.

2.3 Excerpt from Encoded Design Decisions and Options

Table 3 shows an overview of the identified DSML design decisions and their corresponding options. Projects P1–P10 were performed by us (see also Table 1) while the remaining projects were collected via the SLR. Table 3 shows only an excerpt from the DSML development projects found during the SLR for the purpose of demonstrating different combinations of options (as referenced in the decision records presented in Section 4). Because of its size the complete table was moved to Appendix B at the end of this paper.

Table 3: Overview of encoded design decision points and corresponding options (excerpt).

Decision/Option		P1 [151]	P2 [150]	P3 [137, 138]	P4 [136, 138]	P5 [68]	P6 [135]	P7 [67, 72]	P8 [139]	P9 [69]	P10 [161]	P17 [4]	P30 [16]	P39 [93]	P53 [143]	P58 [144]	P60 [102]	P61 [42]
D1	*Language-model definition*																	
O1.1	Textual description	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
O1.2	Formal textual model		×	×	×			×	×							×		
O1.3	Informal diagrammatic model																×	
O1.4	Formal diagrammatic model		×	×	×			×	×								×	×
D2	*Language-model formalization*																	
O2.1	M1 structural model															×		
O2.2	Profile (re-)definition	×		×				×		×	×	×	×	×			×	
O2.3	Metamodel extension	×	×	×	×	×	×	×	×	×	×				×			×
O2.4	Metamodel modification			(×)[a]					(×)	(×)	(×)				(×)			×
D3	*Language-model constraints*																	
O3.1	Constraint-language expression	×	×	×	×	×	×	×	×	×	×	×	×		×			
O3.2	Code annotation																	
O3.3	Constraining M2M/M2T transformation							×										
O3.4	Informal textual annotation	×	×	×	×	×	×	×	×	×	×		×		×			
O3.5	None/Not specified													×		×	×	×
D4	*Concrete-syntax definition*																	
O4.1	Model annotation	×		×				×		×	×	×	×	×			×	
O4.2	Diagrammatic syntax extension	×	×	×	×	×			×		×							
O4.3	Mixed syntax (foreign syntax)									×			×	×				
O4.4	Frontend-syntax extension (hybrid syntax)														×			
O4.5	Alternative syntax									×								
O4.6	Diagram symbol reuse	×		×				×		×	×	×	×	×	×		×	×
O4.7	None/Not specified						×									×		
D5	*Behavior specification*																	
O5.1	M1 behavioral model			×														
O5.2	Formal textual specification												×					
O5.3	Informal textual specification			×											×			
O5.4	Constraining model execution																	
O5.5	None/Not specified	×	×		×	×	×	×	×	×	×	×		×		×	×	×
D6	*Platform integration*																	
O6.1	Intermediate model representation		×					×				×						
O6.2	Generation template									×		×		×				
O6.3	API-based generator							×								×		
O6.4	(Direct) model execution		×															
O6.5	M2M transformation											×						
O6.6	None/Not specified	×		×	×	×	×		×		×		×		×		×	×

[a]Parentheses denote an underspecification; i.e. it is not unambiguously clear whether the option has been applied or not.

2.4 Notation Conventions

In this report, as well as in related papers [141], we use the following conventions to refer to decision points, corresponding options, associations between options, and DSML projects in a consistent way:

- *Dx*, where x is a number between one and six, refers to one of the D̲ecision points described in Section 2.1. For instance, decision point *D2* refers to the *language-model formalization* point.

- *Ox.y*, where x is a number between one and six and y is a number between one and seven, refers to a corresponding O̲ption y at decision point x. The allowed value of y depends on the number of design options for a specific decision point (e.g., we identified five options for D3 and seven options for D4; see Table 3 for an overview). For instance, *O5.3* refers to the third option (*informal textual specification*) at decision point D5 (*behavior specification*).

- In some cases we use an abbreviated notation *x.y* instead of *Ox.y*. The two notations are fully exchangeable and have identical meaning. The truncated form is only used when it is referred to multiple options under limited space (e.g., for the definition of option sets; for examples see Table 4). For instance, *6.5* ($\widehat{=}$*O6.5*) refers to the fifth option (*M2M transformation*) at decision point D6 (*platform integration*).

- *Ax*, where x is a number between one and *21*, refers to a dedicated A̲ssociation between two or more decision options of either one decision point or two or more decision points. The associations between options are numbered consecutively throughout this paper. For instance, *A16* refers to the association between options *O3.3*$\leftrightarrow$*O6.6* named *mandatory platform integration* (see Section 5.2).

- *Px*, where x is a number between one and 90, refers to a dedicated DSML P̲roject. The DSML projects are numbered consecutively, the first ten being our own developments (P1–P10). The remaining DSMLs were retrieved via the SLR (P11–P90). For instance, *P48* refers to the DSML named *SystemC* published in [126] (see Table 14).

3 Frequently Adopted Decision Options

We describe the design-decision making for a given UML/MOF-based DSML as a set of decisions or, more precisely, as a set of decision options (*option sets*, hereafter). Each decision is about evaluating and finally adopting one or several decision options, with the available decision options being listed by a corresponding decision record. This way, a decision links to and conforms with a decision record, such as the ones contained by our catalog (see also Figure 1). By collecting the decision options which apply to a given DSML as an option set, we characterize the DSML in the scope of the cataloged decision records (see Section 4).

At the time of conducting the SLR, this catalog offered 27 decision options to describe a DSML.[3] Six subsets of these decision options are associated with the six decision points (see Section 2.1). For example, four options are specific to defining a language model (D1; see Table 3). In this paper, we used this design-decision space to code the selected DSMLs according to this option scheme, thereby, yielding one characteristic *decision-option set* per DSML.[4]

In order to characterize the observable design-decision space for UML/MOF-based DSMLs, we mined for frequent and characteristic option sets using an analysis that is based on *frequent item-sets* [22, 61]. Frequent option sets are recurring combinations of decision options (or of other, smaller option subsets). Thus, we were interested in option sets adhering to certain constraints (i.e. minimum support, closedness, freeness, maximality; [22, 61]). *Support* denotes the occurrence frequency of a given option (sub-)set in a collection of observed option sets as any possible subset. We define an option (sub-)set as *relatively highly supported* when found three or more times in the DSML projects gathered via the SLR.[5] Otherwise, an option (sub-)set is defined as *relatively lowly supported*. Options not found in any of the selected DSML projects are defined as *candidate options* ([141] provides detailed background on those concepts in the context of our study).

Option (sub-)sets can express fragments of a DSML design as well as complete DSML designs (also called *prototype option-sets*). Option (sub-)sets can differ in terms of the number of options contained by them (size), in terms of their relatively higher or lower levels of support, and whether they are contained as-is in the base of observed option sets or not (i.e., whether they totally describe at least one DSML design alone, rather than a fragment of it).

The 80 DSMLs obtained via the SLR contain seven distinct prototype designs, that is, option sets which are frequent and describe entire DSML designs, with and without extensions. Six prototype option-sets come *with frequent extensions* (see Table 4). One prototype option-sets with frequent extensions is an option set which represents a *highest-common, largest option subset* (i.e., a design fragment of maximal size, which is frequently observed and has a relatively high support) which was also frequently found as complete DSML design. Because for this option set frequently occurring supersets exist, this (*evolutionary*) prototype option-sets is often extended by adding other (frequently observed) options [141].

For example, the option set of P26 (UML-PMS [57]) describes five observed and complete DSML designs (*frequency*) while it is found as a large subset in 25 other DSMLs ($support - frequency$) in an extended form. Five prototype option-sets involve UML profiles only (O2.2), just one frequently found prototype option-set builds solely on metamodel extensions (O2.3; e.g., P13 [13]). All six designs involve at least one concrete-syntax decision option (see also Figure 2, indicating D4 as mandatory). The only platform-integration option found adopted in three prototype option-sets (and twelve more extensions of it) are M2T generator templates (O6.2).

A seventh prototype option-set was found which comes *with infrequent exten-*

[3]Note that there are actually 31 decision codes (see, e.g., Table 3). Four of those codes/numbers serve for coding pseudo-decision options; e.g., not taking any decision.

[4]The complete list of each option set per DSML is shown in Table 14 in Appendix C.

[5]This way, we adopt a commonly followed rule of thumb in the software-pattern community (see, e.g., [30, 36]).

Table 4: Overview of the six prototype option-sets which are frequently extended (ordered by decreasing absolute support).

Prototype	Support (abs.)	Frequency (abs.)	DSMLs (excerpt)
{1.1,2.2,3.4,4.1,4.6}	30	5	P26 (UML-PMS [57]), P44 (PredefinedConstraints [37]), P68 (UML-AOF [84])
{1.1,2.2,3.1,4.1,4.6}	26	4	P18 (UML4PF [63, 64]), P62 (CUP [15]), P63 (REMP [78])
{1.1,1.4,2.2,4.1,4.6}	22	5	P25 (RichService [48]), P54 (SPArch [6]), P55 (MoDePeMART [23])
{1.1,2.2,4.1,4.6,6.2}	15	3	P51 (WCAAUML [73]), P64 (DPL [10]), P85 (WS-CM [86])
{1.1,2.2,3.1,3.4,4.1,4.6}	13	3	P22 (C2style [122]), P59 (SHP [106]), P70 (ArchitecturalPrimitives [160])
{1.1,2.3,4.6}	10	4	P13 (UML4SPM [13]), P14 (MDATC [14]), P49 (UML2Ext [24])

sions. A prototype option-set with infrequent extensions is an option set which represents a *lowest-common, largest option subset* (i.e., a design fragment of maximal size, which is frequently observed and has a relatively low support) which was also frequently found as complete DSML design. Because for this option set no frequent supersets exist, this prototype option-set is often employed as is. Extensions that add options to this (*evolutionary*) prototype are rarely observed [141]. The identified prototype option-set with infrequent extensions is realized by three DSMLs (P34, P37, and P52; see Table 5) and it is found extended twice ($support - frequency$). The option subset reflects a widely documented and recommended—but not necessarily frequently used—way of creating a DSML using UML profiles, by two-option strategies to define the language model (O1.1, O1.4) and the language-model constraints (O3.1, O3.4), respectively. The concrete-syntax choices O4.1 and O4.6 are often implied by adopting UML profiles.

Table 5: Overview of one prototype option-set which is infrequently extended.

Prototype	Support (abs.)	Frequency (abs.)	DSMLs (excerpt)
{1.1,1.4,2.2,3.1,3.4,4.1,4.6}	5	3	P34 (UACL [133]), P37 (SafeUML [166]), P52 (IEC61508 [119, 120])

The seven prototype option-sets which are realized as-is and with extensions for 63 out of the 80 DSMLs obtained via the SLR (∼79%) and for 68 out of the total 90 DSMLs (∼76%) are summarized in terms of their commonalities and differences as a feature diagram [39] in Figure 2.

The seven designs are combinations of nine options (see Table 6). By looking at these nine options and their characteristic combinations (see Tables 4 and 5), 27 out of both, the 80 DSMLs retrieved via the SLR (∼34%), and the total 90

DSMLs (30%) can be described in their entirety (prototype option-set).

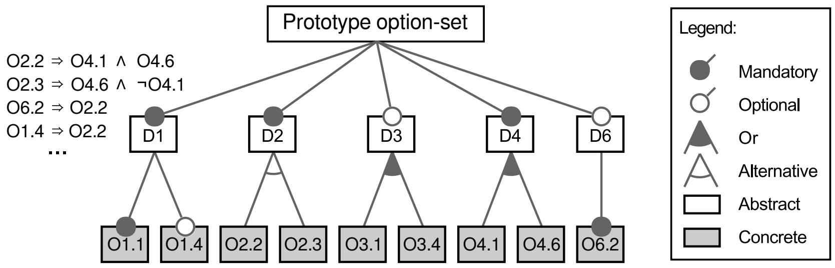


Figure 2: A feature model which represents the prototype option-sets found in the pool of 80 DSMLs; that is, each configuration of the feature space represents one of the seven observed prototype option-sets listed in Tables 4 and 5.

The seven prototype option-sets are composed of nine decision options. Table 6 provides thumbnail descriptions of these frequently adopted decision options. The complete decision record for each decision point is documented in the following Section 4.

4 Decision Records

This section presents the complete list of decision records, structured according to our template defined in Section 2.2, for each decision point as introduced in Section 2.1.

4.1 D1 Language-Model Definition

Problem statement. *How should the domain (or domain fragment) be described?*
Decision context. A prerequisite for DSML design is a systematic analysis and the structuring of the language domain. By applying a domain analysis method, such as domain-driven design [49], information about the selected domain is collected and evaluated (e.g. based on literature reviews, scenario analyzes, and collected expert knowledge). If the domain is already captured by an existing software system, artifacts related to the software system (e.g. code base, documentation, test suites) provide valuable input for the domain analysis. Based on this material, a structured domain description (referred to as a *generic language model* [152], hereafter) is defined. The domain description provides a domain definition, the domain vocabulary, and a catalog of domain abstractions and abstraction relations. The domain abstractions can be described using narrative text and/or using textual or diagrammatic specification formalism. These concept descriptions (models) form the basis for subsequent steps of formalizing a *core* language model (i.e, the abstract syntax of a DSML; see Section 4.2).
Decision options.

<u>*O1.1 Informal textual description:*</u>[6] Use primarily textual artifacts to iden-

[6]Frequently adopted options are underlined in the decision-record descriptions (for more information see Section 3).

Table 6: Thumbnail descriptions of nine frequently adopted decision options.

Problem statement	Options	Drivers
How should the domain (or domain fragment) be described?	O1.1 Textual description O1.4 Formal diagrammatic model	Availability of existing diagrammatic domain descriptions, intended target audience, correspondence mismatches with UML semantics, consistency preservation effort, cognitive effectiveness of a representational format
In which MOF/UML-compliant way should the domain abstractions be formalized?	O2.2 Profile (re-)definition O2.3 Metamodel extension	Overlap of DSML and UML domain spaces, degree of DSML expressiveness, portability and evolution requirements, compatibility with existing artifacts
Do we have to define constraints over the core language model(s)? If so, how should these constraints be expressed?	O3.1 Constraint-language expression O3.4 Informal textual annotation	Constraint formalization requirements, language-model checking time, integrated language-model constraint requirements, maintainability effort, portability requirements, language model and constraints conformance
In which representation should the domain modeler create models using the DSML?	O4.1 Model annotation O4.6 Diagram symbol reuse	Non-diagrammatic UML notation requirements, degree of cognitive expressiveness, disruptiveness, degree of required modeling-tool support
How should the DSML artifacts be mapped to (and/or integrated with) a software platform?	O6.2 Generation template	Targeting multiple platforms, maintainability effort of static code fragments, non-executable models

tify and to define domain abstractions and their relationships in an informal way. Exemplary text types are domain-vision (scoping) statements in narrative prose text, domain-distillation documents containing lists of core domain-abstractions [49], and/or domain-definition and feature tables [39].

O1.2 Formal textual model: Use textual formalisms to identify and to define domain abstractions and their relationships. For example, mathematical expressions (e.g. universal algebra [81]) or formal grammars (e.g. the Extended Backus-Naur Form [75]) provide means for well-formed and unambiguous definitions of domain concepts and relations.

O1.3 Informal diagrammatic model: Use ad hoc diagrams to identify and to describe domain abstractions and their relationships. Ad hoc diagrams are diagrammatic representations not being compliant to any standardized software modeling language and corresponding diagrammatic production rules. Examples are forms of visual concept modeling (e.g. early feature diagrams [39]) or pseudo UML diagrams (e.g. class diagram notations being used as re-composable drawing shapes).

O1.4 Formal diagrammatic model: Use diagrams defined by a (formally) specified/standardized modeling language (e.g. MOF, UML, ER, STATEMATE), which adopts a graphical representation (e.g. UML class models, UML activity models, and/or STATEMATE statecharts), to identify and to describe domain abstractions and their relationships.

Combination of options: For instance, to facilitate communicating concepts, diagrammatic models (O1.3, O1.4) can be used in support of a predominantly informal textual description (O1.1; see also related association A1 in Section 5.1). For explanatory purposes, normative and formal textual definitions (O1.2) are commonly supported by non-normative and informal textual descriptions (O1.1).

Decision drivers.

Availability of existing diagrammatic domain descriptions: If either formal or informal diagrammatic descriptions are available (e.g. a UML M1 class model), a domain description could be devised as a refinement (see also A6 in Section 5.2). For instance, by perfective refinement (e.g. turning an informal into a formally correct diagram; O1.4) and/or by refining the domain description as such (e.g. adding additional classes and associations to integrate previously uncovered domain abstractions).

Intended target audience: The language-model definition is used as a mutual communication vehicle for both, the domain experts and the DSML engineers. Depending on the domain, different views and notations must be considered. If, for example, the domain experts are mathematicians, a mathematical expression (O1.2) is suitable (see, e.g., [81]). In case of a DSML for software engineers (e.g. a DSML for defining software tests) the UML can be used to define the language model (O1.4). For non-technical business experts, prior experiences (see, e.g., [97, 131]) suggest that a process-oriented view (e.g. task and data flows) and process-oriented notations (e.g. UML activity models or BPMN models) are more adequate (O1.3, O1.4).

Correspondence mismatches with UML semantics: If the domain is described in a generic manner by adopting a formal notation (O1.2, O1.4), it needs to be transformed into a formal UML-compliant operationalization model (see D2 in Section 4.2 and also related association A6). Different transformation needs may result from various mismatches:

1. *A mismatch between modeling languages:* For example, when using ER modeling for describing domain abstractions, a transformation from ER elements into a UML class model or a MOF-compliant UML metamodel (extension) is needed. This bears the risk of impedance mismatches due to diverging definitional foundations (e.g. UML-elements have unique identifiers independent of attribute values; ER models use a minimal set of uniquely identifiable attributes for entity identification).

2. *A mismatch between modeling views:* There might also be a discrepancy between the views stressed by different model representations (e.g. ER diagrams cannot model behavior, for instance there are no UML operation or message type equivalents in ER diagrams). A domain description might stress a behavioral angle (e.g. using statecharts) while an operationalized language model (e.g. due to the specificity of the modeling language) requires additional structure details (e.g. properties of domain abstractions).

3. *A mismatch between different modeling levels:* Finally, even from the same view and within the same language framework, different levels of model granularity (e.g. for the MOF/UML context: meta-metamodel, metamodel, model, instance/repository model [116, 115]) can be adopted.

Having defined, for example, the domain description at the UML M1 level raises the issue of creating a mapping up to a metamodel (M2) for operationalizing the language model.

4. *Multiple mismatches:* Mismatches can occur in any combination of the previous three mismatch categories.

Consistency preservation effort: The effort required to preserve the consistency between different domain description artifacts (e.g. diagrams and textual descriptions) is a critical factor when considering combined options. The negative effects of introducing inconsistency, for instance between a diagram and its textual description, can be mitigated by declaring either representation the normative one.

Cognitive effectiveness of a representational format: A decision between any formal textual (O1.2) or any formal diagrammatic notation (O1.4) must consider the cognitive load caused by either representation choice. Irrespective of the target domain, diagrammatic representations benefits from their capacity to spatially group information bits otherwise spread in their textual form. Also, supporting visual perception and visual reasoning facilitate processing and communicating domain abstractions (see [74] for an overview). At the same time, there is a major tension between cognitive effectiveness of diagrams and the complexity of the perception task. This complexity is determined by the level of diagrammatic detail (e.g. in a formal notation) and the multiplicity of diagrams and views covered. For extensive domain descriptions or a high level of detail (views), textual representations (in support of visualizations) are considered more appropriate. This has been reported for inadequately designed visual variability models [35]. However, given the intentionally limited expressiveness of DSMLs (in terms of concepts covered), diagrammatic representations at the level of a generic domain description are suitable; especially if supported by (formal) textual descriptions to cover certain details. Besides, perceptional biases of the domain audience affect the cognitive effectiveness of the adopted representation type (see also above).

An overview of positive and negative links between decision drivers and available options is shown in Table 7. The following coding schema is used in the table. (+)+: (very) positive influence; o: no influence; (−)−: (very) negative influence. An option having either a (very) positive or a (very) negative influence—depending on the intended DSML's application domain, professional background as well as prior knowledge and experience of users etc. (see above)—is denoted by (+)+/(−)−.

Table 7: Positive/negative links between drivers and options.

Driver/Option	O1.1	O1.2	O1.3	O1.4
Availability of existing diagrammatic domain descriptions	+	+	++	++
Intended target audience	o	++/−−	++/−−	++/−−
Correspondence mismatches with UML semantics	o	−	o	−
Consistency preservation effort	−	−	−	−
Cognitive effectiveness of a representational format	o	+/−	o	+/−

Decision consequences. The initial phase of the generic language-model definition has to cover all domain-specific concepts from the selected target domain and precedes the formalization via the MOF/UML.

Output artifacts: These are the core language-model concepts, whereas the description form depends on the application domain and involved domain experts. The generic language model description can be informal (O1.1, O1.3), in a structured form (O1.2, O1.4), or defined via combinations of these options.

Mapping to metamodeling infrastructure: If the definition is not based on the MOF (e.g. an option in O1.2 or O1.4), the concepts have to be mapped to MOF-equivalent elements (correspondence mismatches can occur; see the *drivers* section above).

Application. As all our DSMLs were created from scratch (see Table 1), there were no existing domain descriptions available (e.g. in terms of code or documentation artifacts). This context affected our decisions as the option space was not constrained per se: In a combined form, we adopted formal textual (O1.2) and UML/MOF-based diagrammatic definitions (O1.4) in P2–P4, P7, and P8. All generic language-model definitions (e.g. P1–P10 and P17, P30, P39, P53, P58, P60, P61 in Table 3) are defined via or accompanied by informal textual descriptions (O1.1). Additionally, P60 serves as an example for the application of informal diagrammatic models (O1.3).

Sketch. An excerpt from a formal and textual domain description (O1.2) in combination with surrounding textual explanations (O1.1) is shown beneath. The example is taken from P7 which requires a definitional basis to express data flow semantics (i.e., object flows, later to be mapped to object flows in UML activities). In this context, a selector expression for collecting succeeding object nodes is needed. That is, the set of object nodes for which a direct path exists between a source and a target object node must be selectable. The selector definition expresses certain conditions, for instance, the object flow path must only include arcs or control nodes, whereas tasks or intermediary object nodes are to be excluded. The domain description adopts a set-theoretical model (O1.2) to express the selector operation as a mapping and the selection conditions as mapping constraints:

The mapping $successors : O \mapsto \mathcal{P}(O)$ is called **succeeding object nodes**. For $successors(o_s) = O_{succ}$ with $o_s \in O$ and $O_{succ} \subseteq O$ we call o_s source node and each $o_t \in O_{succ}$ a direct successor of o_s. In particular, O_{succ} is the set of object nodes for which a path exists between o_s and each $o_t \in O_{succ}$. Formally: $\forall o_s \in O, o_t \in successors(o_s) : ofpath(o_s, o_t) \neq \emptyset$.

4.2 D2 Language-Model Formalization

Problem statement. *In which MOF/UML-compliant way should the domain concepts be formalized?*

Decision context. After the identification of language-model concepts, the corresponding definitions serve as input for the phase of formalizing the domain constructs into a MOF/UML compliant core language model.

Decision options. For UML-based DSMLs, the language model can be formalized via dedicated language extension constructs (such as UML profiles) or by extending the modeling language to provide the required semantics (see, e.g. [28, 116]).

O2.1 M1 structural model: Implement the core language model using structural UML models at level M1. In a class model, for instance, domain abstractions can be expressed as classes and their relationships as associations. Other examples are composite structure, component, and package diagrams.

O2.2 Profile (re-)definition: Implement the core language model by creating (or by adapting existing) UML profiles. A profile consists of a set of stereotypes which define how an *existing* UML metaclass may be extended.

O2.3 Metamodel extension: Implement the core language model by creating one or several metamodel extensions. A metamodel extension introduces new metaclasses and/or new associations between metaclasses to the UML metamodel or to other, pre-existing metamodel extensions [116, 115]. The extension elements are typically organized into dedicated «`metamodel`» packages. The structure and semantics of existing elements of the UML metamodel are preserved.

O2.4 Metamodel modification: Implement the language model by creating one or several MOF-based metamodel extensions which modify existing metaclasses; for example, by changing the type of a class property or by redefining existing associations [116, 115]. The extension elements are typically organized into dedicated «`metamodel`» packages.

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

Decision drivers. An overview of positive and negative links between decision drivers and available options is shown in Table 8.

Overlap of DSML and UML domain spaces: The degree of overlap between the domain space of the DSML concepts and the general purpose language constructs (i.e., the UML specification) has, for instance, a direct impact on whether a profile definition is sufficient (O2.2) or on whether a metamodel extension/-modification is needed (O2.3, O2.4).

Degree of DSML expressiveness: A UML profile (O2.2) can only customize a metamodel in such a way that the profile semantics do not conflict with the semantics of the referenced metamodel. In particular, UML profiles cannot add new metaclasses to the UML metaclass hierarchy or modify constraints that apply to the extended metaclasses (see, e.g., [146]). Therefore, profile constraints may only define well-formed rules that are more constraining (but consistent with) those specified by the metamodel [116] (see also A9). In contrast, a metamodel extension/modification (O2.3, O2.4) is only limited by the constraints imposed by the MOF metamodel (i.e. the abstract syntax of the UML can be extended via new metaclasses and associations between metaclasses; see also A11).

Portability and evolution requirements: A newly created metamodel (O2.3, O2.4) is an extension of a certain version of the UML specification. Thus, the domain-specific metamodel extension possibly needs to be adapted to conform with newly released OMG specifications. Re-usability of a UML extension is also affected by being either compliant with the UML standard (e.g. O2.2 or O2.3) or not (e.g. O2.4).

Compatibility with existing artifacts: Pre-existing DSMLs, software systems, and tool support have a direct impact on the design process of a DSML in terms of compatibility requirements and integration possibilities (see also A17 and A20). For instance, the UML specification defines a standardized way to use icons and display options for profiles (O2.2). Tool support for authoring UML models and profiles (O2.1 and O2.2) is widely available (see, e.g., [146]).

Table 8: Positive/negative links between drivers and options.

Driver/Option	O2.1	O2.2	O2.3	O2.4
Overlap of DSML and UML domain spaces	+/−	+/−	+/−	+/−
Degree of DSML expressiveness	−−	−	+	++
Portability and evolution requirements	+	+	−	−−
Compatibility with existing artifacts	++	++	−	−

Decision consequences.

Formalization style dependencies: Certain dependencies can result from combined language-model formalizations (e.g. O2.2 and O2.3; see also A13). For instance, profiles are dependent on the corresponding metamodel (i.e., the UML). If a profile is combined with a metamodel modification (O2.4), changes to the metamodel can affect the respective stereotypes (e.g. if a stereotype-extended metaclass is modified).

Language-model ambiguities: If no further constraints to the language model are specified (see Decision D3), the language model must be fully and unambiguously defined using the chosen formalization option and implicitly enforced restrictions (e.g. by using profiles and thus inheriting all semantics from the UML metamodel; O2.2; see also A7 and A9).

Application. In all our DSML projects, we formalized the language models as metamodel extensions (O2.3). Additionally, profiles (O2.2) were employed in P1, P3, P7, P9, and P10. Therefore, we effectively adopted combined strategies. In related approaches, we also found the application of M1 structural models (O2.1, e.g. in P58) and the modification of the UML metamodel (O2.4, e.g. in P61) for the formalization of the language model. As an example for O2.4, P61 documents a UML metamodel modification by adding new attributes to existing UML classes (e.g. to classes `Class` and `Property`). This is in contrast to several other approaches which employ metamodel extensions (O2.3), but do not explicitly document whether they perform modifications to the UML metamodel (O2.4), as well. For instance, in P53, existing classes from the UML metamodel (e.g. `UseCase`) are associated with newly defined classes (e.g. `UseCaseDescription`). The metamodel definition in P53 remains uncertain regarding the ownership of association ends: (1) Both ends could be owned by the association (O2.3); (2) one end could be owned by the association, one by a class (O2.3 or O2.4, depending if the owning class is coming from the UML metamodel); or (3) both ends could be owned by their corresponding classes (O2.4). To avoid such ambiguities, association end ownership can be made explicit with the dot-notation [116]. Furthermore, accompanying textual annotations can provide clarifying details. In Tables 3 and 13 such underspecified DSML projects are denoted with an option mark being put in parentheses.

Sketch. Figure 3 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 (O2.3) and, on the right hand side, a UML profile specification named `SOF::Services` (O2.2). 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 [67, 68, 72]) with the SoaML specification [110].

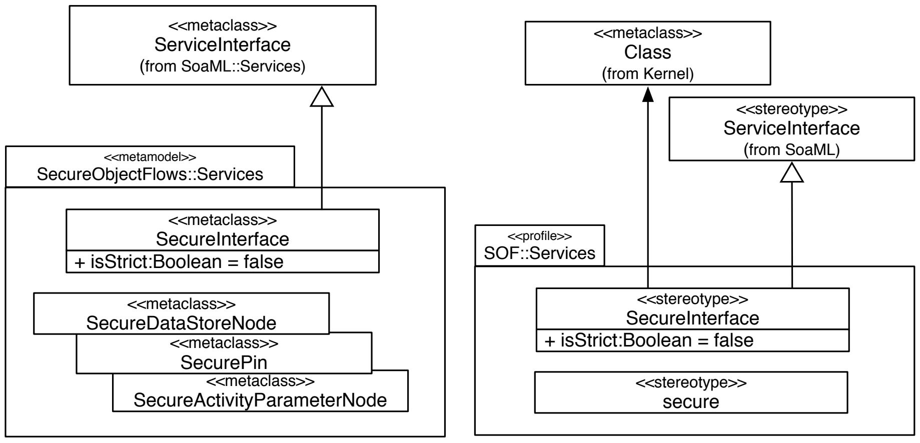


Figure 3: Exemplary UML metamodel extension and profile definition [67].

4.3 D3 Language-Model Constraints

Problem statement. *Do we have to define constraints over the language model(s)? If so, how should these constraints be expressed?*
Decision context. A core language model has been formalized using either a UML metamodel extension/modification, a UML profile, or a UML class model (D2). The resulting language model describes the domain-specific language in terms of its language elements and their interrelations. The definition 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 abstractions that are relevant for the description of the target domain. Examples are invariants for domain abstractions, pre- and post-conditions, as well as guard conditions. As a result, the language-model formalization could be incomplete or ambiguous (see also A9).

If the language model has been realized by creating multiple formalizations (e.g. multiple profiles), there is an additional risk of introducing inconsistencies if the DSML can be used in different configurations (e.g. different profile compositions). Consider, for example, profiles which integrate two (independent) UML extensions.
Decision options.

O3.1 Constraint-language expression: Make language-model constraints explicit using a constraint-expression language, for example, via the Object Constraint Language (OCL [118]) or the Epsilon Validation Language (EVL [89]).

(*O3.2 Code annotation:*)[7] Make language-model constraints explicit using expressions in (or a specialized sub-language embedded within) a general-purpose programming language (GPL). This programming language can be the

[7]Candidate options not applied in any of the selected DSML projects are put in parentheses in the decision-record descriptions.

host language of the DSML. In the UML, for instance, this can be realized by using model annotations and UML's `OpaqueExpression` [116] (see also A15).

(*O3.3 Constraining model transformation:*) Express language-model constraints via model (M2T/M2M) transformations. The respective 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. As for M2M transformations, for example, conditional statements in the Epsilon Transformation Language (ETL [89]) based on Epsilon Object Language (EOL [89]) expressions can be used to specify structural constraints over the language model (i.e. at the model instance level) and to enforce them in each transformation run.

O3.4 Informal textual annotation: Use informal and unstructured text annotations to capture constraint descriptions in the core language model (e.g. prose text in UML comments). Certain constraints (e.g. temporal bindings) elicited from the target domain cannot be captured sufficiently via evaluable expressions (i.e. constraint-language expressions or code annotations) and/or the constraints are intended to serve a documentary purpose (esp. annotations for domain experts).

O3.5 None/Not specified: Static constraints over the language model are not made explicit in (or along with) the language model.

Combination of options: For instance, textual annotations (prose text) can be used in addition to constraint-language expressions to provide natural-language constraint descriptions for readers not familiar with a specific constraint language (see also A2).

Decision drivers. An overview of positive and negative links between decision drivers and available options is shown in Table 9.

Constraint formalization requirements: 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 (O3.4). When the language model is maturing due to subsequent iterations, these annotations can be changed into evaluable expressions (O3.1–O3.3).

Language-model checking time: If tool integration for model checking is a requirement, we have to choose one ore more of the options O3.1–O3.3 (see also A3 and A15). 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 transformation expressions (O3.3) realizes the latest possible checking point. Therefore, this option does not offer any constraint-based feedback during model development.

Integrated language-model constraint requirements: Constraint-language expressions (O3.1) are developed with the purpose of integrating the constraints with the (meta)model representations. Examples are standard-compliant and vendor-specific OCL expressions for the UML. Models and constraints can also be integrated, for instance, via programming-language-based expressions over secondary Ecore representations of UML models (e.g. Eclipse EValidator framework; O3.2) as well as via natural-language UML comments (O3.4). The latter two options having the drawback that they are specific to a certain platform (O3.2) and lacking automatic evaluation (O3.4).

Maintainability effort: Explicitly defined model constraints (O3.1–O3.3) create structured text artifacts which must be maintained along with the model artifacts (e.g. a corresponding XMI representation). Toolkits and their model representations offer different strategies for this purpose, for example 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 requirements: If the portability of constraints between different MDD toolkits (e.g. Eclipse MDT, Rational Software Architect, MagicDraw, Dresden OCL) is a mandatory requirement, platform-dependent options O3.2 and O3.3 most often have to be excluded. However, due to version incompatibilities and vendor-specific constraint-language dialects (e.g. Eclipse MDT OCL), even O3.1 does not guarantee basic portability for the ambiguously specified sections of the UML/OCL specifications (esp. for semantic variation points such as navigating stereotypes in model instances or for transitive quantifiers such as `closure`).

Conformance between language model and constraints: Constraints on the language model can be defined separately from the corresponding metamodel (e.g. using code annotations; O3.2) or at a later stage (e.g. for M2T transformations; O3.3). 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 metamodel changes (e.g. OCL navigation expressions; O3.1).

Table 9: Positive/negative links between drivers and options.

Driver/Option	O3.1	O3.2	O3.3	O3.4	O3.5
Constraint formalization requirements	–	–	–	+	o
Language-model checking time	++	++	+	– –	– –
Integrated language-model constraint requirements	++	+	o	+	o
Maintainability effort	–	–	–	o	o
Portability requirements	+	– –	– –	o	o
Conformance between language model and constraints	–	–	–	o	o

Decision consequences.

Output artifacts: When we choose to define constraints for a DSML, we receive a catalog of language-model constraints that offer additional structural semantics for the DSML. Depending on the actual option(s) adopted, an explicit catalog of formally defined constraints (e.g. via OCL) is available which can be used to (automatically) test the validity of UML diagrams modeled with the corresponding DSML. Moreover, a set of M2M/M2T transformation template expressions used to validate model instances or code/textual annotations can be produced as output artifacts. The decision which kind of constraint definition is the most suitable is highly dependent on the actual stage of the DSML project, available tool support, and tool integration (see, e.g., A15 and A16). The DSML core language model and the DSML language-model constraints serve as an input for the subsequent definition of the DSML's concrete syntax and behavior specification.

Tool support: The availability of tool support for different lifecylce stages

(development, verification, evaluation) of formally defined language-model constraint expressions (O3.1) determines whether adopting a specification or expression language is justified beyond completing and disambiguating the formalized language model. Support for constraint development relates to IDE integration and IDE awareness of constraint expressions. Another important issue is how constraint artifacts can be managed along with model artifacts. Support for constraint verification includes, for example, model-checking support to assess the satisfiability of expressions. Finally, support for constraint evaluation requires an execution engines (e.g. an OCL engine) for a given metamodeling infrastructure and linking back evaluation results into the development support. Depending on the specification or expression language adopted (e.g. OCL, EVL), deployment of the DSML and the models created using the DSML might be limited to certain MDD tool chains providing the necessary capabilities (verification, evaluation).
Application. In our DSMLs, we encountered all options but code annotations (O3.2) and unconstrained language models (O3.5; for examples that use O3.5 see, e.g., P39, P58, P60, and P61). In particular, we provide constraint-language expressions (O3.1) via the OCL for all of our DSMLs. This is because we needed to define precise execution semantics for extended UML activities (such as token flows in P1) and of the UML state machines (state/transition models in P10). In eight out of ten DSMLs (P2–P9), these semantics are based on the same generic and MOF-compliant metamodel and provide 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 (O3.2) were not considered because the additional model constraints should not be specific to any platform (e.g. model representation APIs, generator language). Note that while we did not find any source for constraining code annotations in our SLR, this may still be a viable option for MDD environments that use a single, pre-determined platform technology (such as a Java or a C# framework for example). In P7, we additionally incorporated constraining M2T transformations (O3.3). Informal textual annotations (O3.4) are either used to complement OCL constraints or as full substitutes for otherwise formally expressed constraints.
Sketch. Consider the following excerpt from P8: For a UML activity, each action can be guarded by a constraint whose conditions refer to a set of operands and checking operations. At runtime (level M0), the operations are called to evaluate whether an action should be entered, depending upon some contextual state. Constraint 1 shows a constraint-language expression (OCL) accompanied by 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 folded together to yield the runtime evaluation of the guard) become manifest at the occurrence-level of an activity only (M0).

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 fulfilled$_{\mathrm{CD}}$ Operations must evaluate to true to fulfill the corresponding

ContextCondition.

4.4 D4 Concrete-Syntax Definition

Problem statement. *In which representation should the domain modeler create models using the DSML?*
Decision context. The concrete syntax serves as the DSML's interface. Different syntax types can be defined and tailored to the need of the modeler. For instance, different syntax styles may be chosen depending on the modeler's domain and/or software-technical proficiency.

The UML has a concrete syntax that provides a visual notation, with its symbol vocabulary being organized into 14 diagram types [116]. The number of distinct graphical symbols applicable in these diagram types ranges from 8 (in communication diagrams) to 60 (e.g. in class diagrams) [104]. A DSML derived from the MOF and/or the UML can add new elements to this symbol vocabulary or reuse existing ones (see also A10 and A17.

In addition, secondary, non-diagrammatic representation candidates are available for the MOF and for the UML. Important examples are textual, tree-structured, and tabular notations (see, e.g., [162]). A *textual* concrete syntax expresses DSML models in a text-based format. Typically, textual grammars are used to define a textual concrete syntax (e.g. via the Extended Backus-Naur Form [75]). Based on such a grammar a parser infrastructure is build (in some cases the parser can even be generated automatically). A tree-structured concrete syntax is a graphical, but non-diagrammatic representation. It represents a MOF or a UML model as a nested, collapsible structure with composite and leaf elements having text labels and/or symbols (for example, the default UML editor provided by the Eclipse MDT uses a tree structure). A tabular and form-based concrete syntax organizes DSML elements in a table-like layout. Textual labels and corresponding input fields populate a structure of table rows and columns (such a syntax is similar to the user interface of language workbenches [53]).
Decision options.

O4.1 Model annotation: Attach UML comments as concrete-syntax cues to a UML model, containing complementary domain information such as keywords, narrative statements, or formal definitions (see, e.g., [85]). The expressions can be predefined at the level of the language-model definition or they are tailored for each instance. In addition, the UML specification describes the use of keywords and maintains a list of predefined keywords [116].

O4.2 Diagrammatic syntax extension: Extend one or multiple UML diagram types by creating novel symbols adding to the basic UML symbol set. The new symbols can be derived from existing shapes. The DSML is to be used primarily in a diagrammatic manner adopting these extended UML diagram types. In principle, the design space for the new symbols is unlimited but has to be aligned with the requirements of the target domain. However, existing guidelines for designing UML symbols should be considered (e.g. avoidance of synographs; see, e.g., [104]). The symbol description can be structured according to the form adopted by the UML specification documents [116]: 1) A descriptive and detailed statement about each symbol, 2) the optional elements of the symbols, 3) exact styling guidelines for the symbol's components (e.g. text labels, font faces), 4) an abstracted example of each symbol, and 5) a concrete example of a model that uses the new symbol(s). This facilitates cross-reading between the UML specification and the DSML extension document. A notable example of a diagrammatic extension is the option to equip UML stereotype elements with dedicated icons which appear as addition to the standard notions of stereotyped elements (e.g. tags or nested icons in classifier rectangles [116]).

O4.3 Mixed syntax (foreign syntax): Create your DSML's concrete syntax as either a non-diagrammatic syntax (textual, tree-based, or tabular) or as a diagrammatic syntax not integrated with the UML's. Thus, in contrast to O4.2, this option would define a new and domain-specific diagram type. Hence, the DSML concrete syntax is independent of and thereby *foreign* to the basic UML symbol vocabulary. For example, model specifications in the foreign syntax are managed and stored separately from the UML diagrams. The UML base syntax is not extended, the symbols of the refined or modified metaclasses are reused (see O4.6). The extension syntax maps only to the DSML abstract syntax, no UML metamodel element is covered. The foreign syntax is used exclusively to model the domain-specific parts of an extended UML model. For instance, a non-diagrammatic foreign syntax can be embedded into the primary, diagrammatic UML syntax (e.g. via UML comments or expression elements). In the resulting mixed syntax, there is a hierarchical relation between the basic UML diagram notation and the nested foreign notation. To fully capture a DSML model, the two syntaxes are mutually dependent. The unextended UML base syntax cannot capture DSML specifics (unambiguously), the foreign syntax cannot represent basic UML concepts.

O4.4 Frontend-syntax extension (hybrid syntax): Create your DSML's concrete syntax as a non-diagrammatic syntax (textual, tree-based, tabular) which extends a non-diagrammatic frontend syntax to the UML (e.g. a textual UML notation). As a result, the syntax extension represents a visual vocabulary independent of the graphical UML base syntax. The UML base syntax remains unchanged, the symbols of the refined or modified metaclasses are reused (see O4.6). The extended frontend syntax has more expressive power than the UML base syntax because the modeler can express DSML models unambiguously in the frontend syntax. In the UML base syntax, the notational defaults (i.e., base symbols representing DSML elements) limit the expressiveness (i.e., instances of DSML elements cannot be distinguished from standard UML elements).

O4.5 Alternative syntax: Create a diagrammatic syntax extension to the UML (O4.2) and provide one or more alternative syntaxes (see O4.3 and O4.4). As a result, DSML models can either be expressed diagrammatically in the extended UML notation, as a combination of standard UML diagrams with an (embedded) foreign syntax, or as a non-diagrammatic specification in the extended frontend syntax. Each of these three variants has equal expressive power in terms of abstract syntax elements covered. Lossless back-and-forth transformations are possible.

O4.6 Diagram symbol reuse: Reuse built-in UML diagram symbols without modification. No custom, DSML-specific extension to the standard UML symbol vocabulary is created. With the family of UML specifications [116] not being explicit about the case of undeclared notations (i.e., missing "Notation" sub clauses), the effective reuse of symbols defined for UML metaclasses refined by the DSML must be stated explicitly (see also A12). This resembles the practice applied in the UML specification itself. For example, for the `Class` metaclass which specializes the `Classifier` metaclass. Section 7.3.7 of the UML standard [116] says:

> **Notation**
> ...
> A class is shown using the classifier symbol.
> ...

O4.7 None/Not specified: The DSML specification does not contain any notational details, not even the explicit reuse of diagram symbols (see O4.6). The concrete syntax remains undefined.

Decision drivers. An overview of positive and negative links between decision drivers and available options is shown in Table 10.

Non-diagrammatic UML notation requirements: Textual notations [59] for the UML are auxiliary representations and act as frontend syntaxes (O4.4). As an important example, in XMI [117] a DSML concrete syntax extension would be realized as an XML schema which extends the XMI schema itself. Besides, as XMI is meant to represent MOF models natively, the availability of a UML extension for XMI (e.g. the Eclipse UML2XMI schema) is a prerequisite. Major pitfalls of an XMI-based extension are syntactic complexity and cognitive load imposed on the modeler via the XML representation. Also, the respective UML extensions are often vendor or tool specific.

HUTN [108] is an alternative that suffers from similar limitations as XMI. In particular, HUTN is specified for use with the MOF or MOF-like modeling infrastructures, such as Ecore. Thus, via HUTN only MOF views of the UML can be captured. For example, while class and object models (diagrams) map naturally to MOF models (HUTN specifications), other types of diagrams such as UML activities are represented via their repository model notation [21, 116]. However, as a repository model, an activity is presented as an instance structure of the (extended) UML metamodel, omitting any process flow notation. This surrogate view is therefore not lossless and the predominantly structural repository perspective misses the process flow metaphor which might be critical for the target domain of a DSML (see D1 in Section 4.1). Another comparable but vendor-specific notation is offered by TextUML [33].

Moreover, other non-standard (grammar-based) textual notations that explicitly target (subsets of) the UML's abstract syntax exist. For example, the Activity Diagram Linear Form (ADLF [52]) provides a textual representation (and parser infrastructure) based on a Yacc grammar specification for a subset of UML activities (action nodes, control flows, control nodes). Similar text-based but feature-wise incomplete forms for other UML metamodel fragments have been proposed (see, e.g., [65] for an example of UML state machines). A major limitation of such approaches is the missing support of, e.g., notations interlacing between different diagram types of the UML (for example, nested interactions in activities).

For a variety of tooling purposes, freestanding textual layout descriptions for the UML come with a variety of modeling and auxiliary tools. Important examples are direct diagram specifications (e.g. Graphviz-like specifications [7]) or intermediate textual notations (e.g. yUML [62]) for rendering and layouting UML diagrams, also in an embedded manner for document processors. However, these notations are freestanding in the sense that they are *not* meant to map to a complete (sub-)set of the UML abstract syntax and to conform to notational restrictions derived from the abstract syntax. Rather, these notations serve backend purposes (e.g. diagram rendering and formatting).

Degree of cognitive expressiveness: UML stereotypes have limited visual expressiveness in contrast to tailored model elements (O4.2) which are not restricted with respect to their visual representation. A textual representation can have a steeper learning curve but might be used to express models in a shorter period of time (for advanced users). Nevertheless, it is often not the best way to get an overview (i.e. not well-suited for large models). A tree-based syntax fits, for instance, a hierarchically structured DSML, but falls short in an adequate representation of process flow constructs such as loops and sequences.

Disruptiveness: The UML includes symbolic (e.g., class, state, association, generalization) as well as iconic signs (e.g., actor, component, fork and join nodes) for its graphical notation (concrete syntax) [116]. The perception of symbolic and iconic signs differ and is influenced by the intended application domain as well as the professional background and individual preferences of model users. A set of experiments published as [147] provides evidence that UML models (class and collaboration diagrams) mostly consisting of iconic signs (in the form of stereotype icons) improve comprehension compared to models mostly consisting of symbolic signs (annotated non-stereotyped elements). These findings are supported by results of another study in which the authors conclude that "iconic UML graphical notations are more accurately interpreted by subjects and that the number of connotations is lower for iconic UML graphical notations than for symbolic UML graphical notations" [140]. While a DSML designer must keep this information in mind, the concrete syntax must also be developed to fit its purpose (i.e. conform to domain requirements, integrate with other DSMLs etc.). For example, when the domain's graphical notation set has a long history of symbolic signs, a change may cause confusion and comprehension problems which may lead to a decrease of DSML users' efficiency.

In the UML, stereotypes (O4.1/O4.6 without icons) are the native domain-specific visual presentation option. As stereotypes employ the same notation as classes, they count as symbolic signs unless icons are graphically attached to the model elements extended by the stereotype (see also A10). As an example of domain-specific notation characteristics, software engineers may be most familiar with textual syntaxes (O4.3, O4.4). Regarding tool support, Eclipse MDT provides a tree-based view (O4.4) in one of its standard UML model editors. Moreover, no explicit concrete syntax (O4.7) might be necessary if the DSML only defines language-model constraints, limited behavioral specifications, or provides tool support for standard UML means (see also A14).

Degree of required modeling-tool support: A textual concrete syntax (O4.4) can be processed by a parser and (most often) does not need specific editor tools (as they are required for a graphical/diagrammatic syntax). It can be integrated with existing developer tools, such as version management systems or diff and merge tools (an advantage for joint modeling as well as model evolution). Due to its hierarchical form, a tree-based syntax is easy to be serialized to or created from XML-based textual representations (e.g. XMI). Modeling support for UML stereotypes (O4.1/O4.6) as well as for tree-based syntaxes exists in standard tools, but must be explicitly integrated for new graphical elements (O4.2).

Decision consequences.

Usability evaluation: The DSML syntax is especially important from the DSML user perspective. If a DSML is mainly used by non-programmers, a

Table 10: Positive/negative links between drivers and options.

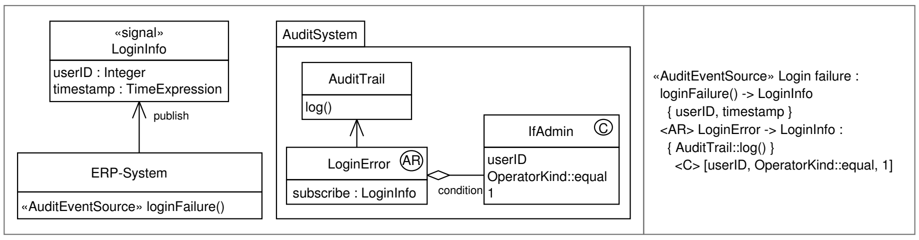
Driver/Option	O4.1	O4.2	O4.3	O4.4	O4.5	O4.6	O4.7
Non-diagrammatic UML notation requirements	o	o	−	−	−	o	o
Degree of cognitive expressiveness	−	+	+/−	+/−	+/−	−	o
Disruptiveness	−	+	+	+	++	−	+/−
Degree of required modeling-tool support	++	−	+/−	+	−−	++	o

special focus on usability aspects is needed.

Output artifacts: After defining suitable graphical and/or textual notation symbols, as well as composition and production rules, we receive the DSML concrete syntax definition as an output from this decision point. Together with all other artifacts created during the DSML development process, the concrete syntax definition is then mapped to the features of a selected software platform.

Application. In our case studies we provide a couple of different concrete syntax definitions such as UML stereotype-specific annotations for reusing symbols (P1, P3, P7, P9, P10), new diagrammatic modeling elements (P1–P5, P8, P10), and an alternative syntax style (a combination of a diagrammatic syntax extension and an alternative foreign syntax; P9). Additionally, one of our DSMLs applies extended language-model constraints and does not need a concrete syntax (P6). Approaches from the related work use the full range of options for concrete syntax definitions (see, e.g., P17, P30, P53, P58, and P60). As an example for an option not covered so far, P53 defines a textual frontend-syntax extension.

Sketch. Figure 4 shows an example of two concrete syntax definitions consisting of a diagrammatic representation on the left hand side and its textual equivalent on the right hand side (excerpt taken from P9). In the example, an audit rule is specified for an information system which records data when a failed login attempt from a user with administrator privileges is recognized (see [69] for details). Both syntaxes operate on the same abstraction level and can be used complementary (O4.5).



```
«AuditEventSource» Login failure :
 loginFailure() -> LoginInfo
  { userID, timestamp }
 <AR> LoginError -> LoginInfo :
  { AuditTrail::log() }
    <C> [userID, OperatorKind::equal, 1]
```

Figure 4: Exemplary graphical and textual concrete syntax [69].

4.5 D5 Behavior Specification

Problem statement. *Do we have to define (additional) behavioral semantics for the DSML? If so, how should the additional behavior of DSML elements be defined?*

Decision context. The behavioral specification of a DSML (sometimes referred to as *dynamic semantics*) defines the behavioral effects that result from using one or more DSML language element(s). It determines how the language elements of the DSML interact to produce the behavior intended by the DSML engineer. Moreover, the behavior specification defines how the DSML language elements can interact at runtime [152]. The behavior of a DSML can be defined in various ways, for instance, via behavioral models or (formal as well as informal) textual specifications.

Explicitly specified behavior allows for a correct mapping of the (platform-independent) DSML specifications to a certain software platform (see D6 in Section 4.6). However, sometimes a DSML's behavior is not explicitly specified. Behavioral relationships may also emerge from the language-model formalization or the language-model constraints definition. Furthermore, accompanying textual descriptions, while not formally specifying behavior, may hint at intended usage and interaction scenarios of DSML language elements. If no behavioral specification exists, the DSML's runtime behavior is implicitly defined via the DSML's platform integration (e.g. via chains of method calls in a source code implementation).

Decision options.

O5.1 M1 behavioral model: Specify additional behavior of language-model elements using UML behavioral models at level M1 (e.g. state machines, interaction diagrams, or activity diagrams; see, e.g., A18). For instance, in the UML, a classifier can reference owned behavior specifications. Behavior is then executed in the context of the directly owning classifier [116].

O5.2 Formal textual specification: Specify the additional DSML behavior using a textual formalism (e.g. algebraic expressions). In this context, a formal textual specification is a set of expressions in a formal language at some level of abstraction [94] with the purpose that its correctness can be proven (e.g. by using the Z notation [76, 77]).

O5.3 Informal textual specification: In contrast to formal specifications, informal textual specifications may be ambiguous. Thus, they are used to informally specify the behavior of a DSML, for example via narrative prose text.

(*O5.4 Constraining model execution:*) Implement behavioral constraints in a (partial) execution engine for DSML models, so that they are evaluated at model-processing and/or runtime. The constraints become enforced at model-processing and/or runtime by the execution engine (e.g. xMOF [99] based on the fUML [113], with constraints defined over fUML activities).

O5.5 None/Not specified: No explicit behavior specification.

Combination of Options: For instance, textual comments (O5.3) are used to annotate models (O5.1) or to clarify formal specifications (O5.2).

Decision drivers. An overview of positive and negative links between decision drivers and available options is shown in Table 11.

Model consistency preservation: UML behavioral models (O5.1) allow for a native integration of behavioral semantics into UML/MOF-based DSMLs (see also A18). For example, behavior of a DSML element can be defined via an "owned behavior" specification [116]. This facilitates support for integrated modeling tools as well as execution engines (O5.4). Nevertheless, some semantics elements may be left unconstrained in the specifications to defer behavioral interpretations to the platform integration phase (which could slightly differ from one software platform to the other; e.g. the semantics of concurrency or event dispatch scheduling in the fUML [113]).

Behavioral definition requirements: For narrow domains, no explicit behavior specification may be needed (O5.5). Behavioral intentions can be drawn from the descriptive part of the language model, its formalization, and the language-model constraints. In such a case, an explicit behavior specifications may be expendable.

Limited expressiveness: If, for some behavioral expressions, it is not feasible or even impossible to be sufficiently expressed via models (O5.1) or formal statements (O5.2), informal textual specifications are an option (O5.3). For instance, the specification of the fUML execution model incorporates a degree of generality for the semantics of inter-object communication mechanisms [113]. The execution model is written as if all communications are perfectly reliable and deterministic (e.g. it is assumed that signals and messages are never lost or duplicated) which is not realistic. As raising exceptions and exception handling are excluded from the fUML specification, an informal and descriptive addition (O5.3) may be useful.

Behavior verification requirements: Depending on the language and/or formalism that is to specify behavior, the correctness of formal specifications (O5.2) and executable (i.e., well-formed) models (O5.4) can be proven (see, e.g., [32, 76, 77]). If it is the objective to verify all artifacts a DSML consists of (such as, language model, language-model constraints, behavior specification, platform-specific artifacts), O5.2 and O5.4 are options. This is in contrast to non-executable behavioral models (O5.1) and informal textual specifications (O5.3) for which behavioral semantics may remain underspecified. The benefit of proving the correct behavior of a DSML may come with the additional effort of a precise specification and the development (or, at least, employment) of adequate verification methods and tools.

Visualization preferences: Behavior specifications may be aligned with other visualization options. For instance, if all DSML artifacts (such as, language-model definition, language-model constraints, concrete syntax, platform-specific artifacts) are text-based, a textual behavior specification may satisfy user requirements best (O5.2, O5.3). For example, in case of the fUML, UML models can be entirely represented using the action language ALF [112]. ALF acts as a textual surface representation for UML modeling elements that can be used to specify executable behavior.

Table 11: Positive/negative links between drivers and options.

Driver/Option	O5.1	O5.2	O5.3	O5.4	O5.5
Model consistency preservation	+	−	−	+	o
Behavioral definition requirements	o	o	o	o	++
Limited expressiveness	−−	−−	++	o	o
Behavior verification requirements	−	++	−	++	o
Visualization preferences	+/−	+/−	+/−	o	o

Decision consequences.

Semantic variation points: No behavior specification or implicitly defined behavior (O5.5) may introduce semantic variation points. The same applies to under specified informal textual definitions (O5.3). If precise behavior specifications are missing, it is nearly impossible to verify the platform integration (i.e., to prove that the DSML behaves as intended). A missing behavioral guidance may result in multiple (incompatible) possibilities for defining execution semantics while mapping the DSML to a software platform. Semantic variation points affect the consistency (e.g. to ensure intended behavior), traceability (e.g. to be able to reproduce behavior), and transferability (e.g. to enable a mapping to another software platform) of a DSML.

Platform-specific behavior specification: When no explicit behavior is defined (O5.5), a DSML's behavior specification is deferred to the platform integration phase. Runtime semantics (e.g. function calls or if-clauses) can be used to establish a platform-specific behavior specification. Platform-bound execution semantics do not provide a generalized description of a DSML's behavioral intentions (i.e., no explicit behavioral documentation exists). Mapping the DSML to another platform may be cumbersome as DSML behavior needs to be abstracted from the platform-specific developments first. Furthermore, no automatic behavior verification is possible (in contrast to O5.4 for example).

Different behavior enforcement points: Constraining model executions (O5.4) establish two behavior enforcement points: 1) during model execution and 2) after each execution step [32]. In the first case, behavioral conformance is checked at each execution step. The second option relies on an execution trace: after each execution step, the resulting model state is stored. Behavioral properties are then validated against models from this trace. The behavioral enforcement options then depend on the corresponding execution engine. Furthermore, all behavior specification options (O5.1–O5.4) allow to check behavioral aspects before mapping the DSML to a specific platform (e.g. peer-reviewed behavioral model walk-through via informal textual specifications; O5.3). If no behavior is defined (O5.5), the earliest checking point is at the time of platform integration (e.g. reviewing or debugging source code).

Application. As most of our case studies defines a DSML for a narrow domain they do not include explicit behavior specifications (P1, P2, P4–P10). In P3, we employ UML state machines (O5.1) in combination with narrative free-text (O5.3). In related approaches, we identified different options of behavioral specifications. For example, formal textual specifications (mathematical models; O5.2) in P30 and informal textual specifications (narrative semantics descriptions; O5.3) in P53. No approaches constraining models via an execution engine (see, e.g., [32, 99]) qualified for the decision catalog. This may be due to the fact that we did not find any publications targeting executable models prior to 2010. Furthermore, first (beta) versions of related specifications were just recently published in 2010 [112] and 2011 [113], respectively.

Sketch. In P3, a UML state machine (O5.1) with accompanying informal textual descriptions (O5.3) are used to specify states for process-related duties (see Figure 5). For different tasks in a business process, a duty defines an action which must be performed by a certain subject [149]. Among others, P3 extends the UML metamodel with a new `Duty` metaclass for which the state machine in Figure 5 defines possible state changes (e.g. a transition from a passive to a pending state). Additionally in P3, the occurring sequence of steps when entering a `Duty` are listed textually (O5.3).

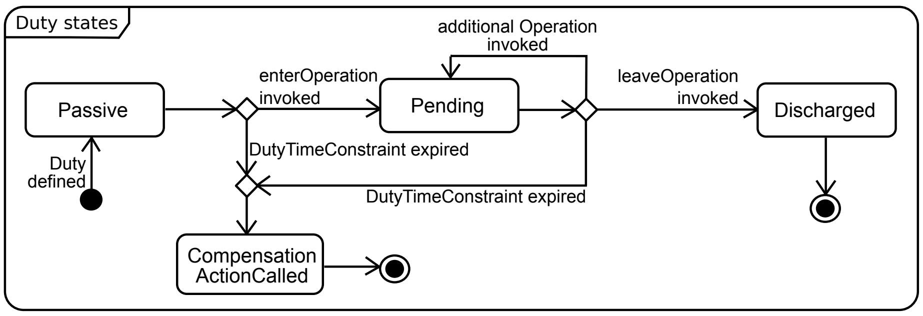

Duty states
additional Operation invoked
enterOperation invoked
leaveOperation invoked
Passive
Pending
Discharged
Duty defined
DutyTimeConstraint expired
DutyTimeConstraint expired
Compensation ActionCalled

Figure 5: Exemplary behavior specification via a UML state machine [137].

4.6 D6 Platform Integration

Problem statement. *How should the DSML artifacts be mapped to (and/or integrated with) a software platform?*
Decision context. Before platform integration, we have defined the DSML's core (i.e., formalized) language model, a set of (additional) structural and behavioral constraints, as well as a concrete syntax specification. At this stage, DSML models (or an executable subset of the models) should be mapped to a software platform (e.g. programming languages, frameworks, components, service applications) and to customized platform artifacts (e.g. source code and execution specifications that are tailored for the respective DSML).

Most often platform integration is achieved via model transformations (see, e.g., [41, 101]) that convert a model into another platform-specific model (also: model-to-model transformation, M2M) or into executable software artifacts (also: model-to-text transformation, M2T; see also A4). Alternatively, DSML models can also be evaluated and executed without intermediate transformations (to be more precise DSML models are then directly transformed into executable machine code via a corresponding DSML interpreter; see also A19).
Decision options.

O6.1 Intermediate model representation: Provide for generating a second and intermediate model (i.e., the target model) based on a DSML model (i.e., the source model) using M2M transformations. This intermediate model can be described via an own metamodel. The source model and target model are separate model entities. From the intermediate model we can create platform-specific artifacts/models (e.g. using M2T transformations). This intermediate structure can be used to optimize the source model (e.g. model canonization and compression) and to attach debugging meta-data (see, e.g., [40]). More specifically, the intermediate model can act as a decorator and/or as an adapter (see, e.g., [54]).

A *decorator model* (e.g. an Eclipse Modeling Framework (EMF) generator model [148]) manages references to the source DSML model and stores meta-data (e.g. code docstrings, prefixes for generated code entities, code package names) which are specific to the platform integration tasks (e.g. code generation). As a result, the domain-specific model data and the transformation-specific model data can be maintained independently from each other.

An *adapter model* does not preserve links back to the source DSML model but replicates the DSML model in a restructured manner. The restructuring aims at facilitating subsequent platform integration tasks (e.g. code generation) by adjusting the model structure (see, e.g., [40]). For example, to overcome certain abstraction mismatches between the DSML model (e.g. graph abstractions in UML activities) and a family of platform-specific artifacts (e.g. block-based process descriptions [100]).

O6.2 Generator template: Create transformation template which turn DSML models into platform-specific execution specifications (e.g. markup documents) and/or source code in the host programming language. Templates access input model data via metamodel-based selections and extraction expressions (e.g. OCL or XPath) and integrate the extracted model data into opaque output strings that represent code fragments. Examples are the Eclipse-based Xpand or the Epsilon Generation Language (EGL).

O6.3 API-based generator: Realize the platform-specific model transformation (e.g., code generation) by instrumenting a programmatic representation of DSML models. The DSML core language model and, thereby, each DSML model (i.e. each instance of the core language model) are internally represented as a collaboration of programmatic entities (e.g. objects). Based on a dedicated API for traversing this internal representation (e.g. a visitor-based API [40] or a mixin-based API [159]), model transformation is achieved by instrumenting this API (e.g. implementing visitors or mixins) to travel the object-based DSML model representation and, for example, to serialize the model data to an output string (see, e.g., [145]). The resulting platform-specific artifacts are independent from the generator language or the generator implementation.

O6.4 (Direct) model execution: Use a (partial) model-execution engine to generate platform-specific instructions directly from DSML models. This requires that the target software platform (and its DSML-specific functions) can be accessed through the same programming language which is used to represent the internal, programmatic DSML model structure (e.g. object-based). Alternatively, inter-language bridges (e.g. wrappers, cross-language reflection) are available to realize such a feature. Given that this internal model representation is accessible through an API (e.g. using visitors [40] or mixins [159]), the internal representation is processed and instrumented to emit platform instructions directly (rather than to generate and to store away instruction statements to be performed at a different point in time). In particular, this options (re)uses and/or extends an existing interpreter or compiler infrastructure for the execution of DSML models.

O6.5 M2M transformation: Perform platform integration via (multiple) endogenous model-to-model (M2M) transformations specified via M2M transformation languages (ATL [19] or ETL [89]). The source and target models share the same metamodel infrastructure on the M3 level (e.g. several refined platform-specific UML profiles). This is in contrast to O6.1 which describes platform-specific model chains not necessarily sharing the same metamodel (e.g. a transformation between a UML-based model and an intermediate Java object model). Target models can either be executed directly (O6.4) or they need further processing, for instance, via subsequent M2T transformations (O6.2, O6.3).

O6.6 None/Not specified: No platform integration is performed; for example, the DSML serves only for documentation purposes, for sketching a software design, or for analyzing requirements.

Combination of options: Template-based (O6.2), generator-driven (O6.3), and model-interpreting (O6.4) platform integration can be combined with intermediate structures (O6.1) to benefit from the advantages of an intermediate representation. In this way, transformation templates can operate on compressed and canonicalized DSML models, generators run against decorator models providing generation-specific meta-data, and a model interpreter finds a prefabricated and execution-oriented model representation (e.g. an unfolded control flow).

In model-driven language workbenches [53], intermediate models (O6.1) can be instantiated from metamodels that are defined via the host language used by the corresponding target platform (e.g. JetBrains MPS/Java). In such a setup, platform integration involves two steps: 1) A M2M transformation turning the DSML model into a programmatic language model; 2) the direct interpretation of the model via the interpreter/compiler infrastructure of the respective host language (see O6.4; e.g. for prototyping and debugging purposes). Additionally, source code artifacts can be generated (O6.2) to keep the code base separated (e.g. for deployment purposes).

Decision drivers. An overview of positive and negative links between decision drivers and available options is shown in Table 12.

Targeting multiple platforms: An intermediate model (O6.1) can act as a common, canonicalizing representation that can be mapped to multiple target platforms which have similar platform-specific abstractions (e.g. a family of process-engine execution specification languages such as BPEL4WS and WS-BPEL). If the constructs of the modeling language differ significantly from their intended platform integrations, an intermediary representation can increase the efficiency of subsequent M2T transformations. For instance, in P7, we transform into an intermediate model first, to bridge between the graph-based PIMs and the block-based PSMs (see also [100]).

Maintainability effort of static code fragments: With an API-based generator (O6.3), the code independent from the DSML model must be integrated with the generator implementation (e.g. a custom visitor). When using generation templates (O6.2), non-changeable and non-parametric code fragments can be clearly separated from generator statements in templates [145]. Depending on the relative amount of static code fragments, an API-based generator involves extra maintenance effort for managing the interwoven fragments of generative code and static code.

Non-executable models: If the DSML should only serve modeling purposes, for example via the definition of a UML profile (O2.2) and the utilization of a standard modeling editor, no explicit platform integration might be needed (O6.6). In this case, the DSML is not meant to be executed on a software platform (see also A16). However, the DSML might primarily serve as a communication instrument between domain experts and software engineers.

Table 12: Positive/negative links between drivers and options.

Driver/Option	O6.1	O6.2	O6.3	O6.4	O6.5	O6.6
Targeting multiple platforms	++	o	o	o	o	o
Maintainability effort of static code fragments	o	+	−	o	o	o
Non-executable models	−−	−−	−−	−−	−−	++

Decision consequences. Depending on which option(s) were chosen, this decision-making step produces a set of output artifacts. Important examples include transformation specifications, test suites (e.g. to test generated code), and platform extensions. The latter are functional additions to the target software platform to cover DSML-specific execution requirements (e.g. through a framework extension or integration of auxiliary frameworks).

Constraint inconsistencies: If the PIM-to-PSM model transformations are performed via multiple M2M transformations generating intermediate model representations (O6.1, O6.5), it has to be ensured that the language-model constraints (O3.1) also hold in the intermediate model(s). Either a second set of explicit constraints (O3.1) must be provided for the intermediate model or constraining M2M transformations (O3.3) must be applied.

Different constraint enforcement points: Code generation templates (O6.2) are applied to instances of the language model. Therefore, constraints enforced on the language model (O3.1) can also be checked for the generator templates (e.g. with EVL, O3.3; see also A21). However, this only prevents wrong usage of the construction rules in the code templates. As no constraints are enforced on the generated code, it may not entirely conform to the constraints defined at the level of the DSML's language model. This means, in contrast to language workbenches, code templates work by generating free-text not conforming to any metamodel.

Application. In P9, we transform Ecore-based language models into Java code via generator templates (O6.2). In P5, no platform integration has been performed (O6.6) because the primary contribution was a non-executable DSML to capture selected security concerns in UML activities. Only later, in P7, the DSML was integrated with the SoaML in an executable manner, with support for generating web Services execution specifications. For this purpose, we employed API-based generators (O6.3) for intermediate models (O6.1) in P7 (a combined option). This is because we had to address certain abstraction mismatches between the DSML model and the platform-specific model.

While not explicitly documented, platform integration and DSML execution (e.g. for testing and simulation purposes) via direct model execution (O6.4) is prepared in several of our projects (e.g. P3, P4, and P8). Based on a model representation and model runtime environment implemented in a DSL toolkit comparable to the one in [159], object-oriented DSML model representations can be created and inspected. For platform integration, these representations could be instrumented for model execution (O6.4).

In related approaches, we find, for instance, intermediate models (O6.1) in P17, generation templates (O6.2) in P17 and P39, API-based templates (O6.3) in P58, M2M transformations (O6.5) in P17, and no documented platform integration (O6.6) in P30, P53, P60, and P61.

Sketch. The following EGL code shows an excerpt from an M2T generation template applied in P9. Here, a Java method is generated for the specification of an audit rule according to the structure of a corresponding metamodel. An audit rule consists of a set of evaluable conditions, whereas the validity of each condition is checked via a generated if-clause. True is returned (see variable `passed`) when all condition checks passed successfully, otherwise the method returns false.

```
[% operation auditRule(auditRule) { %]
  private boolean [%=auditRule.name%]() {
    Map<String, String> data;
    boolean passed = true;
    [% for (signal in auditRule.subscribe) {
      out.println('data = ' + signal.name + '.getData();');
      for (condition in auditRule.conditions) {
        out.println('if (!(' + condition.name + ')) passed = false;');
      }
    } %]
    return passed;
  }
[% } %]
```

5 Associations between Options

A decision option chosen at one decision point may influence options at the same or at subsequent decision points (for example, a choice can favor, determine, or exclude following options). By reviewing our DSML projects and related approaches (see Table 13 in Appendix B) using the decision catalog from Section 4, we identified 21 decision associations within a single decision (Section 5.1) or between two or more decisions (Section 5.2; see also Figure 6). Each association is denoted by a pairing of affected decision options as explained in Section 4. In Figure 6, an association is shown as an edge connecting either two encircled options (e.g. O5.4↔O6.4) or connecting an option and a decision point (shown as a rectangle). An edge between an option and a decision point shows an association between the option and all options of the corresponding decision point (e.g. O1.4↔D2 which is equivalent to O1.4↔O2.1–O2.4).

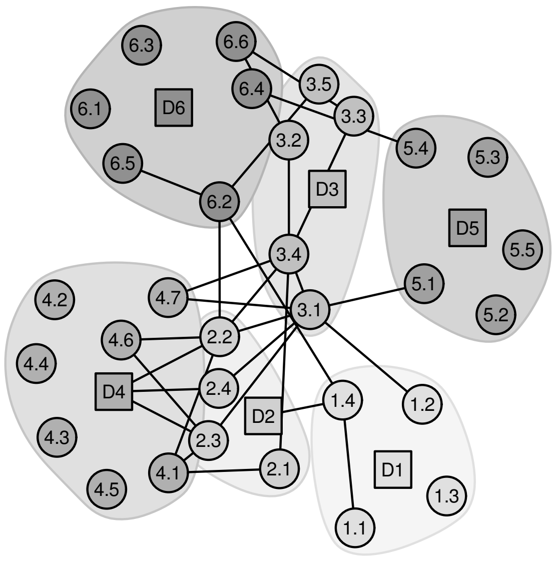


Figure 6: Overview of associations between decision options.

5.1 Associations between Options of one Decision Point

A1 O1.1↔O1.4 Textually accompanied formal diagrammatic models:[8] Diagrammatic models complying to a formal specification (O1.4; e.g., the MOF) may not be sufficient to describe a DSML's language model unambiguously without further explanations. Textual descriptions (O1.1) were found for all of the 90 DSML projects (see Table 13), particularly explaining the semantics of accompanying language models and providing additional information (e.g., intentions behind model and package designs, explanation of model elements, attributes, and associations; see, e.g., P34 [133] or P37 [166]).

A2 O3.1↔O3.4 Textually accompanied constraint-language expressions:[8] Similarly to the former association (A1), constraint-language expressions are also annotated textually (e.g., an OCL statement is explained in natural language, as well; see, e.g., P30 [16] or P40 [32]). This is merely due to increase the readability of constraints as the reader may not be familiar with a certain constraint language (e.g., the OCL). Furthermore, this association emerges also from the fact that not every language-model constraint can formally be described with a constraint language (see next association A3).

A3 O3.4↔O3.1–O3.3 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., [116]). Such constraints have to be provided as text annotations in a natural language. Either these constraints have a documentation purpose only, or they serve for porting the constraints to another environment as they are not locked to a concrete expression form. For example, in P8, language-model constraints are defined via the OCL. However, some constraints need to be expressed in natural language due to a model-level mismatch. Constraints are captured at the language-model level (M2), but some operation calls become manifest at the occurrence level of an activity (M0) only.

A4 O6.2↔O6.5 Model transformation chains:[8] The observed association is characterized by endogenous M2M transformations (O6.5) prior to the code generation step (O6.2; see, e.g., P17 [4] or P32 [5]). In these M2M transformations, source and target models share the same metamodel infrastructure on the M3 level (e.g., the MOF). For example, we found the association being employed for analyzing models (P32) as well as for generating test cases (P17). On the one hand, P32 provides an approach for analyzing OCL-constrained UML class models for inconsistencies via Alloy [80]. A UML class model is transformed into an instance model of the Alloy metamodel (both instantiating the MOF; O6.2). From the Alloy model, a M2T transformation generates a textual representation (O6.5) which serves as input to the Alloy analyzer. Located conflicts can then be traced back to the original model elements in the UML class diagram. On the other hand, P17 uses M2M transformations to generate platform-independent and platform-specific test models (e.g., UML sequence diagrams) from the actual application models (O6.2). Via M2T transformations application code and corresponding test cases are generated (O6.5). In both examples, the Alloy model (P32) and the platform-specific application and test models (P17), all serve as intermediate representations (O6.1) from which textual artifacts are

[8]The association was added or revised based on the findings on *smallest option subset* as well as the option combinations identified for generating prototype option-sets and largest components [141].

created.

5.2 Associations between Options of two or more Decision Points

A5 O1.2↔O3.1 Shared expression foundations: Adopting certain formal textual (e.g. set-theoretical) representations affect the choice of a language (e.g. the OCL) for defining constraints over the core language model explicit and vice versa. If there is a common definitional foundation of both languages, a transformation is facilitated. For example, as basic OCL semantics have been defined in terms of a set-theoretical model (see, e.g, [127]), set theory and set algebras are a natural choice to define a language model at the CIM (computation independent model) level. This underlying correspondence allows for mapping set definitions (e.g. set builders) to equivalent, built-in or custom-defined OCL expressions (e.g. OCL selectors).

A6 O1.4↔D2 Language-model formalization as refinement:[8] If the domain description includes MOF or UML diagrams, a stepwise transition into a UML-based core language model is facilitated. In particular, an association between options O1.4 and O2.2 is a candidate (see, e.g., P16 [83] or P20 [103]). Nevertheless, in some DSML projects found via the SLR, the definition of a MOF-based or modeling-language independent metamodel and the corresponding mapping to a UML profile was not documented explicitly (see, e.g., [44, 153, 163]).[9] This leaves the reader of profile applications, for instance, in the example sections of the papers, with assuming a direct 1:1 mapping of language-model elements into equally named stereotypes—a rather silent naming convention. This lack of explicit documentation is problematic, because it is implicitly assumed that the modeling-language independent metamodel and the UML profile share underlying semantics, which is not necessarily the case. As an example contributing to overcome such impedance mismatches emerging due to diverging definitional foundations of modeling languages, in [91] an approach for the semi-automatic transformation of MOF-based conceptual domain models (O1.4) into UML profiles (O2.2) is presented.

A7 O2.1↔O3.4 Constraint limitations for structural models: An M1 structural model (e.g. a class model) defines a language model at the UML instance level (i.e. at the M1 layer [115]). This means, no metamodel is employed to reflect the domain space and, therefore, domain abstractions can neither be instantiated nor explicitly constrained for their usage as modeling constructs (contradicting the meta-layer architecture of MDD). Thus, restrictions can only be defined in terms of text annotations attached to the language model.

A8 O2.1↔O4.1 Impossible diagram extensions: The decision to define the core language model at the UML M1 level (O2.1) is in conflict with a UML syntax extension (O4.2). In other words, if we use an M1 model to define the DSML's core language model, an extension of the UML's concrete syntax (within the UML framework) is not an option. In this case, model annotations (O4.1) remain the only viable option. However, UML profiles can tailor existing UML metaclasses. In particular, profiles can be used to define dedicated icons which appear as full replacements for the standard notation of stereotyped elements

[9]Please note that projects exhibit formalization and/or critical documentation defects were excluded from the extraction of encoded DSML design decisions.

and which can act as a limited diagrammatic syntax extension.

A9 O2.2↔O3.1∨O3.4 *Constrained UML profiles:*[8] The specification of a UML profile (O2.2) was found accompanied by either formal (O3.1) or informal textual (O3.4) constraint definitions (or both; see, e.g., P70 [160] or P80 [79]). The profile-specific part represents an extension to A2 in Section 5.1 and may indicate a demand for the definition of dedicated constraints besides native UML profile semantics. This can be interpreted as a possible hint that the definitional foundations of particular UML-profile-specific elements, such as `Stereotype` or `Extension`, are not explicit enough to fulfill the requirements for formalizing a DSML language model.

A10 O2.2↔O4.1∧O4.6 *Native stereotype specification:*[8] A UML profile definition (O2.2) for the language-model formalization was observed in combination with a concrete syntax specification via annotating model elements (O4.1) and reusing diagram symbols (O4.6; see, e.g., P22 [122] or P54 [6]). This association has its cause in the stereotype definition of the UML specification: "A Stereotype uses the same notation as a Class, with the addition that the keyword «`stereotype`» is shown before or above the name of the Class" [116]. Hence, a reused symbol (from `Class`; O4.6) is annotated with the keyword «`stereotype`» (O4.1). Please note that this association does not cover icons graphically attached to the model elements extended by the stereotype (O4.2).

Furthermore, by applying the native UML profiling mechanism, the abstract and the concrete syntax of extended model elements are coupled [121]. A stereotype inherits all semantics (abstract syntax) and the notation (concrete syntax) from its extended UML base class. A DSML designer has to choose a base class whose either abstract syntax or concrete syntax most closely matches that of the domain-specific concept. In every case, either the semantics or the notation may not resemble the DSML's application domain. On the one hand, by focusing on concrete-syntax conformance, the stereotyped element may need to be constraint heavily (an increase in complexity). On the other hand, by focusing on abstract-syntax conformance, the addition of keywords and/or icons might not be sufficient or suitable for the intended domain. An approach to overcome this discrepancy is presented in [121]. Therein, the abstract and concrete syntax is decoupled by defining notational extensions via the UML Diagram Interchange specification (DI [109])[10] allowing the modeling of arbitrary notations.

A11 O2.2–O2.4↔D4 Concrete syntax drives UML extension: The formalization strategy for the language model affects the selection of a concrete syntax style. If the language model is defined via a UML profile (O2.2), different presentation options for stereotypes may be considered. A textual presentation (i.e., tags) does not extend the basic UML symbol vocabulary (O4.1, O4.6; see former association A10). Stereotype icons, however, are extensions in the sense of O4.2. For a metamodel extension (O2.3), the definition of new modeling elements (O4.2) is an option. The different combined diagrammatic and non-diagrammatic options are also applicable.

A12 O2.3↔O4.6∧¬O4.1 *Underspecified concrete syntax definition:*[8] Extending the UML metamodel (O2.3) without an explicit concrete syntax definition (O4.6)—even not annotating model elements (O4.1)—was an observed association (see, e.g., P13 [13] or P88 [34]). The authors of these DSMLs silently

[10]Please note that the UML Diagram Definition specification (DD [114]) has replaced the DI specification.

assume that symbols defined for UML metaclasses (in the UML specification [116]) are inherited by the DSML-specific extensions (e.g., via a generalization relationship). This is in contrast to the practice applied in the UML specification itself (see O4.6 in Section 4.4).

A13 O3.1↔O2.2–O2.4 Constraint inconsistencies: A combination of different language-model formalizations (e.g. a UML profile and a metamodel extension) may require the duplication and modification of constraint definitions. For instance, in P7 [67, 72] we define both, a UML metamodel extension and a profile definition to integrate with the SoaML specification [110]. Hence, we define constraint-language expressions as OCL invariants over both language model formalizations. Thus, both constraint definitions need to be maintained and held consistent.

A14 O3.1∧O3.4↔O4.7 Tailoring semantics only:[8] Customizing the UML or any extensions of it (e.g., SoaML, SysML [111]) via explicit constraint expressions (O3.1, O3.4) without a concrete syntax definition (O4.7) to specify a DSML was an observed association (see, e.g., P40 [32] or P84 [124]). This association bears the risk that while the formal semantics of DSML elements may be well-defined, they cannot be distinguished from non-constrained UML elements (see also A12 and A17). The DSML should only be used in isolation, not mixing concrete syntaxes of tailored and UML model elements. The problem of ambiguity exists also for extensions of DSMLs (e.g., a revision of a previously defined DSML). Whether new features are added or formerly defined semantics are changed, a unique identifier (e.g., version number, modified name) should be used to distinguish between different releases of a DSML.

A15 O3.2↔O6.6 Specific host/platform language: If code annotations were used to express constraints over the core language model, a runtime environment to execute the code statements would be needed, for instance, as part of the platform integration step. As an example, consider Java expressions attached to an extended UML metamodel. In such a case, a JVM is needed to evaluate these Java expressions and execute them on the system-level.

A16 O3.3↔O6.6 Mandatory platform integration: Whether constraining M2M or M2T transformations are actually an option for defining language-model constraints depends directly on the decision if we want to perform platform integration or not. Likewise, if the use of constraining M2M/M2T transformations is a mandatory requirement known up front (e.g. due to the toolkit choice or in a legacy system scenario), integrating language-model constraints into the transformation template suite avoids duplicated specification effort as well as redundant model-level artifacts (e.g. OCL constraints plus corresponding template expressions). However, the specialized constraint languages coming with M2M/M2T generation languages (e.g. EVL for EGL) are commonly restricted in their constraint-expressing power compared to model-level constraint languages (e.g. an equivalent to OCL's message introspection might be missing). Besides, integrating constraint-checking and generation-specific template expressions can hinder a separation of concerns by including expressions which are irrelevant for the actual generation task. In particular, this may cause overly complex or even conflicting template code. These pitfalls can be avoided when applying the constraint-checking M2M templates in a transformation of the DSML into an intermediate model representation (O6.1), with the actual platform integration step (M2T code generation) being performed on the validated intermediate model.

A17 O4.6↔O2.2 Symbol ambiguity in diagrams: When reusing existing UML symbols, the resulting "extended" diagrams are most often ambiguous. In particular, using the same symbol for different concepts means that refining concepts cannot be distinguished from the refined ones. To introduce a simplistic discriminator without creating new symbols, one can provide a UML profile to define a series of stereotype tags which can then be attached to the reused symbols to denote the DSML-specific refinements. In this case, UML profiles serve primarily for clarifying the concrete syntax elements used for a DSML. This resembles the usage of standard profiles as defined by the UML [116], however, without adding to the abstract syntax and semantics of the language model. In P7 [67, 72], for example, we do not define a dedicated concrete syntax (that is, no diagrammatic extension) to a newly defined metamodel element called `SecureInterface`. It is only distinguishable from a pre-existing `ServiceInterface` via its profile mapping and stereotyped representation as «`SecureInterface`».

A18 O5.1↔O3.1 M1 behavioral models as constraints: M1 models can be attached to metamodel elements for behavioral specifications (e.g. via the `ownedBehavior` relation of a `BehavioredClassifier` [116]). In doing so, they are constraining/defining the behavior of metamodel elements. For example, in P3 [137, 138] we make use of a UML state machine to define states (e.g. passive, pending, discharged) and transition options between those states for DSML elements.

A19 O5.4↔O6.4 Integrated model execution: Interpreting UML models directly (e.g. via an execution engine) demands a precise specification of 1) DSML structural properties, 2) well-formed instance models, 3) execution semantics, and 4) operational semantics. Regarding (1), structural properties of a UML/MOF-based DSML (domain elements, relationships etc.) are represented via a formalized language model (i.e., the abstract syntax of a DSML; D2). An executable model must adhere to certain well-formedness criteria (2). For a DSML, these are specified via language-model constraints (D3) and enforced at the level of the instance model. Execution semantics (3) of a DSML are defined via behavior specifications (D5) to be processed via a certain platform (D6). The definition of behavioral constraints (O5.4) and the model execution (O6.4) may be supported by the same model execution environment. Operational semantics (4) at the level of the executing environment (i.e. interpretation of the platform-specific implementation as a sequence of computational steps) are specific to an execution engine and do not need to be defined specifically via the DSML.

A20 O6.2↔O1.4∧O2.2 Existing toolchain support:[8] Tools for editing UML models, including the definition and application of profiles (O2.2), are nowadays frequently available (e.g., MagicDraw, Eclipse Papyrus, Rational Software Architect, Enterprise Architect, Modelio, UModel). In addition, template-based M2T transformations (O6.2) are a widely supported platform integration technique in contemporary MDD tool chains, and a variety of template language implementations exist, such as, Eclipse Xpand, Xtend2, EGL, JET, or Acceleo (see, e.g., [41, 130]). Several UML model editors provide combined tool support for M2T transformations in an MDD-based way, as well (e.g., in the Eclipse-verse based on EMF-compliant models). Thus, the observed association is characterized by a high availability of modeling tools and generator engines (see, e.g., P12 [3] or P67 [128]). Nevertheless, a formal diagrammatic model not compliant with the UML specification (e.g., an ER model; O1.4) must be mapped to native UML constructs first (i.e. a profile definition) to benefit from standard tool support. Alternatively, the EMF-based technical projection of the EMOF [115] (i.e. an Ecore model; O1.4) is also a candidate option to facilitate toolchain support as automatic transformations into and from UML class models exist. Furthermore, a partially tool-supported approach for the semi-automatic transformation of MOF-based models into UML profiles is presented in [91] (see also A6).

A21 O6.2↔O3.5 Platform-specific constraint enforcement:[8] This observed association is characterized by a late and platform-specific constraint enforcement point. Corresponding DSMLs do not define constraints for the language model explicitly (O3.5), but integrate them into (templates of) code generators (see, e.g., P51 [73] or P85 [86]). As generation templates (O6.2) are applied to instances of the language model, constraints can basically be enforced (see also A16). However, constraints are checked late in the DSML development process; i.e. at the time of executing M2T transformations. Until platform integration is performed, the conformance of models to their corresponding constraints is not validated. Furthermore, constraints need to be duplicated for different generator engines and for the support of multiple platforms. In addition, a DSML designer has to keep in mind that—independent of an existing or lacking definition of language model constraints—no constraints are enforced on the generated code (i.e. the output of a M2T transformation is not interpreted by its generator component).

References

[1] B. Agreiter and R. Breu. Model-driven configuration of SELinux policies. In *On the Move to Meaningful Internet Syst.*, volume 5871 of *LNCS*, pages 887–904. Springer, 2009.

[2] M. Alam, R. Breu, and M. Hafner. Model-driven security engineering for trust management in SECTET. *J. Softw.*, 2(1):47–59, 2007.

[3] S. Ali, L. C. Briand, and H. Hemmati. Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. *Softw. Syst. Model.*, 11(4):633–670, 2012.

[4] E. L. Alves, P. D. Machado, and F. Ramalho. Automatic generation of built-in contract test drivers. *Softw. Syst. Model.*, pages 1–25, 2012.

[5] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Softw. Syst. Model.*, 9(1):69–86, 2010.

[6] F. Aoussat, M. Oussalah, and M. Nacer. SPEM extension with software process architectural concepts. In *Proc. 35th Annu. IEEE Int. Conf. Comp. Softw. and Appl.*, pages 215–223, 2011.

[7] AT&T Research. Graphviz – graph visualization software. Available at: `http://www.graphviz.org`, 2013.

[8] P. Baker, P. Bristow, C. Jervis, D. King, R. Thomson, B. Mitchell, and S. Burton. Detecting and resolving semantic pathologies in UML sequence diagrams. In *Proc. 10th Europ. Softw. Eng. Conf. and the 13th ACM SIGSOFT Int. Sym. Found. of Softw. Eng.*, pages 50–59. ACM, 2005.

[9] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based modeling and refinement of service-oriented architectures. *Softw. Syst. Model.*, 5(2): 187–207, 2006.

[10] R. Barrett, L. M. Patcas, C. Pahl, and J. Murphy. Model driven distribution pattern design for dynamic web service compositions. In *Proc. 6th Int. Conf. Web Eng.*, pages 129–136. ACM, 2006.

[11] C. Bartolini, A. Bertolino, G. De Angelis, and G. Lipari. A UML profile and a methodology for real-time systems design. In *Proc. 32nd EUROMICRO Conf. Softw. Eng. and Adv. Appl.*, pages 108–117, 2006.

[12] R. Behjati, T. Yue, S. Nejati, L. Briand, and B. Selic. Extending SysML with AADL concepts for comprehensive system architecture modeling. In *Proc. 7th Europ. Conf. Model. Found. Appl.*, volume 6698 of *LNCS*, pages 236–252. Springer, 2011.

[13] R. Bendraou, M.-P. Gervais, and X. Blanc. UML4SPM: A UML2.0-based metamodel for software process modelling. In *Proc. 8th Int. Conf. Model Driven Eng. Lang. Syst.*, volume 3713 of *LNCS*, pages 17–38. Springer, 2005.

[14] R. Bendraou, P. Desfray, M.-P. Gervais, and A. Muller. MDA tool components: A proposal for packaging know-how in model driven development. *Softw. Syst. Model.*, 7(3):329–343, 2008.

[15] J. Bergh and K. Coninx. CUP 2.0: High-level modeling of context-sensitive interactive applications. In *Model Driven Eng. Lang. Syst.*, volume 4199 of *LNCS*, pages 140–154. Springer, 2006.

[16] K. Berkenkötter and U. Hannemann. Modeling the railway control domain rigorously with a UML 2.0 profile. In *Proc. 25th Int. Conf. Comput. Safety, Reliab., Secur.*, volume 4166 of *LNCS*, pages 398–411. Springer, 2006.

[17] S. Bernardi, F. Flammini, S. Marrone, J. Merseguer, C. Papa, and V. Vittorini. Model-driven availability evaluation of railway control systems. In *Proc. 30th Int. Conf. Comput. Safety, Reliab., Secur.*, volume 6894 of *LNCS*, pages 15–28. Springer, 2011.

[18] S. Bernardi, J. Merseguer, and D. C. Petriu. A dependability profile within MARTE. *Softw. Syst. Model.*, 10(3):313–336, 2011.

[19] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *Proc. 2nd OOPSLA Worksh. Genera. Tech. in the Context of Model Driven Archit.*, 2003.

[20] L. M. Bibbo, D. García, and C. Pons. A domain specific language for the development of collaborative systems. In *Proc. Int. Conf. Chilean Comp. Sci. Soc.*, pages 3–12. IEEE, 2008.

[21] C. Bock. UML 2 activity and action models. *J. Object Technol.*, 2(4): 43–53, July/August 2003.

[22] C. Borgelt. Frequent item set mining. *Wiley Int. Rev. Data Min. and Knowl. Disc.*, 2(6):437–456, Nov. 2012.

[23] M. Bošković and W. Hasselbring. Model driven performance measurement and assessment with MoDePeMART. In *Model Driven Eng. Lang. Syst.*, volume 5795 of *LNCS*, pages 62–76. Springer, 2009.

[24] A. Braganca and R. J. Machado. Extending UML 2.0 metamodel for complementary usages of the «extend» relationship within use case variability specification. In *Proc. 10th Int. Conf. Softw. Product Line*, pages 123–130. IEEE, 2006.

[25] S. Brahe and K. Østerbye. Business process modeling: Defining domain specific modeling languages by use of UML profiles. In *Model Driven Archit. – Found. and Appl.*, volume 4066 of *LNCS*, pages 241–255. Springer, 2006.

[26] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *J. Syst. Softw.*, 80(4):571–583, 2007.

[27] P. Brosch, M. Seidl, M. Wimmer, and G. Kappel. Conflict visualization for evolving UML models. *J. Object Technol.*, 11(3):2:1–30, Oct. 2012.

[28] J. Bruck and K. Hussey. 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`. Last accessed: Sep 9, 2015, 2008. IBM.

[29] J. E. Burge, J. M. Carroll, R. McCall, and I. Mistrík. *Rationale-Based Software Engineering*. Springer, 2008.

[30] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-oriented Software Architecture – On Patterns and Pattern Languages*. John Wiley & Sons, 2007.

[31] P. Cáceres, V. de Castro, J. M. Vara, and E. Marcos. Model transformations for hypertext modeling on web information systems. In *Proc. 21st Annu. ACM Sym. Applied Comput.*, pages 1232–1239. ACM, 2006.

[32] E. Cariou, C. Ballagny, A. Feugas, and F. Barbier. Contracts for model execution verification. In *Model. Found. Appl.*, volume 6698 of *LNCS*, pages 3–18. Springer, 2011.

[33] R. Chaves. TextUML toolkit. Available at: `http://abstratt.com/textuml/`, 2013.

[34] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *J. Object Technol.*, 6(9):165–185, Oct. 2007.

[35] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Sci. Comput. Program.*, 76(12):1130–1143, 2011.

[36] J. Coplien. *Software Patterns*. SIGS Manag. Briefings. SIGS Books & Multimedia, 1996.

[37] D. Costal, C. Gómez, A. Queralt, R. Raventós, and E. Teniente. Improving the definition of general constraints in UML. *Softw. Syst. Model.*, 7(4): 469–486, 2008.

[38] Z. Cui, L. Wang, X. Li, and D. Xu. Modeling and integrating aspects with UML activity diagrams. In *Proc. 24th Annu. ACM Sym. Applied Comput.*, pages 430–437. ACM, 2009.

[39] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[40] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proc. OOPSLA Worksh. Genera. Tech. in the Context of Model-Driven Archit.*, 2003.

[41] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[42] A. M. R. da Cruz and J. a. P. Faria. A metamodel-based approach for automatic user interface generation. In *Proc. 13th Int. Conf. Model Driven Eng. Lang. Syst.*, volume 6394 of *LNCS*, pages 256–270. Springer, 2010.

[43] W. Dahman and J. Grabowski. UML-based specification and generation of executable web services. In *Syst. Anal. and Model.: About Models*, volume 6598 of *LNCS*, pages 91–107. Springer, 2011.

[44] E. Damiani, A. Colombo, F. Frati, and C. Bellettini. A metamodel for modeling and measuring scrum development process. In *Agile Processes in Softw. Eng. and Extreme Prog.*, volume 4536 of *LNCS*, pages 74–83. Springer, 2007.

[45] S. Diaw, R. Lbath, and B. Coulette. Specification and implementation of SPEM4MDE, a metamodel for MDE software processes. In *Proc. 23rd Int. Conf. Softw. Eng. & Knowl. Eng.*, pages 646–653, 2011.

[46] A. H. Dutoit, R. McCall, I. Mistrík, and B. Paech. Rationale management in software engineering: Concepts and techniques. In *Rationale Manag. in Softw. Eng.*, chapter 1, pages 1–48. Springer, 2006.

[47] R. Ellner, S. Al-Hilank, J. Drexler, M. Jung, D. Kips, and M. Philippsen. eSPEM – a SPEM extension for enactable behavior modeling. In *Proc. 6th Europ. Conf. Model. Found. Appl.*, volume 6138 of *LNCS*, pages 116–131. Springer, 2010.

[48] V. Ermagan and I. H. Krüger. A UML2 profile for service modeling. In *Proc. 10th Int. Conf. Model-Driven Eng. Lang. Syst.*, volume 4735 of *LNCS*, pages 360–374. Springer, 2007.

[49] E. Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, first edition, 2004.

[50] J. Evermann. A meta-level specification and profile for AspectJ in UML. *J. Object Technol.*, 6(7):27–49, Aug. 2007.

[51] E. Filtz. Systematic literature review and evaluation of DSML-design decisions. Bachelor thesis, WU Vienna, 2013.

[52] D. Flater, P. A. Martin, and M. L. Crane. Rendering UML activity diagrams as human-readable text. In *Proc. Int. Conf. Inform. and Knowl. Eng.*, pages 207–213, 2009.

[53] M. Fowler. Language workbenches: The killer-app for domain specific languages? Available at: `http://martinfowler.com/articles/languageWorkbench.html`, 2005. Last accessed: Feb 2, 2015.

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[55] V. Garousi, L. C. Briand, and Y. Labiche. Control flow analysis of UML 2.0 sequence diagrams. In *Model Driven Archit. – Found. and Appl.*, volume 3748 of *LNCS*, pages 160–174. Springer, 2005.

[56] S. Gilmore, L. Gönczy, N. Koch, P. Mayer, M. Tribastone, and D. Varró. Non-functional properties in the model-driven development of service-oriented systems. *Softw. Syst. Model.*, 10(3):287–311, 2011.

[57] V. Grassi. Performance analysis of mobile systems. In *Formal Methods for Mobile Comput.*, volume 3465 of *LNCS*, pages 107–154. Springer, 2005.

[58] R. Grønmo, M. l. Jaeger, and H. Hoff. Transformations between UML and OWL-S. In *Model Driven Archit. – Found. and Appl.*, volume 3748 of *LNCS*, pages 269–283. Springer, 2005.

[59] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Text-based modeling. In *Proc. 4th Int. Worksh. Softw. Lang. Eng.*, 2007.

[60] M. Hafner and R. Breu. Realizing model driven security for inter-organizational workflows with WS-CDL and UML 2.0. In *Proc. 8th Int. Conf. Model Driven Eng. Lang. Syst.*, volume 3713 of *LNCS*, pages 39–53. Springer, 2005.

[61] M. Hahsler, B. Grün, and K. Hornik. arules—a computational environment for mining association rules and frequent item sets. *J. Stat. Softw.*, 14(15):1–25, 2005.

[62] T. Harris. yUML. Available at: `http://yuml.me`, 2013.

[63] D. Hatebur and M. Heisel. A UML profile for requirements analysis of dependable software. In *Proc. 29th Int. Conf. Comput. Safety, Reliab., Secur.*, volume 6351 of *LNCS*, pages 317–331. Springer, 2010.

[64] D. Hatebur and M. Heisel. Making pattern- and model-based software development more rigorous. In *Formal Methods and Softw. Eng.*, volume 6447 of *LNCS*, pages 253–269. Springer, 2010.

[65] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Model Driven Archit. – Found. and Appl.*, volume 5562 of *LNCS*, pages 114–129. Springer, 2009.

[66] B. Hofreiter. Extending UN/CEFACT's modeling methodology by a UML profile for local choreographies. *Inform. Syst. and e-Bus. Manag.*, 7(2): 251–271, 2009.

[67] B. Hoisl and S. Sobernig. Integrity and confidentiality annotations for service interfaces in SoaML models. In *Proc. Int. Worksh. Secur. Aspects of Process-aware Inform. Syst.*, pages 673–679. IEEE, 2011.

[68] B. Hoisl and M. Strembeck. Modeling support for confidentiality and integrity of object flows in activity models. In *Proc. 14th Int. Conf. Bus. Inform. Syst.*, volume 97 of *LNBIP*, pages 278–289. Springer, 2011.

[69] B. Hoisl and M. Strembeck. A UML extension for the model-driven specification of audit rules. In *Proc. 2nd Int. Worksh. Inform. Syst. Secur. Eng.*, volume 112 of *LNBIP*, pages 16–30. Springer, 2012.

[70] B. Hoisl, S. Sobernig, S. Schefer-Wenzl, M. Strembeck, and A. Baumgrass. Design decisions for UML and MOF based domain-specific language models: Some lessons learned. In *Proc. 2nd Worksh. Process-based Appr. for Model-Driven Eng.*, pages 303–314, 2012.

[71] B. Hoisl, S. Sobernig, S. Schefer-Wenzl, M. Strembeck, and A. Baumgrass. A catalog of reusable design decisions for developing UML- and MOF-based domain-specific modeling languages. Available at: http://epub.wu.ac.at/3578/, 2012. Tech. Rep. 2012/01, WU Vienna.

[72] B. Hoisl, S. Sobernig, and M. Strembeck. Modeling and enforcing secure object flows in process-driven SOAs: An integrated model-driven approach. *Softw. Syst. Model.*, 13(2):513–548, 2014.

[73] I.-C. Hsu. Extending UML to model web 2.0-based context-aware applications. *Softw. Pract. Exper.*, 42(10):1211–1227, 2012.

[74] B. C. Hungerford, A. R. Hevner, and R. W. Collins. Reviewing software diagrams: A cognitive study. *IEEE T. Softw. Eng.*, 30:82–96, 2004.

[75] International Organization for Standardization. Information technology – syntactic metalanguage – extended BNF (ISO/IEC 14977). Available at: http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip, 1996.

[76] International Organization for Standardization. Information technology – Z formal specification notation – syntax, type system and semantics. Available at: http://www.iso.org/iso/catalogue_detail?csnumber=21573, 2002. ISO/IEC 13568:2002.

[77] International Organization for Standardization. Information technology – Z formal specification notation – syntax, type system and semantics – technical corrigendum 1. Available at: http://www.iso.org/iso/catalogue_detail?csnumber=46112, 2007. ISO/IEC 13568:2002/Cor 1:2007.

[78] M. Z. Iqbal, A. Arcuri, and L. Briand. Environment modeling with UML/MARTE to support black-box system testing for real-time embedded systems: Methodology and industrial case studies. In *Proc. 13th Int. Conf. Model Driven Eng. Lang. Syst.*, volume 6394 of *LNCS*, pages 286–300. Springer, 2010.

[79] I. Ivkovic and K. Kontogiannis. A framework for software architecture refactoring using model transformations and semantic annotations. In *Proc. 10th Europ. Conf. Softw. Maint. and ReEng.*, pages 135–144, 2006.

[80] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

[81] E. Jackson and J. Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Softw. Syst. Model.*, 8(4):451–478, 2009.

[82] V. Jain, A. Kumar, and P. R. Panda. A SysML profile for development and early validation of TLM 2.0 models. In *Proc. 7th Europ. Conf. Model. Found. Appl.*, volume 6698 of *LNCS*, pages 299–311. Springer, 2011.

[83] S. K. Johnson and A. W. Brown. A model-driven development approach to creating service-oriented solutions. In *Service-Oriented Comput.*, volume 4294 of *LNCS*, pages 624–636. Springer, 2006.

[84] J. U. Júnior, R. D. Penteado, and V. V. de Camargo. An overview and an empirical evaluation of UML-AOF: An UML profile for aspect-oriented frameworks. In *Proc. 25th Annu. ACM Sym. Applied Comput.*, pages 2289–2296. ACM, 2010.

[85] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[86] G. M. Kapitsaki, D. A. Kateros, G. N. Prezerakos, and I. S. Venieris. Model-driven development of composite context-aware web applications. *Inform. Softw. Tech.*, 51(8):1244–1260, 2009.

[87] B. Kitchenham, O. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering: A systematic literature review. *Inform. Softw. Tech.*, 51(1):7–15, 2009.

[88] N. Koch, G. Zhang, and M. J. Escalona. Model transformations from requirements to web system design. In *Proc. 6th Int. Conf. Web Eng.*, pages 281–288. ACM, 2006.

[89] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige. The Epsilon book. Available at: `http://www.eclipse.org/epsilon/doc/book/`, 2013.

[90] J. Kryštof. An LPGM method: Platform independent modeling and development of graphical user interface. *Informatica (Ljubljana)*, 34(3):353–367, 2010.

[91] F. Lagarde, H. Espinoza, F. Terrier, and S. Gérard. Improving UML profile design practices by leveraging conceptual domain models. In *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pages 445–448. ACM, 2007.

[92] F. Lagarde, F. Terrier, C. André, and S. Gérard. Constraints modeling for (profiled) UML models. In *Model Driven Archit. – Found. and Appl.*, volume 4530 of *LNCS*, pages 130–143. Springer, 2007.

[93] B. P. Lamancha, M. Polo, D. Caivano, M. Piattini, and G. Visaggio. Automated generation of test oracles using a model-driven approach. *Inform. Softw. Tech.*, 55(2):301–319, 2013.

[94] A. v. Lamsweerde. Formal specification: A roadmap. In *Proc. 22nd Int. Conf. Softw. Eng. – Future of Softw. Eng. Track*, pages 147–159. ACM, 2000.

[95] P. Liegl, C. Huemer, and C. Pichler. Registry support for core component-based business document models. *Service Oriented Comput. and Appl.*, 5 (3):183–202, 2011.

[96] K. Lind and R. Heldal. A model-based and automated approach to size estimation of embedded software components. In *Proc. 14th Int. Conf. Model Driven Eng. Lang. Syst.*, pages 334–348. Springer, 2011.

[97] B. List and B. Korherr. An evaluation of conceptual business process modelling languages. In *Proc. ACM Sym. Applied Comput.*, pages 1532–1539. ACM, 2006.

[98] P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-driven service orchestration. In *Proc. 12th Int. Conf. Enterp. Distrib. Object Comput.*, pages 203–212. IEEE, 2008.

[99] T. Mayerhofer, P. Langer, and M. Wimmer. Towards xMOF: Executable DSMLs based on fUML. In *Proc. 12th Worksh. Domain-Specific Model.*, pages 1–6. ACM, 2012.

[100] J. Mendling, K. B. Lassen, and U. Zdun. On the transformation of control flow between block-oriented and graph-oriented process modeling languages. *Int. J. Bus. Process Integration and Manag.*, 3(2):96–108, 2008.

[101] T. Mens and P. v. Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[102] M. Miguel, J. Briones, J. Silva, and A. Alonso. Integration of safety analysis in model-driven software development. *IET Softw.*, 2(3):260–280, 2008.

[103] H. G. Min and S. D. Kim. A technique to represent and generate components in MDA/PIM for automation. In *Fund. Appr. to Softw. Eng.*, volume 3922 of *LNCS*, pages 293–307. Springer, 2006.

[104] D. Moody and J. van Hillegersberg. Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams. In *Softw. Lang. Eng.*, volume 5452 of *LNCS*, pages 16–34. Springer, 2009.

[105] N. Moreno, P. Fraternali, and A. Vallecillo. WebML modelling in UML. *IET Softw.*, 1(3):67–80, 2007.

[106] D. Mouheb, C. Talhi, A. Mourad, V. Lima, M. Debbabi, L. Wang, and M. Pourzandi. An aspect-oriented approach for software security hardening: From design to implementation. In *Proc. 8th Conf. New Trends in Softw. Methodol., Tools and Tech.*, pages 203–222. IOS Press, 2009.

[107] M. Nikolaidou, N. Alexopoulou, A. Tsadimas, A. Dais, and D. Anagnostopoulos. Accommodating EIS UML 2.0 profile using a standard UML modeling tool. In *Proc. Int. Conf. Softw. Eng. Adv.*, 2007.

[108] Object Management Group. Human-usable textual notation (HUTN) specification. Available at: `http://www.omg.org/spec/HUTN`, Aug. 2004. Version 1.0, formal/2004-08-01.

[109] Object Management Group. Diagram interchange. Available at: `http://www.omg.org/spec/UMLDI/`, Apr. 2006. Version 1.0, formal/06-04-04.

[110] Object Management Group. Service oriented architecture modeling language (SoaML) specification. Available at: `http://www.omg.org/spec/SoaML`, May 2012. Version 1.0.1, formal/2012-05-10.

[111] Object Management Group. OMG systems modeling language (OMG SysML). Available at: `http://www.omg.org/spec/SysML`, June 2012. Version 1.3, formal/2012-06-01.

[112] Object Management Group. Action language for foundational UML (ALF): Concrete syntax for a UML action language. Available at: `http://www.omg.org/spec/ALF`, Oct. 2013. Version 1.0.1, formal/2013-09-01.

[113] Object Management Group. Semantics of a foundational subset for executable UML models (fUML). Available at: `http://www.omg.org/spec/FUML`, Aug. 2013. Version 1.1, formal/2013-08-06.

[114] Object Management Group. Diagram definition (DD). Available at: `http://www.omg.org/spec/DD/`, June 2015. Version 1.1, formal/2015-06-01.

[115] Object Management Group. OMG meta object facility (MOF) core specification. Available at: `http://www.omg.org/spec/MOF`, June 2015. Version 2.5, formal/2015-06-05.

[116] Object Management Group. OMG unified modeling language (OMG UML). Available at: `http://www.omg.org/spec/UML`, June 2015. Version 2.5, formal/2015-03-01.

[117] Object Management Group. XML metadata interchange (XMI) specification. Available at: `http://www.omg.org/spec/XMI`, June 2015. Version 2.5.1, formal/2015-06-07.

[118] Object Management Group (OMG). Object constraint language. Available at: `http://www.omg.org/spec/OCL`, Feb. 2014. Version 2.4, formal/2014-02-03.

[119] R. Panesar-Walawege, M. Sabetzadeh, and L. Briand. A model-driven engineering approach to support the verification of compliance to safety standards. In *Proc. 22nd Int. Sym. Softw. Reliab. Eng.*, pages 30–39, 2011.

[120] R. K. Panesar-Walawege, M. Sabetzadeh, and L. Briand. Using UML profiles for sector-specific tailoring of safety evidence information. In *Proc. 30th Int. Conf. Conceptual Model.*, pages 362–378. Springer, 2011.

[121] J. Pardillo and C. Cachero. Domain-specific language modelling with UML profiles by decoupling abstract and concrete syntaxes. *J. Syst. Softw.*, 83 (12):2591–2606, 2010.

[122] J. E. Pérez-Martínez and A. Sierra-Alonso. From analysis model to software architecture: A PIM2PIM mapping. In *Model Driven Archit. – Found. and Appl.*, volume 4066 of *LNCS*, pages 25–39. Springer, 2006.

[123] G. Prezerakos, N. Tselikas, and G. Cortese. Model-driven composition of context-aware web services using ContextUML and aspects. In *Proc. IEEE Int. Conf. Web Services*, pages 320–329. IEEE, July 2007.

[124] A. Queralt and E. Teniente. A platform independent model for the electronic marketplace domain. *Softw. Syst. Model.*, 7(2):219–235, 2008.

[125] I. Reinhartz-Berger and A. Sturm. Utilizing domain models for application design and validation. *Inform. Softw. Tech.*, 51(8):1275–1289, 2009.

[126] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A UML 2.0 profile for SystemC: Toward high-level SoC design. In *Proc. 5th Int. Conf. Embedded Softw.*, pages 138–141. ACM, 2005.

[127] M. Richters and M. Gogolla. OCL: Syntax, semantics, and tools. In *Object Model. with the OCL*, volume 2263 of *LNCS*, pages 447–450. Springer, 2002.

[128] G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel. Reliability prediction in model-driven development. In *Model Driven Eng. Lang. Syst.*, volume 3713 of *LNCS*, pages 339–354. Springer, 2005.

[129] J. Romero and A. Vallecillo. Modeling the ODP computational viewpoint with UML 2.0. In *Proc. 9th Int. EDOC Enterp. Comput. Conf.*, pages 169–180, 2005.

[130] L. M. Rose, N. Matragkas, D. S. Kolovos, and R. F. Paige. A feature model for model-to-text transformation languages. In *Proc. 4th Int. Worksh. Model. in Softw. Eng.*, pages 57–63. IEEE, 2012.

[131] N. Russell, W. M. P. Aalst, A. H. M. T. Hofstede, and P. Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Prof. of the 3rd Asia-Pacific Conf. Conceptual Model.*, volume 53 of *Conf. in Research and Practice in Inform. Technol.*, pages 95–104, 2006.

[132] M. Saleem, J. Jaafar, and M. Hassan. Secure business process modelling of SOA applications using "UML-SOA-Sec". *Int. J. Innov. Comput. I.*, 8 (4):2729–2746, 2012.

[133] P. Salehi, A. Hamoud-Lhadj, P. Colombo, F. Khendek, and M. Toeroe. A UML-based domain specific modeling language for the availability management framework. In *Proc. 12th Int. Sym. High-Assur. Syst. Eng.*, pages 35–44, 2010.

[134] T. Schattkowsky, J. Hausmann, and G. Engels. Using UML activities for system-on-chip design and synthesis. In *Model Driven Eng. Lang. Syst.*, volume 4199 of *LNCS*, pages 737–752. Springer, 2006.

[135] S. Schefer. Consistency checks for duties in extended UML2 activity models. In *Proc. Int. Worksh. Secur. Aspects of Process-aware Inform. Syst.*, pages 680–685. IEEE, 2011.

[136] S. Schefer and M. Strembeck. Modeling support for delegating roles, tasks, and duties in a process-related rbac context. In *Proc. Int. Worksh. Inform. Syst. Secur. Eng.*, volume 83 of *LNBIP*, pages 660–667. Springer, 2011.

[137] S. Schefer-Wenzl and M. Strembeck. Modeling process-related duties with extended UML activity and interaction diagrams. *Electron. Commun. EASST*, 37, 2011.

[138] S. Schefer-Wenzl and M. Strembeck. An approach for consistent delegation in process-aware information systems. In *Proc. 15th Int. Conf. Bus. Inform. Syst.*, volume 117 of *LNBIP*, pages 60–71. Springer, 2012.

[139] S. Schefer-Wenzl and M. Strembeck. Modeling context-aware RBAC models for business processes in ubiquitous computing environments. In *Proc. 3rd Int. Conf. Mobile, Ubiquitous and Intell. Comput.*, pages 126–131. IEEE, 2012.

[140] K. Siau and Y. Tian. A semiotic analysis of unified modeling language graphical notations. *Requir. Eng.*, 14(1):15–26, 2009.

[141] S. Sobernig, B. Hoisl, and M. Strembeck. Protocol for a systematic literature review on design decisions for UML-based DSMLs. Available at: `http://epub.wu.ac.at/4311/`, 2014. Tech. Rep., WU Vienna, 2014/02.

[142] E. Soler, J. Trujillo, E. Fernández-Medina, and M. Piattini. Building a secure star schema in data warehouses by an extension of the relational package from CWM. *Comp. Stand. Inter.*, 30(6):341–350, 2008.

[143] S. S. Somé. A meta-model for textual use case description. *J. Object Technol.*, 8(7):87–106, 2009.

[144] C. Song, E. Cho, and C. Kim. An integrated GUI-business component modeling method for the MDD- and MVC-based hierarchical designs. *Int. J. Softw. Eng. Know.*, 21(3):447–490, 2011.

[145] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[146] M. Staron and C. Wohlin. An industrial case study on the choice between language customization mechanisms. In *Proc. 7th Int. Conf. Product-Focused Softw. Process Improv.*, volume 4034 of *LNCS*, pages 177–191. Springer, 2006.

[147] M. Staron, L. Kuzniarz, and C. Wohlin. Empirical assessment of using stereotypes to improve comprehension of UML models: A set of experiments. *J. Syst. Softw.*, 79(5):727–742, 2006.

[148] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework.* Addison-Wesley, second edition, 2008.

[149] M. Strembeck. Embedding policy rules for software-based systems in a requirements context. In *Proc. 6th IEEE Int. Worksh. Policies for Distrib. Syst. and Networks*, pages 235–238. IEEE, 2005.

[150] M. Strembeck and J. Mendling. Modeling process-related RBAC models with extended UML activity models. *Inform. Softw. Tech.*, 53(5):456–483, 2011.

[151] M. Strembeck and U. Zdun. Modeling interdependent concern behavior using extended activity models. *J. Object Technol.*, 7(6):143–166, 2008.

[152] M. Strembeck and U. Zdun. An approach for the systematic development of domain-specific languages. *Softw. Pract. Exper.*, 39(15):1253–1292, 2009.

[153] J. Suryadevara and R. K. Shyamasundar. UML-based approach to specify secured, fine-grained concurrent access to shared resources. *J. Object Technol.*, 6(1):107–119, 2007.

[154] A. Tretiakov and S. Hartmann. Higher-order entity relationship modelling with UML. In *Proc. 5th Int. Conf. Qual. Softw.*, pages 205–211. IEEE, 2005.

[155] J. Trujillo, E. Soler, E. Fernández-Medina, and M. Piattini. A UML 2.0 profile to define security requirements for data warehouses. *Comp. Stand. Inter.*, 31(5):969–983, 2009.

[156] M. G. Uddin and M. Zulkernine. UMLtrust: Towards developing trust-aware software. In *Proc. 23rd Annu. ACM Sym. Applied Comput.*, pages 831–836. ACM, 2008.

[157] R. Villarroel, E. Fernández-Medina, M. Piattini, and J. Trujillo. A UML 2.0/OCL extension for designing secure data warehouses. *J. Res. Pract. Inf. Tech.*, 38(1), 2006.

[158] H. Wada, J. Suzuki, and K. Oba. Modeling non-functional aspects in service oriented architecture. In *Proc. 3rd Int. Conf. Services Comput.*, pages 222–229, 2006.

[159] U. Zdun. A DSL toolkit for deferring architectural decisions in DSL-based software design. *Inform. Softw. Tech.*, 52(9):733–748, 2010.

[160] U. Zdun and P. Avgeriou. Modeling architectural patterns using architectural primitives. In *Proc. 20th Annu. ACM SIGPLAN Conf. Object-oriented Prog., Syst., Lang., Appl.*, pages 133–146. ACM, 2005.

[161] U. Zdun and M. Strembeck. Modeling composition in dynamic programming environments with model transformations. In *Proc. 5th Int. Sym. Softw. Compos.*, volume 4089 of *LNCS*, pages 178–193. Springer, 2006.

[162] U. Zdun and M. Strembeck. Reusable architectural decisions for DSL design: Foundational decisions in DSL development. In *Proc. 14th Europ. Conf. Patt. Lang. Prog.*, pages 1–37. ACM, 2009.

[163] G. Zhang and M. Hölzl. Weaving semantic aspects in HiLA. In *Proc. 11th Annu. Int. Conf. Aspect-oriented Softw. Dev.*, pages 263–274. ACM, 2012.

[164] H. Zhang, M. A. Babar, and P. Tell. Identifying relevant studies in software engineering. *Inform. Softw. Tech.*, 53(6):625–637, June 2011.

[165] Y. Zhang, Y. Liu, L. Zhang, Z. Ma, and H. Mei. Modeling and checking for non-functional attributes in extended UML class diagram. In *Proc. 32nd Annu. IEEE Int. Conf. Comp. Softw. and Appl.*, pages 100–107. IEEE, 2008.

[166] G. Zoughbi, L. Briand, and Y. Labiche. Modeling safety and airworthiness (RTCA DO-178B) information: Conceptual model and UML profile. *Softw. Syst. Model.*, 10(3):337–367, 2011.

[167] J. Zubcoff, J. Pardillo, and J. Trujillo. A UML profile for the conceptual modelling of data-mining with time-series in data warehouses. *Inform. Softw. Tech.*, 51(6):977–992, 2009.

A Revision to the Initial Version of the Catalog

Based on the findings derived from the extracted and codified decision data (via the SLR), we revised the catalog of decision records—the initial version was published as [70, 71]—to reach its current state (this document). The revision involved changes to the content as well as the presentation as follows:

Document structure

- *Defined decision template structure:* We specified the descriptive parts which must necessarily be included in a decision record (problem statement, decision context, decision drivers etc.) and added a conceptual overview (see Section 2.2).
- *Added format convention:* We added a convention for referencing decision points, corresponding options, associations between options, and DSML projects in a consistent way (see Section 2.4).
- *Added revision history:* The differences and additions to the initial version of the catalog [70, 71] are explained (this section).

Decision points (records)

- *Added decision-point descriptions:* We added a description for each decision point considered (D1–D6; see Section 2.1). The section also highlights the decision records which were applied frequently in our SLR study [141].
- *Added decision record:* We added a decision record and corresponding options for a newly introduced decision point, namely *behavior specification* (D5; see Section 4.5).

Decision options

- *Added decision options:* By studying the 80 related DSMLs found, we required a more fine-grained encoding schema which let us to the definition of a new decision point and new (corresponding) options:
  - *O5.1–O5.5:* Along with adding a new decision point (D5; see above), decision options O5.1–O5.5 were introduced. The pool of 80 DSMLs (obtained via the SLR) also provides known-usage examples for options which we were lacking from our initial resource collection (see, e.g., P53 [143] and P84 [124]).
  - *O6.5:* The description of alternative options for D6 (see Section 4.6) was extended to differentiate between different styles of M2M transformations (exogenous, endogenous; see O6.1 and O6.5). Again, we complemented each amendment to this revised description with examples (see, e.g., P17 [4] and P35 [2]).
- *Updated decision options:* Descriptions of decision options were revised according to the examples found by studying the 80 DSMLs. In addition, selected DSMLs out of this pool were added as known uses of an option to the decision records. The following changes in response to our findings are notable:

  - *O2.1:* The description was limited to M1 structural models only (see Section 4.2) as the newly introduced decision point D5 covers behavioral specifications (including M1 behavioral models; O5.1).
  - *O2.2:* The description was extended to cover the extension and/or redefinition of existing profile(s); rather than introducing new profiles only (see Section 4.2). Examples of this practice can be found in P36 [56, 98] and in P54 [6].
  - *O4.3:* The description was relaxed so that diagrammatic notations other than UML diagram notations (or variants thereof) are covered (see Section 4.4). This was triggered by examples found in P30 [16] and in P39 [93].

- *Made links between drivers and options explicit:* Positive and negative links between decision drivers and available options are listed in an overview table in the decision drivers section of each decision record (see Sections 4.1–4.6).

- *Added option thumbnails:* A comparatively large share of observed DSML designs can be described based on nine out of the 27 decision options provided by the revised catalog.[11] We, therefore, added thumbnail descriptions of nine base options to the catalog (see Table 6 in Section 3).

- *Highlighted options:* Each of the nine frequent options featured by found prototype option-sets and largest subsets was, in addition, highlighted in the individual decision-record descriptions by underlining the corresponding option number and title (see Sections 4.1–4.6).

- *Marked candidate options:* Three options (O3.2, O3.3, and O5.4; see Sections 4.3 and 4.5) not applied in any of the selected DSML projects were marked as candidate options in the decision-record descriptions (option number and title are put in parentheses). The options are preserved in the catalog because they have been identified as such by secondary studies and/or there are known uses which are documented in selected DSML designs not recovered or confirmed by empirical evidence such as with this SLR study.

- *Adapted presentation:* On the one hand, the representation of decision-option sets allowed us to remove pseudo-options (and the respective codes) which signal a combination of options taken at one decision point, which turned out not informative enough for our study [70, 71]. On the other hand, in some DSML projects it was not unambiguously clear whether an option had been applied or not. Thus, we introduced a representation for denoting options as underspecified (for O2.4, see, e.g., P41 [55] and P88 [34] in Table 13 in Appendix B).

- *Updated associations between options:* Based on the findings on *smallest option subsets* as well as the option combinations identified for generating prototype option-sets and largest components, we revised option associations within decision points (options of one and the same record) and

[11]Note that there are actually 31 decision codes (see, e.g., Table 13), the difference of four codes serving for coding pseudo-decision options; e.g., not taking any decision.

between them (see Section 5). These associations are also reflected in the subsections on the decision context, options, drivers, and consequences of the respective decision records (see Sections 4.1–4.6).

- Added navigation structure: *We added navigation structures (reading sequence for option descriptions, feature diagrams etc.) to the catalog to reflect possible reading paths based on the intention of a planned or DSML under review.*

DSML projects encoding

- *Complemented encoded decision options:* The initially encoded design decisions and options for our ten DSML projects were updated (according to the new encoding scheme) and the retrieved 80 related DSML projects were added to Table 13 in Appendix B.
- *Added domains and diagram types:* For each DSML, we added its application domain(s) and the tailored UML diagram type(s). An overview table of all 90 DSML projects was added providing the name, the domains, the diagram types, and the option set for every DSML (see Table 14 in Appendix C). Furthermore, Table 2 in Section 1 and Table 15 in Appendix D list the frequency of DSML-tailored diagram types and DSML application domains.

B Encoded Design Decisions and Options

This section provides an overview of chosen options per design decision for all 90 DSML projects (see Table 13). The DSML projects P1–P10 were performed by ourselves, the remaining projects were collected via the SLR (the protocol of the SLR is available at [141]). The SLR has been performed to find relevant UML/MOF-based DSML engineering approaches and to extract design decisions and corresponding options from them.

Table 13: Overview of encoded design decision points and corresponding options.

Decision/Option		P1 [151]	P2 [150]	P3 [137, 138]	P4 [136, 138]	P5 [68]	P6 [135]	P7 [67, 72]	P8 [139]	P9 [69]	P10 [161]	P11 [25]	P12 [3]	P13 [13]	P14 [14]	P15 [82]	P16 [83]	P17 [4]
D1	*Language-model definition*																	
O1.1	Textual description	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
O1.2	Formal textual model		×	×	×			×	×									
O1.3	Informal diagrammatic model																	
O1.4	Formal diagrammatic model		×	×	×			×	×				×				×	
D2	*Language-model formalization*																	
O2.1	M1 structural model																	
O2.2	Profile (re-)definition	×		×				×		×	×		×			×	×	×
O2.3	Metamodel extension	×	×	×	×	×	×	×	×	×	×	×		×	×			
O2.4	Metamodel modification			(×)[b]					(×)	(×)	(×)	(×)		(×)				
D3	*Language-model constraints*																	
O3.1	Constraint-language expression	×	×	×	×	×	×	×	×	×	×		×			×		×
O3.2	Code annotation																	
O3.3	Constraining M2M/M2T transformation							×										
O3.4	Informal textual annotation	×	×	×	×	×	×	×	×	×	×						×	
O3.5	None/Not specified											×		×	×			
D4	*Concrete-syntax definition*																	
O4.1	Model annotation	×		×				×		×	×		×			×	×	×
O4.2	Diagrammatic syntax extension	×	×	×	×	×			×		×							
O4.3	Mixed syntax (foreign syntax)									×								
O4.4	Frontend-syntax extension (hybrid syntax)																	
O4.5	Alternative syntax									×								
O4.6	Diagram symbol reuse	×		×				×		×	×		×	×	×	×	×	×
O4.7	None/Not specified						×					×						
D5	*Behavior specification*																	
O5.1	M1 behavior model			×														
O5.2	Formal textual specification																	
O5.3	Informal textual specification			×														
O5.4	Constraining model execution																	
O5.5	None/Not specified	×	×		×	×	×	×	×	×	×	×	×	×	×	×	×	×
D6	*Platform integration*																	
O6.1	Intermediate model representation		×					×										×
O6.2	Generation template									×			×			×		×
O6.3	API-based generator							×				×						
O6.4	(Direct) model execution		×															
O6.5	M2M transformation																	×
O6.6	None/Not specified	×		×	×	×	×		×		×			×	×		×	

[b]Parentheses denote an underspecification; i.e. it is not unambiguously clear whether the option has been applied or not.

Decision/Option		P18 [63, 64]	P19 [43]	P20 [103]	P21 [92]	P22 [122]	P23 [17, 18]	P24 [66]	P25 [48]	P26 [57]	P27 [9]	P28 [58]	P29 [47]	P30 [16]	P31 [132]	P32 [5]	P33 [12]	P34 [133]
D1	*Language-model definition*																	
O1.1	Textual description	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
O1.2	Formal textual model																	
O1.3	Informal diagrammatic model																	
O1.4	Formal diagrammatic model			×	×		×		×		×	×				×	×	×
D2	*Language-model formalization*																	
O2.1	M1 structural model																	
O2.2	Profile (re-)definition	×	×	×	×	×	×	×	×	×	×	×		×	×	×	×	×
O2.3	Metamodel extension												×					
O2.4	Metamodel modification												×					
D3	*Language-model constraints*																	
O3.1	Constraint-language expression	×				×	×	×			×			×		×		×
O3.2	Code annotation																	
O3.3	Constraining M2M/M2T transformation																	
O3.4	Informal textual annotation		×	×		×				×				×		×	×	×
O3.5	None/Not specified				×				×			×	×		×			
D4	*Concrete-syntax definition*																	
O4.1	Model annotation	×	×	×	×	×	×	×	×	×	×	×		×	×	×	×	×
O4.2	Diagrammatic syntax extension										×				×			
O4.3	Mixed syntax (foreign syntax)													×				
O4.4	Frontend-syntax extension (hybrid syntax)																	
O4.5	Alternative syntax																	
O4.6	Diagram symbol reuse	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
O4.7	None/Not specified																	
D5	*Behavior specification*																	
O5.1	M1 behavior model																	
O5.2	Formal textual specification													×				
O5.3	Informal textual specification																	
O5.4	Constraining model execution																	
O5.5	None/Not specified	×	×	×	×	×	×	×	×	×	×	×	×		×	×	×	×
D6	*Platform integration*																	
O6.1	Intermediate model representation			×												×		
O6.2	Generation template		×													×		
O6.3	API-based generator			×			×											
O6.4	(Direct) model execution																	
O6.5	M2M transformation			×			×									×		
O6.6	None/Not specified	×			×	×		×	×	×	×	×	×	×	×		×	×

Decision/Option		P35 [2]	P36 [56, 98]	P37 [166]	P38 [155]	P39 [93]	P40 [32]	P41 [55]	P42 [167]	P43 [125]	P44 [37]	P45 [90]	P46 [45]	P47 [20]	P48 [126]	P49 [24]	P50 [31]	P51 [73]
D1	*Language-model definition*																	
O1.1	Textual description	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
O1.2	Formal textual model									×								
O1.3	Informal diagrammatic model																	
O1.4	Formal diagrammatic model			×								×						
D2	*Language-model formalization*																	
O2.1	M1 structural model	×																
O2.2	Profile (re-)definition		×	×	×	×			×	×	×	×			×		×	×
O2.3	Metamodel extension		×				×	×					×	×		×	×	
O2.4	Metamodel modification		(×)				(×)	(×)					(×)			(×)		
D3	*Language-model constraints*																	
O3.1	Constraint-language expression		×	×	×		×		×			×	×	×			×	
O3.2	Code annotation																	
O3.3	Constraining M2M/M2T transformation																	
O3.4	Informal textual annotation			×	×		×				×		×	×			×	
O3.5	None/Not specified	×				×		×		×					×	×		×
D4	*Concrete-syntax definition*																	
O4.1	Model annotation		×	×	×	×			×	×	×	×			×		×	×
O4.2	Diagrammatic syntax extension		×		×			×	×				×		×			
O4.3	Mixed syntax (foreign syntax)					×												
O4.4	Frontend-syntax extension (hybrid syntax)																	
O4.5	Alternative syntax													×				
O4.6	Diagram symbol reuse		×	×	×	×			×	×	×	×			×	×	×	×
O4.7	None/Not specified	×					×											
D5	*Behavior specification*																	
O5.1	M1 behavior model																	
O5.2	Formal textual specification																	
O5.3	Informal textual specification																	
O5.4	Constraining model execution																	
O5.5	None/Not specified	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
D6	*Platform integration*																	
O6.1	Intermediate model representation		×															
O6.2	Generation template	×				×						×						×
O6.3	API-based generator		×															
O6.4	(Direct) model execution						×											
O6.5	M2M transformation	×	×		×								×					
O6.6	None/Not specified			×				×	×	×	×			×	×	×	×	

Decision/Option		P52 [119, 120]	P53 [143]	P54 [6]	P55 [23]	P56 [95]	P57 [1]	P58 [144]	P59 [106]	P60 [102]	P61 [42]	P62 [15]	P63 [78]	P64 [10]	P65 [88]	P66 [38]	P67 [128]	P68 [84]
D1	*Language-model definition*																	
O1.1	Textual description	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
O1.2	Formal textual model							×								×		
O1.3	Informal diagrammatic model						×			×								
O1.4	Formal diagrammatic model	×		×	×					×	×				×		×	
D2	*Language-model formalization*																	
O2.1	M1 structural model					×		×										
O2.2	Profile (re-)definition	×		×	×		×		×	×		×	×	×	×	×	×	×
O2.3	Metamodel extension		×								×							
O2.4	Metamodel modification		(×)								×							
D3	*Language-model constraints*																	
O3.1	Constraint-language expression	×	×						×				×					
O3.2	Code annotation																	
O3.3	Constraining M2M/M2T transformation																	
O3.4	Informal textual annotation	×	×						×			×				×	×	×
O3.5	None/Not specified			×	×	×	×	×		×	×			×	×			
D4	*Concrete-syntax definition*																	
O4.1	Model annotation	×		×	×		×		×	×		×	×	×	×	×	×	×
O4.2	Diagrammatic syntax extension											×			×			
O4.3	Mixed syntax (foreign syntax)																	
O4.4	Frontend-syntax extension (hybrid syntax)		×															
O4.5	Alternative syntax																	
O4.6	Diagram symbol reuse	×	×	×	×		×		×	×	×	×	×	×	×	×	×	×
O4.7	None/Not specified					×		×										
D5	*Behavior specification*																	
O5.1	M1 behavior model																	
O5.2	Formal textual specification																	
O5.3	Informal textual specification		×															
O5.4	Constraining model execution																	
O5.5	None/Not specified	×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
D6	*Platform integration*																	
O6.1	Intermediate model representation																	
O6.2	Generation template											×		×			×	
O6.3	API-based generator							×										
O6.4	(Direct) model execution																	
O6.5	M2M transformation																	
O6.6	None/Not specified	×	×	×	×	×	×		×	×	×		×		×	×		×

Decision/Option		P69 [96]	P70 [160]	P71 [8]	P72 [60]	P73 [134]	P74 [156]	P75 [154]	P76 [105]	P77 [129]	P78 [107]	P79 [11]	P80 [79]	P81 [158]	P82 [142]	P83 [157]	P84 [124]	P85 [86]
D1	*Language-model definition*																	
O1.1	Textual description	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
O1.2	Formal textual model							×				×						
O1.3	Informal diagrammatic model											×						
O1.4	Formal diagrammatic model							×	×									
D2	*Language-model formalization*																	
O2.1	M1 structural model																×	
O2.2	Profile (re-)definition	×	×	×	×	×	×	×	×	×	×	×	×	×		×		×
O2.3	Metamodel extension				×										×			
O2.4	Metamodel modification				(×)													
D3	*Language-model constraints*																	
O3.1	Constraint-language expression		×	×				×	×							×	×	
O3.2	Code annotation																	
O3.3	Constraining M2M/M2T transformation																	
O3.4	Informal textual annotation		×		×	×	×	×			×	×	×	×		×	×	
O3.5	None/Not specified	×								×					×			×
D4	*Concrete-syntax definition*																	
O4.1	Model annotation	×	×	×	×	×	×	×	×	×	×	×	×	×		×		×
O4.2	Diagrammatic syntax extension						×		×		×					×		
O4.3	Mixed syntax (foreign syntax)																	
O4.4	Frontend-syntax extension (hybrid syntax)																	
O4.5	Alternative syntax																	
O4.6	Diagram symbol reuse	×	×	×	×	×	×	×	×	×	×	×	×	×		×		×
O4.7	None/Not specified														×		×	
D5	*Behavior specification*																	
O5.1	M1 behavior model																×	
O5.2	Formal textual specification																	
O5.3	Informal textual specification																×	
O5.4	Constraining model execution																	
O5.5	None/Not specified	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×		×
D6	*Platform integration*																	
O6.1	Intermediate model representation																	
O6.2	Generation template													×				×
O6.3	API-based generator																	
O6.4	(Direct) model execution																	
O6.5	M2M transformation																	
O6.6	None/Not specified	×	×	×	×	×	×	×	×	×	×	×	×		×	×	×	

Decision/Option		P86 [50]	P87 [123]	P88 [34]	P89 [27]	P90 [165]	#	p/o[c]	o/d[d]	∑
D1	*Language-model definition*									
O1.1	Textual description	×	×	×	×	×	90	100%	69%	
O1.2	Formal textual model						10	11%	8%	
O1.3	Informal diagrammatic model						3	3%	2%	
O1.4	Formal diagrammatic model				×		28	31%	21%	131
D2	*Language-model formalization*									
O2.1	M1 structural model						4	4%	4%	
O2.2	Profile (re-)definition	×	×		×		67	74%	67%	
O2.3	Metamodel extension			×		×	27	30%	27%	
O2.4	Metamodel modification			(×)		(×)	2	2%	2%	100
D3	*Language-model constraints*									
O3.1	Constraint-language expression	×					41	46%	34%	
O3.2	Code annotation						0	0%	0%	
O3.3	Constraining M2M/M2T transformation						1	1%	1%	
O3.4	Informal textual annotation	×					45	50%	38%	
O3.5	None/Not specified		×	×	×	×	32	36%	27%	119
D4	*Concrete-syntax definition*									
O4.1	Model annotation	×	×		×		67	74%	38%	
O4.2	Diagrammatic syntax extension						21	23%	12%	
O4.3	Mixed syntax (foreign syntax)					×	4	4%	2%	
O4.4	Frontend-syntax extension (hybrid syntax)						1	1%	1%	
O4.5	Alternative syntax						2	2%	1%	
O4.6	Diagram symbol reuse	×	×	×	×		74	82%	42%	
O4.7	None/Not specified						8	9%	5%	177
D5	*Behavior specification*									
O5.1	M1 behavior model						2	2%	2%	
O5.2	Formal textual specification						1	1%	1%	
O5.3	Informal textual specification						3	3%	3%	
O5.4	Constraining model execution						0	0%	0%	
O5.5	None/Not specified	×	×	×	×	×	86	96%	93%	92
D6	*Platform integration*									
O6.1	Intermediate model representation						6	7%	6%	
O6.2	Generation template	×	×				17	19%	17%	
O6.3	API-based generator		×		×		8	9%	8%	
O6.4	(Direct) model execution						2	2%	2%	
O6.5	M2M transformation					×	9	10%	9%	
O6.6	None/Not specified			×			61	68%	59%	103
										722

[c] *Projects per option*: The percentage of projects which applied the design option.

[d] *Option per decision*: The percentage of an option chosen at a design decision point.

C Domains, Diagram Types, and Option Set per DSML

Table 14 lists all 90 DSMLs this catalog is based on, of which ten are our own developments (P1–P10), the rest has been retrieved via the SLR (P11–P90). The first column states the consecutive DSML project numbering used throughout this paper, the DSML's name (either specified by the authors themselves or—when no explicit name was mentioned—one chosen by us), and a reference to the corresponding publication(s). In the second column, the DSML application domain(s) are encoded according to the 2012 ACM Computing Classification System (CCS).[12] We have extracted the UML diagram type(s) tailored by a DSML as classified by the UML superstructure itself (shown in the third column of Table 14). The last column lists the decision-option set representing a DSML's design as encoded according to our catalog (see Section 4). Please note that Table 13 in Appendix B provides more details on the option set for each DSML in another view.

Table 14: Domains, diagram types, and option set per DSML project.

DSML	Domain(s)	Diagram type(s)	Option set
P1[13] ConcernActivities [151]	Access control, Software design engineering	Activity	$\{1.1, 2.2, 2.3, 3.1, 3.4, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P2 BusinessActivities [150]	Access control, Business process modeling, Software security engineering	Activity, Class	$\{1.1, 1.2, 1.4, 2.3, 3.1, 3.4, 4.2, 5.5, 6.1, 6.4\}$
P3[13] UML-PD [137, 138]	Access control, Business process modeling, Software security engineering	Activity, Class	$\{1.1, 1.2, 1.4, 2.2, 2.3, 3.1, 3.4, 4.1, 4.2, 4.6, 5.1, 5.3, 6.6\}$
P4 UML-DEL [136, 138]	Access control, Business process modeling, Software security engineering	Class	$\{1.1, 1.2, 1.4, 2.3, 3.1, 3.4, 4.2, 5.5, 6.6\}$
P5 SOF [68]	Business process modeling, Software security engineering	Activity	$\{1.1, 2.3, 3.1, 3.4, 4.2, 5.5, 6.6\}$
P6 UML-PD [135]	Access control, Business process modeling, Software security engineering	Activity, Class	$\{1.1, 2.3, 3.1, 3.4, 4.7, 5.5, 6.6\}$
P7[13] SOFServices [67, 72]	Business process modeling, Service-oriented architectures, Software security engineering, Web services	Activity, CompositeStructure	$\{1.1, 1.2, 1.4, 2.2, 2.3, 3.1, 3.3, 3.4, 4.1, 4.6, 5.5, 6.1, 6.3\}$
P8 UML-CC [139]	Access control, Business process modeling, Software security engineering	Class	$\{1.1, 1.2, 1.4, 2.3, 3.1, 3.4, 4.2, 5.5, 6.6\}$
P9[13] SecurityAudit [69]	Publish-subscribe / event-based architectures, Software security engineering	*[14]	$\{1.1, 2.2, 2.3, 3.1, 3.4, 4.1, 4.3, 4.5, 4.6, 5.5, 6.2\}$

[12]Available at `http://www.acm.org/about/class`; last accessed: Feb 2, 2015.

[13]The DSML's option set contains (at least) one of the seven prototype option-sets shown in Tables 4 and 5.

[14]The DSML does not tailor a UML diagram type specifically; for example, a stereotype extension of a UML element applicable in all diagram types, such as, `Element` (see, e.g., [27, 69])

DSML	Domain(s)	Diagram type(s)	Option set
P10[13] MTD [161]	Object oriented languages, Software architectures	Activity, Class, Object, StateMachine	$\{1.1, 2.2, 2.3, 3.1, 3.4, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P11 ADModel [25]	Business process modeling	Activity	$\{1.1, 2.3, 3.5, 4.7, 5.5, 6.3\}$
P12[13] AspectSM [3]	Robustness, Software development techniques, Software testing and debugging	StateMachine	$\{1.1, 1.4, 2.2, 3.1, 4.1, 4.6, 5.5, 6.2\}$
P13[13] UML4SPM [13]	Software development process management	Activity, Class	$\{1.1, 2.3, 3.5, 4.6, 5.5, 6.6\}$
P14[13] MDATC [14]	Reusability, Software development techniques, Software product lines	Activity, Package	$\{1.1, 2.3, 3.5, 4.6, 5.5, 6.6\}$
P15[13] TLM [82]	Model verification and validation, System on a chip	Class	$\{1.1, 2.2, 3.1, 4.1, 4.6, 5.5, 6.2\}$
P16[13] UPSS [83]	Service-oriented architectures	Class, CompositeStructure	$\{1.1, 1.4, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P17[13] BIT [4]	Software testing and debugging	Class	$\{1.1, 2.2, 3.1, 4.1, 4.6, 5.5, 6.1, 6.2, 6.5\}$
P18[13] UML4PF [63, 64]	Design patterns, Model checking, Requirements analysis, Security requirements	Class	$\{1.1, 2.2, 3.1, 4.1, 4.6, 5.5, 6.6\}$
P19[13] UP4WS [43]	Service-oriented architectures, Web services	Class	$\{1.1, 2.2, 3.4, 4.1, 4.6, 5.5, 6.2\}$
P20[13] CB [103]	Reusability, Software development techniques	Class, Component	$\{1.1, 1.4, 2.2, 3.4, 4.1, 4.6, 5.5, 6.1, 6.3, 6.5\}$
P21[13] AbstractSet [92]	Model verification and validation	Class, Package	$\{1.1, 1.4, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6\}$
P22[13] C2style [122]	Architecture description languages, Systems analysis and design	Component, Sequence	$\{1.1, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P23[13] MARTE-DAM [17, 18]	Embedded systems, Fault tree analysis, Real-time systems, Software fault tolerance, Transportation	Component, Sequence, StateMachine, UseCase	$\{1.1, 1.4, 2.2, 3.1, 4.1, 4.6, 5.5, 6.3, 6.5\}$
P24[13] UMM-Local-Choreographies [66]	Business process modeling, Orchestration languages	Activity	$\{1.1, 2.2, 3.1, 4.1, 4.6, 5.5, 6.6\}$
P25[13] RichService [48]	Service-oriented architectures, Web services	Class, Component, StateMachine	$\{1.1, 1.4, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6\}$
P26[13] UML-PMS [57]	Performance, Ubiquitous and mobile computing	Activity	$\{1.1, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6\}$

or `Constraint` (see, e.g., [37]).

DSML	Domain(s)	Diagram type(s)	Option set
P27[13] SOA [9]	Service-oriented architectures	Class, Component, Deployment	$\{1.1, 1.4, 2.2, 3.1, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P28[13] SWS [58]	Semantic web description languages, Web services	Activity	$\{1.1, 1.4, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6\}$
P29[13] eSPEM [47]	Software development process management	Activity, StateMachine	$\{1.1, 2.3, 2.4, 3.5, 4.6, 5.5, 6.6\}$
P30[13] RCSD [16]	Transportation	Class, Object	$\{1.1, 2.2, 3.1, 3.4, 4.1, 4.3, 4.6, 5.2, 6.6\}$
P31 UML-SOA-Sec [132]	Business process modeling, Security requirements, Service-oriented architectures, Web services	Activity	$\{1.1, 2.2, 3.5, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P32[13] UML2Alloy [5]	Model verification and validation	Class, Package	$\{1.1, 1.4, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.1, 6.2, 6.5\}$
P33[13] ExSAM [12]	Avionics, Embedded systems, Engineering	CompositeStructure	$\{1.1, 1.4, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P34[13] UACL [133]	Availability, Telecommunications	Class, Component	$\{1.1, 1.4, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P35 SECTET [2]	Service-oriented architectures, Software security engineering, Web services	Class	$\{1.1, 2.1, 3.5, 4.7, 5.5, 6.2, 6.5\}$
P36[13] UML4SOA [56, 98]	Service-oriented architectures	Activity, Class, Component	$\{1.1, 2.2, 2.3, 3.1, 4.1, 4.2, 4.6, 5.5, 6.1, 6.3, 6.5\}$
P37[13] SafeUML [166]	Avionics, Software safety	Class, Package	$\{1.1, 1.4, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P38[13] IStarDW [155]	Data warehouses, Security requirements	Class, Package	$\{1.1, 2.2, 3.1, 3.4, 4.1, 4.2, 4.6, 5.5, 6.5\}$
P39[13] TestOracle [93]	Software testing and debugging	StateMachine	$\{1.1, 2.2, 3.5, 4.1, 4.3, 4.6, 5.5, 6.2\}$
P40 MOCAS [32]	Model checking, Model verification and validation	Object	$\{1.1, 2.3, 3.1, 3.4, 4.7, 5.5, 6.4\}$
P41 CCFG [55]	Model verification and validation	Activity	$\{1.1, 2.3, 3.5, 4.2, 5.5, 6.6\}$
P42[13] TimeSeriesAnalysis [167]	Data mining, Data warehouses	Class, Object	$\{1.1, 2.2, 3.1, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P43 ADOM-UML [125]	Model verification and validation, Requirements analysis, Software design engineering	*	$\{1.1, 1.2, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6\}$

DSML	Domain(s)	Diagram type(s)	Option set
P44[13] Predefined-Constraints [37]	Model checking	*	{1.1, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6}
P45[13] TAM-PM [90]	Graphical user interfaces, Web interfaces	Activity, Class	{1.1, 1.4, 2.2, 3.1, 4.1, 4.6, 5.5, 6.2}
P46 SPEM4MDE [45]	Software development process management	Activity, StateMachine	{1.1, 2.3, 3.1, 3.4, 4.2, 5.5, 6.5}
P47 CSSL [20]	Collaborative and social computing	Class, StateMachine	{1.1, 2.3, 3.1, 3.4, 4.5, 5.5, 6.6}
P48 SystemC [126]	Embedded systems, System on a chip	CompositeStructure, StateMachine	{1.1, 2.2, 3.5, 4.1, 4.2, 4.6, 5.5, 6.6}
P49[13] UML2Ext [24]	Requirements analysis, Software product lines	UseCase	{1.1, 2.3, 3.5, 4.6, 5.5, 6.6}
P50[13] HM^3 [31]	Hypertext languages	Class, UseCase	{1.1, 2.2, 2.3, 3.1, 3.4, 4.1, 4.6, 5.5, 6.6}
P51[13] WCAAUML [73]	Web applications, Web interfaces	Class, Deployment, Package	{1.1, 2.2, 3.5, 4.1, 4.6, 5.5, 6.2}
P52[13] IEC61508 [119, 120]	Model verification and validation, Safety critical systems	Class, Package	{1.1, 1.4, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.6}
P53[13] UCDM [143]	Use cases	UseCase	{1.1, 2.3, 3.1, 3.4, 4.4, 4.6, 5.3, 6.6}
P54[13] SPArch [6]	Software architectures, Software development process management	Class, Component, Package	{1.1, 1.4, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6}
P55[13] MoDePeMART [23]	Measurement, Metrics, Software performance	Class, StateMachine	{1.1, 1.4, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6}
P56 UPCC [95]	Enterprise data management, Service-oriented architectures, Web services	Class	{1.1, 2.1, 3.5, 4.7, 5.5, 6.6}
P57 SELinux [1]	Access control, Operating systems security, Security requirements	Class	{1.1, 1.3, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6}
P58 UML-GUI [144]	Graphical user interfaces	Class, Component	{1.1, 1.2, 2.1, 3.5, 4.7, 5.5, 6.3}
P59[13] SHP [106]	Software security engineering	Class, Package	{1.1, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.6}
P60[13] SMF [102]	Fault tree analysis, Safety critical systems, Software safety	Class, Component, UseCase	{1.1, 1.3, 1.4, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6}
P61[13] DMM/UCMM [42]	Graphical user interfaces	Class, UseCase	{1.1, 1.4, 2.3, 2.4, 3.5, 4.6, 5.5, 6.6}

DSML	Domain(s)	Diagram type(s)	Option set
P62[13] CUP 2.0 [15]	Graphical user interfaces	Activity, Class, Package	$\{1.1, 2.2, 3.4, 4.1, 4.2, 4.6, 5.5, 6.2\}$
P63[13] REMP [78]	Embedded systems, Real-time systems, Software testing and debugging	Class, StateMachine	$\{1.1, 2.2, 3.1, 4.1, 4.6, 5.5, 6.6\}$
P64[13] DPL [10]	Web services	Activity	$\{1.1, 2.2, 3.5, 4.1, 4.6, 5.5, 6.2\}$
P65[13] WebRE [88]	Requirements analysis, Web applications	Activity, UseCase	$\{1.1, 1.4, 2.2, 3.5, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P66[13] AOM-AD [38]	Software development techniques	Activity	$\{1.1, 1.2, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P67[13] Reliability [128]	Software reliability	InteractionOverview, Sequence	$\{1.1, 1.4, 2.2, 3.4, 4.1, 4.6, 5.5, 6.2\}$
P68[13] UML-AOF [84]	Software development techniques	Class, Package	$\{1.1, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P69 CompSize [96]	Embedded systems, Estimation, Measurement, Metrics	Class, Component	$\{1.1, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6\}$
P70[13] Architectural-Primitives [160]	Design patterns, Software architectures	Component	$\{1.1, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P71[13] CUP [8]	Error detection and error correction, Model checking	CompositeStructure, Sequence	$\{1.1, 2.2, 3.1, 4.1, 4.6, 5.5, 6.6\}$
P72[13] GWfM-Sec [60]	Orchestration languages, Software security engineering, Web services	Activity	$\{1.1, 2.2, 2.3, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P73[13] SoC [134]	Hardware description languages and compilation, System on a chip	Activity, Class, CompositeStructure, Deployment	$\{1.1, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P74[13] UMLtrust [156]	Scenario-based design, Software development techniques, Trust frameworks	Class, Package, UseCase	$\{1.1, 2.2, 3.4, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P75[13] HERM [154]	Database design and models	Class	$\{1.1, 1.2, 1.4, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P76[13] WebML [105]	Web applications, Web interfaces	Class, Component, CompositeStructure	$\{1.1, 1.4, 2.2, 3.1, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P77 ODP [129]	Distributed architectures	Class, Component, Object, Sequence	$\{1.1, 2.2, 3.5, 4.1, 4.6, 5.5, 6.6\}$
P78[13] EIS [107]	Enterprise information systems	Activity, Component	$\{1.1, 2.2, 3.4, 4.1, 4.2, 4.6, 5.5, 6.6\}$

DSML	Domain(s)	Diagram type(s)	Option set
P79[13] SPTExt [11]	Embedded systems, Real-time systems	Activity	$\{1.1, 1.2, 1.3, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P80[13] CAV [79]	Software architectures, Software evolution	Class	$\{1.1, 2.2, 3.4, 4.1, 4.6, 5.5, 6.6\}$
P81[13] SOA-NF [158]	Service-oriented architectures	CompositeStructure	$\{1.1, 2.2, 3.4, 4.1, 4.6, 5.5, 6.2\}$
P82 SECRDW [142]	Data warehouses, Security requirements	Class, Package	$\{1.1, 2.3, 3.5, 4.7, 5.5, 6.6\}$
P83[13] SECDW [157]	Data warehouses, Security requirements	Class, Object	$\{1.1, 2.2, 3.1, 3.4, 4.1, 4.2, 4.6, 5.5, 6.6\}$
P84 EM [124]	Electronic commerce, Web applications	Class, StateMachine, UseCase	$\{1.1, 2.1, 3.1, 3.4, 4.7, 5.1, 5.3, 6.6\}$
P85[13] WS-CM [86]	Web applications, Web services	Class, StateMachine	$\{1.1, 2.2, 3.5, 4.1, 4.6, 5.5, 6.2\}$
P86[13] aspectJ [50]	Software development techniques	Class, Package	$\{1.1, 2.2, 3.1, 3.4, 4.1, 4.6, 5.5, 6.2\}$
P87[13] ContextUML [123]	Service-oriented architectures, Web services	Class	$\{1.1, 2.2, 3.5, 4.1, 4.6, 5.5, 6.2, 6.3\}$
P88[13] DifferenceMM [34]	Software evolution	Class	$\{1.1, 2.3, 3.5, 4.6, 5.5, 6.6\}$
P89[13] Versioning [27]	Software evolution, Version control	*	$\{1.1, 1.4, 2.2, 3.5, 4.1, 4.6, 5.5, 6.3\}$
P90 NFA [165]	Avionics, Model checking	Class	$\{1.1, 2.3, 3.5, 4.3, 5.5, 6.5\}$

D Application Domains

To map the domain coverage of the 90 DSML projects, we classified every DSML according to the 2012 ACM Computing Classification System (CCS).[15] Table 15 shows the frequency of categories assigned to the selected DSML projects. In total, we used 63 distinct CCS categories and we assigned 177 category tags.

Table 15: Frequency of DSML application domains.

Domain	Frequency
Service-oriented architectures	11
Software security engineering	11
Web services	11
Business process modeling	10
Access control	7
Model verification and validation	7
Software development techniques	7
Embedded systems	6
Security requirements	6
Model checking	5
Web applications	5
Data warehouses	4
Graphical user interfaces	4
Requirements analysis	4
Software architectures	4
Software development process management	4
Software testing and debugging	4
Avionics	3
Real-time systems	3
Software evolution	3
System on a chip	3
Web interfaces	3
Design patterns	2
Fault tree analysis	2
Measurement	2
Metrics	2
Orchestration languages	2
Reusability	2
Safety critical systems	2
Software design engineering	2
Software product lines	2
Software safety	2
Transportation	2
Architecture description languages	1
Availability	1
Collaborative and social computing	1
Data mining	1

[15] Available at `http://www.acm.org/about/class`; last accessed: Feb 2, 2015.

Domain	Frequency
Database design and models	1
Distributed architectures	1
Electronic commerce	1
Engineering	1
Enterprise data management	1
Enterprise information systems	1
Error detection and error correction	1
Estimation	1
Hardware description languages and compilation	1
Hypertext languages	1
Object oriented languages	1
Operating systems security	1
Performance	1
Publish-subscribe / event-based architectures	1
Robustness	1
Scenario-based design	1
Semantic web description languages	1
Software fault tolerance	1
Software performance	1
Software reliability	1
Systems analysis and design	1
Telecommunications	1
Trust frameworks	1
Ubiquitous and mobile computing	1
Use cases	1
Version control	1