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

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

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

Abstract

In the process of model-driven development (MDD) of software artifacts, domain-specific modeling languages (DSMLs) are an integral part. They act as the communication vehicle for aligning the requirements of the domain expert with the needs of the software engineer. With the rise of the UML as de facto standard for modeling software systems, MOF/UML-based DSMLs are now widely used for MDD. This paper documents design decisions from ten DSML projects which are based on the MOF/UML and which we conducted over the last years. We present our experiences in the form of reusable decision templates for all decision points detected in each phase of the DSML development process. Furthermore, we report also on identified decision dependencies which may occur within a single decision or between two decisions.

1 Introduction

This paper presents our experiences gained from the development of ten domain-specific modeling language (DSML) projects based on the MOF/UML (see Table 1). The DSML development phases are adopted from [37] and can be summarized as: 1) core language model definition, 2) behavior definition, 3) concrete syntax definition, and 4) platform integration. The sequence we developed our DSMLs in, maps to a language model driven approach (wherein step 2) and 3) are performed in parallel) [43].

Projects P2–P9 provide support for various security properties, such as, role-based access control (RBAC), process-related duties, data confidentiality and integrity etc. These DSMLs are based on a common and generic metamodel defined in P2. The other two DSMLs support, on the one hand, the modeling

*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.

of interdependent concern behavior (P1) and, on the other hand, the modeling of composition in dynamic programming environments (P10).

#	Objectives	Domain
P1	An approach to model interdependent concern behavior using extended UML activity models [39].	Separation of concerns
P2	An integrated approach for modeling processes and process-related RBAC models (roles, hierarchies, statically and dynamically mutual exclusive tasks etc.) [38].	Business processes, role-based access control (RBAC)
P3	A UML extension for an integrated modeling of business processes and process-related duties; particularly the modeling of duties and associated tasks in business process models [32, 34].	Business processes, process-related duties
P4	An approach to provide modeling support for the delegation of roles, tasks, and duties in the context of process-related RBAC models [34, 33].	Business processes, delegation of roles, tasks, and duties
P5	A UML extension to model confidentiality and integrity of object flows in activity models [15].	Data confidentiality and integrity
P6	UML modeling support for the notion of mutual exclusion and binding constraints for duties in process-related RBAC models [31].	RBAC (consistency checks for duties)
P7	Incorporation of data integrity and confidentiality into the MDD of process-driven SOAs [13, 14].	Integrity and confidentiality for service invocations
P8	Integration of context constraints with process-related RBAC models and thereby supporting context-dependent task execution [35].	Business processes, RBAC, context constraints
P9	A generic UML extension for the definition of audit requirements and specification of audit rules at the modeling-level [16].	Audit rules
P10	An approach based on model transformations between the valid structural and behavioral runtime states that a system can have [42].	Model transformation

Table 1: Overview of conducted DSML development projects.

2 Collected Design Decisions

2.1 D1 Language Model Definition

Decision *How should the domain (or domain fragment) be described?*
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 [7], information about the selected domain is collected (e.g., based on literature reviews, scenario analyzes, and collected expert knowledge) and is evaluated. If the domain is already captured by an existing software system, artifacts related to the software system (e.g., code base, documentation, test suites) act as input for the domain analysis. Based on this material, a structured domain description (referred to as a *generic language model* [40], hereafter) is defined. The domain description provides a domain definition, the domain vocabulary, and a catalog of domain concepts and concept relations. The domain concepts can be described using narrative text and/or using textual or diagrammatic specification formalisms. 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 2.2).
Options

O1.1 Textual descriptions: Textual artifacts describe domain abstractions in an informal way (e.g., narrative free text).

O1.2 Formal textual models: For instance, mathematical expressions (e.g., set algebra) or formal grammars (e.g., the Extended Backus-Naur Form [18])

#	Decision/Option	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
D1	*Language model definition*										
O1.1	Textual descriptions	×	×	×	×	×	×	×	×	×	×
O1.2	Formal textual models		×	×	×			×	×		
O1.3	Informal diagrammatic models										
O1.4	Formal diagrammatic models		×	×	×			×	×		
O1.5	Combination of options[1]		∧	∧	∧			⊂	∧		
D2	*Language model formalization*										
O2.1	M1 structural and behavioral models			×							
O2.2	Profile definition	×		×				×		×	×
O2.3	Metamodel extension	×	×	×	×	×	×	×	×	×	×
O2.4	Metamodel modification										
O2.5	Combination of options[1]	⊂		⊂				≍		⊂	⊂
D3	*Language model constraints*										
O3.1	Explicit constraint expressions	×	×	×	×	×	×	×	×	×	×
O3.2	Code annotations										
O3.3	Constraining M2M/M2T transformations							×		×	
O3.4	Textual annotations										
O3.5	Combination of options[1]							≍		∧	
O3.6	None										
D4	*Concrete syntax definition*										
O4.1	Model annotations										
O4.2	Diagrammatic syntax extension	×	×	×	×	×			×		×
O4.3	Mixed syntax (foreign syntax)										
O4.4	Frontend-syntax extension (hybrid syntax)										
O4.5	Alternative syntax									×	
O4.6	Reusing diagram symbols							×			
O4.7	None						×				
D5	*Platform integration*										
O5.1	Intermediate model representation							×			
O5.2	Generation templates									×	
O5.3	API-based generators							×			
O5.4	Direct model execution		×	×	×		×		×		
O5.5	Combination of options[1]							⊂			
O5.6	None	×				×					×

Table 2: Overview of design decision points and options.

provide means for well-formed and unambiguous definitions of domain concepts and relations.

O1.3 Informal diagrammatic models: Domain concepts are sketched in ad hoc diagrams, with the diagrammatic representation not being compliant to a specified software modeling language and corresponding diagrammatic production rules. Examples are forms of visual concept modeling (e.g., early feature diagrams) or pseudo UML diagrams (e.g., class diagram notations being used as recomposable drawing shapes).

O1.4 Formal diagrammatic models: The domain concepts are expressed by means of a formally specified modeling language (e.g., MOF, UML, ER, STATEMATE) which adopts a graphical representation (e.g., UML class, UML activity, and/or STATEMATE statecharts).

O1.5 Combination of options: For instance, to facilitate communicating concepts, diagrammatic models (O1.3, O1.4) can be used in support of a predominantely informal textual description (O1.1). For explanatory purposes, normative and formal textual definitions (O1.2) are commonly supported by non-normative and informal textual descriptions (O1.1).

[1]The symbol ⊂ indicates that the options for a decision point are complementing each other; if the options for a decision point are equivalent, the symbol ≍ is used; if more than three options are applied for a decision point, the symbol ∧ shows that some of these options are complementary and some are equivalent.

Drivers

Existing diagrammatic domain descriptions: If either formal or informal diagrammatic descriptions are available (e.g., a pseudo or compliant UML M1 class model), a domain description could be devised as a refinement. 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).

Audience of the description: The language model definition is used as a mutual communication vehicle for both, the targeted 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. Or, addressing software engineers, the UML family of diagrams can be assumed known (O1.4). For non-technical business experts, prior experiences [20, 30] suggest that a process-oriented view (e.g., task and data flows) and process-oriented notations (e.g., UML activities, BPMN models) are more adequate.

Correspondence mismatches: If the domain is described in a generic manner by adopting a formal representation (O1.2, O1.4, O1.5), it needs to be transformed into a formal UML-compliant operationalization model (see D2 in Section 2.2). Transformation needs result from various mismatches:

1. A mismatch between modeling languages: For example, when using ER modeling for describing domain concepts, 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 as ER concepts). A domain description might stress a behavioral angle (e.g., using statecharts) while an operationlized language model (e.g., due to the specificity of the modeling language) requires additional structure details (e.g., properties of domain concepts).

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 UML/MOF context: meta-metamodel, metamodel, model, instance/repository model [27, 25]) can be adopted. Having defined, for example, the domain description at the UML level M1 raises the issue of creating a mapping up to a metamodel (M2) for operationalizing the language model.

4. Mismatches can occur in any combination of the previous three mismatch categories.

Consistency: The effort required to preserve the consistency between different domain description artifacts (e.g., diagrams and textual descriptions) is a critical factor when considering a combined option (O1.5). The negative effects

of introducing inconsistency, e.g., between a diagram and its textual description, can be mitigated by declaring either representation the normative one.

Cognitive effectiveness: A decision between any formal textual (O1.1) 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 concepts (see [17] 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 [5]. 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).

Consequences The initial phase of the language model definition has to cover all domain-specific concepts from the selected target domain and precedes the formalization into the MOF/UML. Output of this phase are the core language model concepts, whereas the description form depends on the application domain and involved domain experts. The language model description can be informal (O1.1, O1.3), in a structured form (O1.2, O1.4), or as combinations thereof (O1.5). If the definition is not based on the MOF (e.g., an option in 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, 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 (O1.5), we adopted formal textual (O1.2) and UML/MOF-based diagrammatic definitions (O1.4) in P2–P4, P7, and P8. Additionally, all language model definitions of our DSML projects (P1–P10) are accompanied by informal textual descriptions (O1.1).

Example 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 definitorial 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, e.g., 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 (untyped) 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: $\forall o_s \in O, o_t \in successors(o_s) : ofpath(o_s, o_t) \neq \emptyset$.

2.2 D2 Language Model Formalization

Decision *In which UML-compliant way should the domain concepts be formalized?*
Context After the identification of language model concepts, these definitions serve as input for the phase of formalizing the domain constructs into the core language model expressed via the UML.
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. [27, 3]).

O2.1 M1 structural and behavioral models: UML structural models are an ad-hoc instrument to formalize domain abstractions. In a class model, for instance, domain concepts can be expressed as classes and relationships as associations. UML behavioral models (e.g., state machines) can be used to specify the behavior of language model elements.

O2.2 UML profiles: Profiles are a language extension option to tailor the UML for different purposes. A profile consists of a set of stereotypes which define how an *existing* UML metaclass may be extended.

O2.3 Metamodel extension: A metamodel extension introduces new metaclasses and/or new associations between metaclasses (MOF-based extension [25]).

O2.4 Metamodel modification: In contrast to a metamodel extension, existing metaclasses of the UML metamodel are modified; e.g., by changing the type of a class property or by deleting existing associations (MOF-based extension [25]).

O2.5 Combination of options: A combination may include the definition of a metamodel extension as well as an equivalent profile definition (e.g., P7). Similarly, stereotype definitions can be provided to accompany a metamodel modification (e.g., P9).
Drivers

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

DSML expressiveness: For instance, a UML profile (O2.2) can only specialize a metamodel in such a way that the profile semantics do not conflict with the semantics of this referenced metamodel. Therefore, profile constraints may only define well-formed rules that are more constraining (but consistent with) those specified by the metamodel [27]. In contrast, a metamodel extension/-modification (O2.3 and O2.4) is only limited by the constraints imposed by the MOF metamodel (e.g., the abstract syntax of the UML can be extended).

Portability and evolution: A newly created metamodel (O2.3 and O2.4) is a fork of a certain version of the UML specification. Thus, the metamodel possibly

needs to be adapted to conform with newly released OMG specifications. Reusability of a UML extension is also affected by being either compliant with the UML standard (e.g., O2.2) or not (e.g., O2.4).

DSML integration: Preexisting DSMLs, software systems, and tool support have a direct impact on the design process of a DSML in terms of integration possibilities. For instance, the UML specification defines a standardized way to use icons and display options for profiles (O2.2). Tool support for authoring UML models and profiles (O2.1 and O2.2) is widely available.

Consequences

Metamodel dependencies: Certain dependencies can result from combined language model formalizations (O2.5). 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).

Unambiguous language model: If no further constraints to the language model are specified (see 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; O1.2).

Application In all 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. In P3 we define the DSML's behavior also via UML M1 models (O2.1). Therefore, we effectively adopted combined strategies (O2.5).

Example Fig. 1 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 and, on the right hand side, a UML profile specification named `SOF::Services`. Mappings between these two language-model representations are provided as M2M transformations. Both UML customizations provide the same modeling capabilities for using one of our UML security extensions (for details see [15, 13]) with the SoaML specification [28].

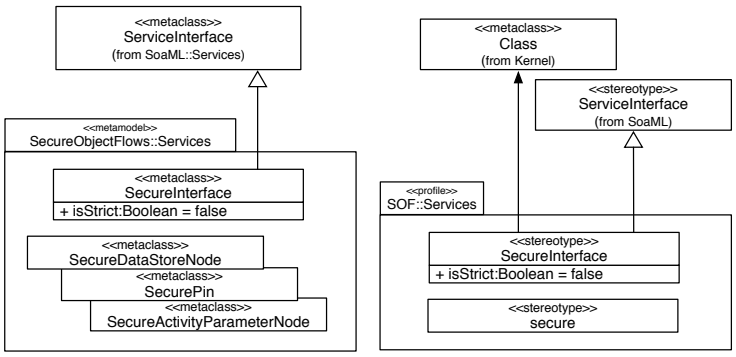

Figure 1: Exemplary UML metamodel extension and profile definition [13].

2.3 D3 Language Model Constraints

Decision *Do we have to define constraints over the core language model(s)? If so, how should these constraints be expressed?*
Context A core language model has been formalized in the UML, using either a UML metamodel extension/modification, a UML profile, or a UML class model (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 concepts that are relevant for the description of the target domain. Examples are invariants for domain concepts, pre-conditions and post-conditions, as well as guards. As a result, the language model formalization could be incomplete or ambiguous.

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

O3.1 Constraint-language expressions: One can make language model constraints explicit using an constraint-expression language, for example, via the Object Constraint Language (OCL) or the Epsilon Validation Language (EVL).

O3.2 Code annotations: The language model and its elements are enriched through annotations which contain expressions in the host language (or a language embedded within the host language). For instance, this can be realized by using model annotations and UML's `OpaqueExpression` [27].

O3.3 Constraining M2T/M2M transformations: The constraints over the language model are expressed at the level of transformation templates. That is, template expressions contain checks (e.g., conditional statements based on model navigation expressions) which test model instances for the implicit fit with corresponding domain constraints; As for M2T transformations, for example, conditional statements in the Epsilon Transformation Language (ETL) based on Epsilon Object Language (EOL) expressions can be used to specify structural constraints over the language model (i.e., at the model instance level) and to enforce them at each transformation run.

O3.4 Textual annotations: Certain constraints (e.g., temporal bindings) elicited from the target domain cannot be captured sufficiently via evaluable expressions (i.e., constraint language expressions, code annotations) and/or the constraints are intended to serve a documentary purpose (to the domain expert). In such cases, unstructured text annotations may capture constraint descriptions, meant for the human reader only (for example via UML comments).

O3.5 Combination of options: For instance, textual annotations are used in addition to constraint-language expressions.

O3.6 None: Static constraints over the language model are not made explicit in (or along with) the language model.
Drivers

Constraint formalization: In early iterations (e.g., DSML prototyping), constraints might not be expressed via well-formed, syntactically valid constraint-language expressions, but rather as pseudo-expressions or unstructured text.

With the language model maturing during subsequent iterations these annotations can be transformed into evaluable expressions.

Automated language model checking: If tool integration for model checking is a requirement, the options O3.1–O3.3 are candidates. A driver toward either option is the intended model-checking time. Relevant points in time follow from the model formalization option adopted (e.g. class model vs. metamodel-based) and the platform-support (model-level or instance-level checks). Language-model checking based on template expressions (O3.3) realizes the latest possible checking point. Therefore, this option does not offer any constraint-based feedback during model development.

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

Maintainability: Explicitly defined model constraints (O3.1 through O3.3) create structured text artifacts which must be maintained along with the model artifacts (e.g., the XMI representation). Toolkits and their model representations offer different strategies for this purpose, e.g. embedding constraints into model elements (i.e.; model annotations, such as UML comments), maintaining constraint collections as external resources (e.g. separate text files), or editor integration. Each strategy affects the artifact complexity and the effort needed to keep the constraints and the models synchronized.

Portability: If the portability of constraints between different MDD toolkits (e.g., Eclipse MDT, Rational Software Architect, MagicDraw, Dresden OCL) is a mandatory requirement, platform-dependent options O3.2 and O3.3 can 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 OCL/UML specifications (semantic variation points, e.g., 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; O2.2) or at a later stage (e.g., for M2T transformations; O2.3). It must be ensured that language model constraints do not contradict their language model formalization and vice versa. Moreover, constraints may need to be adapted when the metamodel changes (e.g., OCL navigation expressions).

Consequences When choosing 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, either 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 M2T/M2M transformation template expressions used to validate model instances or code/textual annotations can be produced as output artifacts. The decision on which kind of constraint definition is the most applicable is highly dependent on the actual stage of the DSML project, available tool support and tool integration. The DSML core language model and the DSML language model constraints serve as an input

for the subsequent definition of the concrete DSML syntax.
Application In our DSMLs, we encountered all options but code annotations (O3.2) and unconstrained language models (O3.6). In particular, we provide constraint-language expressions (O3.1) in the OCL for all of our DSMLs. This is because precise execution semantics were to be expressed in terms of the foundations of UML activities (token flows, e.g., in P1) and of the UML state machines (state/transition; in P10). In eight out of ten DSMLs (P2–P9), these semantics are 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). For two DSMLs (P7, P9), we additionally incorporated constraining M2T transformations (O3.3). Textual annotations (O3.4) are either used to complement OCL constraints (P2, P5, P10) or as full substitutes (P8) for otherwise formally expressed constraints.
Example 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 (M0), the operations are called to evaluate whether an action should be entered, depending upon some contextual state. Constraint 1 shows a complementary textual annotation. Constraint 5 exemplifies a constraint expressed in natural language due to a model-level mismatch: While the constraint is captured at the language-model level (M2), the operation calls (whose boolean return values are folded together to yield the runtime evaluation of the guard) become manifest at the occurrence level of an activity (M0) only.

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

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

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

2.4 D4 Concrete Syntax Definition

Decision *In which representation should the domain modeler create models using the DSML?*
Context The concrete syntax style of a UML-based DSML serves as the interface presented to the user. 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 13 diagram types [27]. The number of distinct graphical symbols applicable in these diagram types ranges from 8 (in communication diagrams) to 60 (e.g., in class diagrams) [23]. A DSML derived from the MOF and/or the UML can add new elements to this symbol vocabulary.

There are also secondary, non-diagrammatic representation candidates available for the MOF and for the UML. Important examples are textual, tree-structured, and tabular notations. A *textual* concrete syntax expresses DSML models in a text-based format. To explicitly specify the format rules, grammars of the textual syntax needs are defined (e.g., via the Extended Backus-Naur Form [18]) and a parser infrastructure is generated therefrom. 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 (e.g., the default UML editor as provided by the Eclipse MDT). 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 (similar to the user interface of language workbenches [9]).

Options

O4.1 Model annotations: Based on UML comments, keywords, narrative statements, or formal definitions (see, e.g., [19]) are attached to the elements of language model instances. 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 and maintains a list of predefined keywords [27].

O4.2 Diagrammatic syntax extension: The DSML is to be used in a diagrammatic manner by extending one or multiple UML diagram types. This syntax extension is achieved by creating and by specifying novel symbols to be added to the basic UML symbol set. The new symbols can be derived from existing shapes. In principle, the design space for the new symbols is unlimited and only follows from the requirements of the target domain. However, existing guidelines for designing UML symbols should be considered (e.g., avoidance of synographs; see, e.g., [23]). The symbol description can be structured according to the form adopted by the UML specification documents [27]: 1) A descriptive and detailed statement about each symbol introduced, 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 and integrated example showing the symbols in interaction. 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 full replacements for the standard notions of stereotyped elements (e.g., tags, nested icons in classifier rectangles).

O4.3 Mixed syntax (foreign syntax): The DSML's concrete syntax is described in any non-diagrammatic syntax type (textual, tree-based, tabular). Hence, the DSML concrete syntax remains *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. The foreign syntax is embedded into the primary, diagrammatic UML syntax, most importantly by using 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): The DSML's concrete syntax is non-diagrammatic (textual, tree-based, tabular) and is realized as an extension to 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 from 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, while covering (subsets of) the UML abstract 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: For the DSML, a graphical syntax extension of the UML is applied (O4.2). In addition, an alternative foreign syntax (O4.3) and/or an alternative frontend-syntax extension (O4.4) are introduced. As a result, DSML models can either be expressed diagrammatically in the extended UML notation, as a combination of base UML diagrams with 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 Reusing diagram symbols: No custom, DSML-specific extension to the standard UML symbol vocabulary is created. With the family of UML specifications [27] 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. This resembles the practise applied in the UML specification itself. For example, for the `Class` metaclass which specializes the `Classifier` metaclass, the UML states (see Section 7.3.7 in [27]):

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

O4.7 None: 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.

Drivers

Non-diagrammatic UML notations: Textual notations [10] for the UML are auxiliary representations and limited in their visual expressiveness; that is, from the perspective of the DSML engineer, the textual representations do not qualify as a valid choice of the concrete syntax style for their DSML. They are mere frontend syntaxes (O4.4). As an important example, in XMI [26] 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 an UML extension for XMI (e.g., the Eclipse UML2XMI schema) is presupposed. Major pitfalls of an XMI-based extension are syntactic com-

plexity and cognitive load imposed on the modeler by the XML representation. Also, the required UML extensions are commonly vendor or tool specific.

As an alternative and a *standard* textual notation aiming at a human reader/writer, HUTN [24] suffers from similar limitations as XMI. First, its expressiveness targets the MOF (or MOF-like modeling infrastructures, such as, Ecore). With this, only MOF views of the UML can be captured. For example, while class and object models (diagrams) map naturally to MOF models (HUTN specifications), UML activities must be considered in their repository model notation [2, 27]. 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 not lossless and the predominantly structural repository perspective misses the process flow metaphor which might have been identified as critical for the target domain (see D1 in Section 2.1). A comparable, vendor-specific notation is offered by TextUML [4].

At another point of the notational spectrum, non-standard, formally specified (i.e., grammar-based) textual notations explicity targeting (subsets of) the UML abstract syntax have been defined. For example, for a subset of UML activities (action nodes, control flows, control nodes), the Activity Diagram Linear Form (ADLF; [8]) provides a textual representation (and parser infrastructure) based on a Yacc grammar specification. Similar text-based but feature-wise incomplete forms for other UML metamodel fragments have been proposed (see, e.g., [12] for an example of UML state machines). A major limitation of these 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 [1]) or intermediate textual notations (e.g., yUML [11]) 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).

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 longer learning curve but might be used to express models in a shorter and—for the advanced user—in a faster way. Nevertheless, it is often not the best way to get an overview (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 processes, loops, and sequences.

Domain-specific application: UML stereotypes (O4.2) are the native visual presentation option of the UML. Software engineers may be familiar with textual syntaxes (O4.4). Eclipse MDT provides a tree-based view (O4.4) in one of its standard UML model editors. No explicit concrete syntax (O4.6) might be necessary if the DSML only defines language model constraints, limited behavioral specifications, or provides tool support for standard UML means.

Modeling support: A textual concrete syntax (O4.4) can be processed by a parser and does not need specific editor tools (as it is required for a graphical

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 (O4.2) as well as for tree-based syntaxes exists in standard tools, but must be explicitly integrated for new graphical elements (O4.2).

Consequences The DSML syntax is especially important from the DSML user perspective. If a DSML is mainly used by non-programmers, a special focus on usability aspects is needed. After defining suitable graphical and/or textual notation symbols as well as composition and production roles, 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 platform.

Application In our case studies we provide a couple of different concrete syntax definitions: model annotations (P5), UML stereotypes (P1, P3, P7, P9, P10), new graphical modeling elements (P1–P5, P7–P10), and a textual syntax (P9). We make use of the tree-based syntax provided by Eclipse MDT in P5 and P7. Additionally, one of our DSMLs applies extended language model constraints and does not need a concrete syntax (P6). In our DSML projects we have not encountered an application domain requiring the adoption of a form-/table-based syntax, so far.

Example Fig. 2 shows an example of two concrete syntax definitions consisting of a graphical representation on the left hand side and its textual equivalent on the right hand side (taken from P9). In the example, an audit rule is specified for an information system which records data when a successful login attempt from a user with administrator privileges is recognized (see [16] for details). Both syntaxes operate on the same abstraction level and can be used complementary.

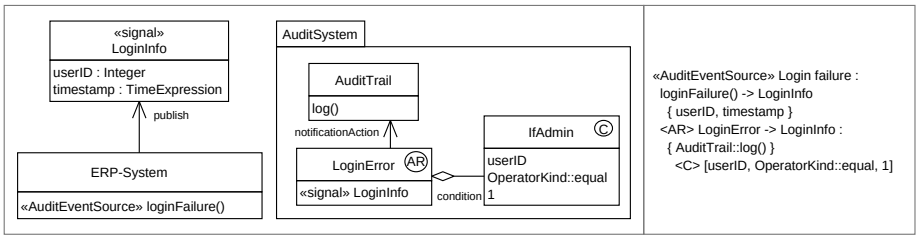

Figure 2: Exemplary graphical and textual concrete syntax [16].

2.5 D5 Platform Integration

Decision *How should the DSML artifacts be mapped to a software platform?*

Context The DSML has been developed and manifests in various artifacts. These are the core (i.e., formalized) language model as well as the set of structural and behavioral constraints. Optionally, a concrete syntax specification is available. 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 corresponding platform artifacts (e.g., source code and execution specifications).

Platform integration is achieved by defining model transformations (see, e.g., [22]) to convert a model into another platform-specific model (model-to-model transformation, M2M) or into executable software artifacts (model-to-text transformation, M2T). Alternatively, DSML models can also be evaluated and executed without any intermediate transformations.

Options

O5.1 Intermediate model representation: Based on a DSML model (i.e., the source mode), a second and intermediate model (i.e., the target model) is generated in a M2M transformation step. This intermediate model can be described via an own metamodel. The source model and target model are separate model entities. From the resulting intermediate model, platform-specific artificats (models) are then created (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 [6]. More specifically, the intermediate model can act as a decorator and/or as an adapter.

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

An adapter model does not preserve links back to the source DSML model and 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 [6]. 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 [21]).

O5.2 Generation templates: M2T transformation is achieved by taking model instances as the input to a transformation template for generating execution specifications (e.g., markup documents) and/or source code. Templates access input model data via metamodel-based selections and extraction expressions (e.g., OCL, XPath) and integrate the extracted model data into opaque output strings representing code fragments. Examples are Eclipse-based Xpand or the Epsilon Generation Language (EGL).

O5.3 API-based generators: The DSML model is 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 [6] or a mixin-based API [41]), code generation is achieved by instrumenting this API (e.g., implementing visitors or mixins) to travel the object-based DSML model representation and to serialize the model data to an output string [36]. The resulting platform-specific code fragments are independent from the generator language or the generator implementation.

O5.4 Direct model execution: 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 [6] or mixins [41]), 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).

O5.5 Combination of options: Template-based (O5.2), generator-driven (O5.3), and model-interpreting platform integration (O5.4) can be combined with intermediate structures (O5.1) 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 [9], intermediate models (O5.1) can be instantiated from metamodels reflecting the targeted platform host language (e.g., JetBrains MPS/Java). The model-represented language can optionally offer custom extensions relevant to the target domain. 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 by targeting the language runtime of the model-represented platform language (see O5.4; e.g., for prototyping and debugging purposes). Additionally, source code artifacts can be generated (O5.2) to maintain the code base separately (e.g., for deployment purposes).

O5.6 None: No platform integration is performed; e.g., the DSML serves only for documentation purposes, for sketching a software design, or for analysing requirements.

Drivers

Multiple target platforms: An intermediate model (O5.1) 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 much 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 [21].

Static code fragments: With an API-based generator (O5.3), the code independent from the DSML model must be integrated with the generator implementation (e.g., a custom visitor). When using generation templates (O5.2), non-changeable and non-parametric code fragments can be clearly separated from generator statements in templates [36]. Depending on the relative amount of static code fragments, an API-based generator (O5.3) involves extra maintenance effort for managing the interwoven fragments of generative code and static code.

Modeling only: If the DSML should only serve modeling purposes—for example, with the definition of a UML profile (O2.2) and the utilization of a standard modeling editor (O6.1)—no explicit platform integration might be needed (O5.6). In this case, the DSML is not meant to be executed on a software platform. However, the DSML might serve as a communication instrument between domain experts based on a well-defined language model definition, on a formalization and constraints, as well as on a concrete syntax to be used by the

domain experts.

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 the layout of generated code over time), and platform extensions. The latter are functional additions to the target software platform to cover DSML-specific execution requirements (e.g., through framework extension or integration of auxiliary frameworks).

Constraint consistency: If the PIM-to-PSM model transformations are not done directly, but via an intermediate model representation, it has to be ensured that the language model constraints (O3.1) also hold in the intermediate model. 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.

Constraint enforcement: Code generation templates (O5.2) are applied over instances of the language model. Therefore, explicit constraints enforced on the language model (O3.1) can also be checked on the generator templates (e.g., with EVL, O3.3). But this only prevents wrong usage of the construction rules in the code templates. As no constraints are enforced on the generated code, it must not conform to the expressed model-level constraints. 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 (O5.2). In P5, no platform integration has been performed (O5.6) 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 (O5.3) for intermediate models (O5.1) in P7 (a combined option, O5.4). 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) based on direct model interpretation (O5.4) is prepared in P1–P4, P6, P8, and P10. Based on a model representation und model runtime environment, implemented in a DSL toolkit comparable to the one in [41], object-oriented DSML model representations can be created and inspected. For platform integration, these representations could be instrumented for model execution (O5.4) as future work.

Example The following EGL code shows an excerpt from a M2T generation template applied in P9. Here, a Java method is generated for a 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 by a generated if-clause. True is returned (see `passed`) when all condition checks passed successfully; false otherwise.

```
[% 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;
  }
[% } %]
```

3 Decision Dependencies

A decision option chosen at one decision point may influence options at the same or at a subsequent decision point (for example, a choice can favor, determine, or exclude following options). By reviewing our DSML projects (see Table 1) using the decision catalog from Section 2, we identified 11 decision dependencies within a single decision or between two decisions. Each dependency is denoted by a pairing of two affected decision options as summarized in Table 2.

[O2.1↔O3.4] Constraint limitations for class models: A class model defines a language model at the UML instance level (i.e., at the M1 layer, see [25]). This means, no metamodel is defined to reflect the domain space and, therefore, domain concepts 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.

[O2.5↔O3.1] 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 explicit constraint definitions. For instance, in P7 we define both, a UML metamodel extension and a profile definition to integrate with the SoaML specification. Hence, we define explicit constraint expressions as OCL invariants over both language model formalizations. Thus, both constraint definitions need to be maintained and held consistent.

[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., [27]). 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 into 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.

[O3.2↔O5.6] Specific host/platform language: If code annotations were used to express constraints over the core language model, the language and language runtime to perform the code statements would be needed—for instance, as part of the platform integration step. Consider Java expressions attached to an extended UML metamodel. A JVM is needed to evaluate these Java expressions and enforce them on the system-level.

[O1.4↔O2.1–O2.4] Language model formalization as refinement: If the domain description included MOF or UML diagrams, a stepwise transition into a UML-based core language model is facilitated. For example, certain M2M trans-

formations would not be needed to overcome certain impedance mismatches (e.g., as between an ER metamodel and a UML-based metamodel).

[O1.2↔O3.1] Shared expression foundations: Adopting certain formal textual (e.g., set-theoretical) representations affects the choice of a language (e.g., the OCL) for making 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 [29], set models and set algebras are a natural choice at the domain description 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).

[O3.3↔O5.6] Mandatory platform integration: Whether M2M/M2T template expressions are actually an option for defining language model constraints depends directly if we want to perform platform integration or not. Likewise, if the use of M2M/M2T templating 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 template suite avoids duplicated specification effort and keeping possibly 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, causing 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 (05.1), with the actual platform integration step (M2T code generation) being performed on the validated intermediate model.

[O2.1↔O4.1] Impossible diagram extensions: By deciding to define the core language model at the UML M1 level, extending the UML concrete syntax (within the UML framework) is not an option anymore. Rather, model annotations remain the only viable option. However, UML stereotypes can be equipped with dedicated icons which appear as full replacements for the standard notation of stereotyped elements and which can act as a limited diagrammatic syntax extension.

[O2.1↔O3.1] M1 models as constraints: M1 behavioral models can be attached to metamodel elements for behavioral specifications (e.g., UML `ownedBehavior` relation of a `BehavioredClassifier`). In doing so, they are constraining/defining the behavior of metamodel elements. For example, in P3 we make use of a UML state machine to define possible states (e.g., passive, pending, discharged) and transition options between those states for DSML elements.

[O4.6↔O2.2] Symbol ambiguity in diagrams: When simply reusing existing UML symbols, the resulting "extended" diagrams are ambiguous: 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 [27]—however, without adding to the abstract syntax and semantics of the language model. In P7, 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`».

[O2.2–O2.4↔O4.1–O4.7] Concrete syntax drives UML extension: The formalization strategy for the language model influences the selection of a concrete syntax style. If the language model is defined via a UML profile (O2.2), different presentation styles for stereotypes may be considered. A textual presentation (i.e., tags) do not extend the basic UML symbol vocabulary. 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 a frequently adopted option. The different combined diagrammatic and non-diagrammatic options are also applicable.

References

[1] AT&T Research. Graphviz – Graph Visualization Software. Available at: `http://www.graphviz.org`, 2012.

[2] C. Bock. UML 2 Activity and Action Models. *Journal of Object Technology*, 2(4):43–53, July/August 2003.

[3] 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`, 2008. IBM.

[4] R. Chaves. TextUML Toolkit. Available at: `http://abstratt.com/textuml/`, 2012.

[5] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, 2011.

[6] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proc. of the OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, California, USA, 2003.

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

[8] David Flater, Philippe A. Martin, and Michelle L. Crane. Rendering UML Activity Diagrams as Human-Readable Text. In *Proceedings of the 2009 International Conference on Information and Knowledge Engineering*, pages 207–213, 2009.

[9] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Available at: `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[10] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Text-based Modeling. In *Proceedings of the 4th International Workshop on Software Language Engineering (ateM 2007)*, number 4 in Informatik-Bericht. Johannes-Gutenberg-Universität Mainz, 2007.

[11] T. Harris. yUML. Available at: `http://yuml.me`, 2012.

[12] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag, 2009.

[13] Bernhard Hoisl and Stefan Sobernig. Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models. In *Proc. of the International Workshop on Security Aspects of Process-aware Information Systems (SAPAIS)*. IEEE, 2011.

[14] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. Modeling and Enforcing Secure Object Flows in Process-driven SOAs: An Integrated Model-driven Approach. *Software and Systems Modeling*, forthcoming.

[15] Bernhard Hoisl and Mark Strembeck. Modeling Support for Confidentiality and Integrity of Object Flows in Activity Models. In *Proc. of the 14th International Conference on Business Information Systems (BIS)*, pages 278–289. Springer, LNBIP, 2011.

[16] Bernhard Hoisl and Mark Strembeck. A UML Extension for the Model-driven Specification of Audit Rules. In *Proc. of the 2nd International Workshop on Information Systems Security Engineering (WISSE'12)*. Springer, LNBIP, 2012.

[17] Bruce C. Hungerford, Alan R. Hevner, and Rosann W. Collins. Reviewing Software Diagrams: A Cognitive Study. *IEEE Transactions on Software Engineering*, 30:82–96, 2004.

[18] 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.

[19] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[20] Beate List and Birgit Korherr. An Evaluation of Conceptual Business Process Modelling Languages. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06), Dijon, France*, pages 1532–1539, New York, NY, USA, 2006. ACM.

[21] J. Mendling, K. B. Lassen, and U. Zdun. On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages. *International Journal of Business Process Integration and Management*, 3(2):96–108, 2008.

[22] T. Mens and P. van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[23] Daniel Moody and Jos van Hillegersberg. Evaluating the Visual Syntax of UML : An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams. In Dragan Gaševic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2009.

[24] Object Management Group. Human-Usable Textual Notation (HUTN) – Version 1.0. Available at: `http://www.omg.org/spec/HUTN/`, 2004. Version 1.0, formal/2004-08-01.

[25] Object Management Group. OMG Meta Object Facility (MOF) Core Specification – Version 2.4.1. Available at: `http://www.omg.org/spec/MOF`, 2011.

[26] Object Management Group. OMG MOF 2 XMI Mapping Specification. Available at: `http://www.omg.org/spec/XMI`, August 2011. Version 2.4.1, formal/2011-08-09.

[27] Object Management Group. OMG Unified Modeling Language (OMG UML): Superstructure. Available at: `http://www.omg.org/spec/UML`, August 2011. Version 2.4.1, formal/2011-08-06.

[28] Object Management Group. Service oriented architecture Modeling Language (SoaML) – Version 1.0. Available at: `http://www.omg.org/spec/SoaML`, 2012.

[29] Mark Richters and Martin Gogolla. OCL: Syntax, Semantics, and Tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL*, volume 2263 of *Lecture Notes in Computer Science*, pages 447–450. Springer Berlin / Heidelberg, 2002.

[30] Nick Russell, Wil M. P. Aalst, Arthur H. M. Ter Hofstede, and Petia Wohed. On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In *In Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006)*, volume 53 of *Conferences in Research and Practice in Information Technology*, pages 95–104. Australian Computer Society, 2006.

[31] Sigrid Schefer. Consistency Checks for Duties in Extended UML2 Activity Models. In *Proc. of the International Workshop on Security Aspects of Process-aware Information Systems (SAPAIS)*. IEEE, 2011.

[32] Sigrid Schefer and Mark Strembeck. Modeling Process-Related Duties with Extended UML Activity and Interaction Diagrams. In *Proc. of the International Workshop on Flexible Workflows in Distributed Systems*, 2011.

[33] Sigrid Schefer and Mark Strembeck. Modeling Support for Delegating Roles, Tasks, and Duties in a Process-Related RBAC Context. In *Proc. of the International Workshop on Information Systems Security Engineering (WISSE)*. Springer, LNBIP, 2011.

[34] Sigrid Schefer-Wenzl and Mark Strembeck. An Approach for Consistent Delegation in Process-Aware Information Systems. In *Proc. of the 15th International Conference on Business Information Systems (BIS)*, May 2012.

[35] Sigrid Schefer-Wenzl and Mark Strembeck. Modeling Context-Aware RBAC Models for Business Processes in Ubiquitous Computing Environments. In *Proc. of the 3rd International Conference on Mobile, Ubiquitous and Intelligent Computing*, 2012.

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

[37] M. Strembeck and U. Zdun. An Approach for the Systematic Development of Domain-Specific Languages. *Software: Practice and Experience (SP&E)*, 39(15):1253–1292, 2009.

[38] Mark Strembeck and Jan Mendling. Modeling Process-related RBAC Models with Extended UML Activity Models. *Information and Software Technology*, 53(5):456–483, 2010.

[39] Mark Strembeck and U Zdun. Modeling Interdependent Concern Behavior using Extended Activity Models. *Journal of Object Technology*, 7(6):143–166, 2008.

[40] Mark Strembeck and Uwe Zdun. An Approach for the Systematic Development of Domain-Specific Languages. *Software: Practice and Experience (SP&E)*, 39(15):1253–1292, 2009.

[41] U. Zdun. A DSL Toolkit for Deferring Architectural Decisions in DSL-based Software Design. *Information and Software Technology*, 52(9):733–748, 2010.

[42] U. Zdun and M. Strembeck. Modeling Composition in Dynamic Programming Environments with Model Transformations. In *Proc. of the 5th International Symposium on Software Composition*. LNCS, Vol. 4089, Springer, 2006.

[43] U. Zdun and M. Strembeck. Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Projects. In *Proc. of the 14th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 1–36, July 2009.