This is the authors' post-print of the manuscript titled "Reusable and generic design decisions for developing UML-based domain-specific languages" accepted for publication by "Information and Software Technology". The publisher's version is available at https://doi.org/10.1016/j.infsof.2017.07.008.

(C) 2017. This manuscript version is made available under the CC-BY-NC-ND 4.0 license http://creativecommons.org/licenses/by-nc-nd/4.0/.

Reusable and Generic Design Decisions for Developing UML-based Domain-specific Languages

Bernhard Hoisl[a], Stefan Sobernig[a,*], Mark Strembeck[a,b,c]

[a]*Vienna University of Economics and Business (WU), Welthandelsplatz 1, 1020 Vienna, Austria*
[b]*Secure Business Austria (SBA) Research gGmbH, Favoritenstraße 16, 1040 Vienna, Austria*
[c]*Complexity Science Hub Vienna (CSH), Josefstädter Straße 39, 1080 Vienna, Austria*

Abstract

Context: In recent years, UML-based domain-specific model languages (DSMLs) have become a popular option in model-driven development projects. However, making informed design decisions for such DSMLs involves a large number of non-trivial and inter-related options. These options concern the language-model specification, UML extension techniques, concrete-syntax language design, and modeling-tool support.
Objective: In order to make the corresponding knowledge on design decisions reusable, proven design rationale from existing DSML projects must be collected, systematized, and documented using an agreed upon documentation format.
Method: We applied a sequential multi-method approach to identify and to document reusable design decisions for UML-based DSMLs. The approach included a Web-based survey with 80 participants. Moreover, 80 DSML projects[☆], which have been identified through a prior systematic literature review, were analyzed in detail in order to identify reusable design decisions for such DSMLs.
Results: We present insights on the current state of practice in documenting UML-based DSMLs (e.g., perceived barriers, documentation techniques, reuse potential) and a publicly available collection of reusable design decisions, including 35 decision options on different DSML development concerns (especially concerning the language model, concrete-syntax language design, and modeling tools). The reusable design decisions are documented using a structured documentation format (*decision record*).
Conclusion: Our results are both, scientifically relevant (e.g. for design-space analyses or for creating classification schemas for further research on UML-based DSML development) *and* important for actual software engineering projects (e.g. by providing best-practice guidelines and pointers to common pitfalls).

Keywords: model-driven software development, domain-specific language, design decision, design rationale, Unified Modeling Language, survey

1. Introduction

In model-driven development (MDD), a domain-specific modeling language (DSML) is a domain-specific language (DSL) for specifying design-level and platform-independent concerns in the target domain, rather than implementation-level concerns (see, e.g., [1]). In this context, DSMLs typically provide (but are not limited to) a graphical concrete syntax. A DSML is built on top of a tailored *abstract syntax* (i.e. the core language model) which is typically defined using metamodeling techniques. In addition to a DSML's abstract syntax (metamodel), DSML developers often use formal textual specification techniques to express the DSML's structural and behavioral semantics [2]. Once the abstract syntax and a corresponding concrete syntax are specified, a DSML is typically integrated into an MDD tool chain, such as the Eclipse Modeling Framework (EMF).

In recent years, the development of DSMLs based on the Unified Modeling Language (UML [3]) and/or on the Meta Object Facility (MOF [4]) has become a popular choice among software engineers: In a related survey, we found that more than 50% of the participating MDD researchers and practitioners have contributed to at least one UML-based DSML between 2000 and 2015 [5]. In addition to our own findings, the UML's relevance for DSML development is also reported in numerous other contributions (see, e.g., [6, 7, 8, 9, 10]). On the one hand, this momentum is due to a general trend towards the usage of DSLs in MDD [11]. On the other hand, the UML and the MOF provide native extension techniques for a) developing fully customized modeling languages (e.g., new diagram types) and b) for adapting the UML to domain-specific purposes while reusing UML features. Examples of such techniques include UML profiles [9, 12], pruning/reduction [13], metamodel slic-

[☆]Note that it is pure coincidence that there were 80 participants in the survey and that 80 DSML projects were reviewed.

[*]Corresponding author
Email addresses: bernhard.hoisl@wu.ac.at (Bernhard Hoisl), stefan.sobernig@wu.ac.at (Stefan Sobernig), mark.strembeck@wu.ac.at (Mark Strembeck)

ing [14], or package referencing and merging [15, 16]. In this paper, we focus on DSMLs that are based on and embedded into the UML.

Design Knowledge for Reuse. At the time of writing, experiences and lessons learned from developing UML-based DSMLs in a disciplined manner are barely documented. Furthermore, even when documented, the level of detail necessary to become useful to other DSML developers is often missing. In recent years, different research approaches and research methods have been applied to collect, organize, and review current (best and worst) practices, such as case studies (see, e.g., [10]), controlled experiments (see, e.g., [17]), critical-analytical studies based on a reference theory (see, e.g., [18]), and systematic literature reviews (SLRs; see, e.g., [19]). So far, these contributions focused on isolated elements of a DSML design (e.g., on the concrete syntax for improving its cognitive effectiveness or on patterns of structuring the abstract syntax). However, design-decision making for DSML development includes multiple, interrelated decisions on language-model definition, constraint specification, concrete-syntax design, platform integration [20], and adequate software tooling [21]. To be useful, a design decision must be captured along with the *rationale* on why this particular decision is important or meaningful. Rationale details include a) different solutions (so called *decision options*) that should be considered before making a final design decision, b) the decision-makers' positive and negative assessments of the respective options given a set of corresponding DSML requirements (so called *decision drivers*), and c) the positive and negative effects on subsequent design-decision making (the *decision consequences*).

The lack of documented design-decision rationale in software engineering is sometimes referred to as the *capture problem* of design-rationale documentation [22, 23]. An important barrier to documenting design rationale (DR) in all necessary detail is the considerable overhead of creating and maintaining DR documentation. For example, a study on capturing architectural design knowledge quantified the time effort needed for a project *including* capturing design rationale to be *twice* the time needed for a project *without* that extra effort [24]. Other problems explored in the research on documenting DR include the intrusiveness of documentation techniques, lack of incentives, and cognitive barriers in software-design processes (see, e.g., [23, 25, 26]). As a consequence, new DSML development projects cannot benefit from the experiences gained in prior projects and valuable design knowledge might be lost [27].

In this context, two important objectives in software-engineering research are to *limit the effort* for documenting design decisions and to *increase the quality* of the documented rationale [25, 22, 23]. To achieve these objectives, existing documentation approaches distill common or reusable knowledge—similar to software patterns [27]—from decisions made in actual development projects to document and share proven solutions along with their forces, consequences, and (alternative) solutions (see, e.g., [28, 29, 30, 31]). For developing DSLs (including DSMLs) prior work has started by gathering DR and best practices. Results include procedural models on systematic DSL development (see, e.g., [20, 32]) as well as pattern collections (see, e.g., [33, 34]). However, so far the empirical evidence gathered from corresponding UML-related projects is limited to the UML core, for example as reported in a review of 49 empirical studies [35]. Moreover, only tentative results exist on applying DR and best practices in DSML development projects (see, e.g., [36, 37]). In this context, our work complements existing approaches by documenting reusable design knowledge for developing UML-based DSMLs.

Synopsis. Our study for capturing reusable and generic design decisions includes three consecutive stages (see Figure 1). In particular, we started by documenting our own DSMLs, reviewed DR found in the related work via backward snowballing, and conducted an initial pilot SLR (see [38, 39]). The result of these preparatory studies was a first revision of the catalog of reusable design decisions [40]. Next, we designed, conducted, and documented an SLR (see [21, 41]) to arrive at a revised version of our decision catalog (post-study revision [42]). Finally, we performed a Web-based survey with 80 MDD researchers and practitioners on documenting and reusing DR. The survey's main objective was to collect data to decide on a design-decision documentation format (see [5, 42]).

Contribution. The results of our Web-based survey show the potential for DR reuse on UML-based DSMLs, as well as the perceived barriers for documenting DR in DSML development projects. For example, missing standards or requirements for documenting design decisions, time and budget constraints, absence of documentation-tool support, or the lack of prior design decisions for reuse (see above). A majority of the survey participants confirmed the importance of using DR as part of DSML design documentation. This importance extends to all forms of DR, whether self-documented (e.g. via DR documentation activities, such as meeting/interview protocols, participant observations, or written documentation) or reused from other sources (such as scientific publications, guidelines, or pattern collections).

In a long-term research effort of about three years [38, 41, 21], we have collected, documented, and systematized design rationale from 80 UML-based DSMLs. The key result is a publicly available catalog of reusable design decisions [42]. The *decision catalog* consists of seven reusable design decisions (*decision records*), each describing a repeatedly observed *decision context* (e.g., a development phase or certain technology choices), a repeatedly reported *design problem* regarding a DSML design element, as well as corresponding *design options* to solve the problem. In total, the catalog documents 35 decision options. In addition, the reusable design decisions also report on the inter-dependencies between different reusable decisions (e.g., between designing the abstract syntax and the concrete syntax of a DSML).

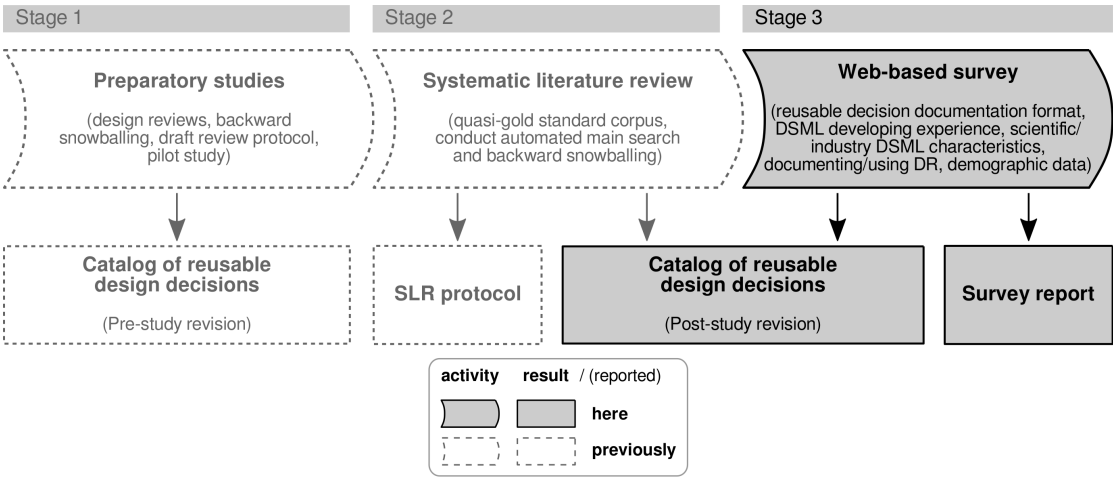

Figure 1: Overview of research stages for collecting, documenting, systematizing design rationale (DR) on UML-based DSMLs. In this paper, we report on the design and the execution of the research activities at Stage 3 (web-based survey), as well as the corresponding research results: i) the survey and ii) the catalog of reusable design decisions.

The reusable decisions and their details are defined in a way such that they can be directly referenced from other documents (e.g., decision templates). This way, the design-decisions catalog aims at providing a practical means for assisting DSML engineers in reusing DR from prior projects and in documenting the rationale behind their own decision making.

This paper is accompanied by (publicly available) supplemental material: 1) a detailed technical report on the survey design and survey results [5], 2) a complete design-decisions catalog [42], 3) an earlier systematic literature review (SLR) which is documented via a prior publication [41].

The remainder of this paper is structured as follows: In Section 2, we elaborate on DR documentation for UML-based DSMLs, on the content structure of the design-decisions catalog, and on the origins of the empirical data which entered the construction of the catalog (survey, literature review). Section 3 gives a motivating application example for the presented design-decisions catalog. Section 4 is dedicated to a selective presentation of the catalog's contents, including design-decision options, decision drivers, and associations between different decisions. The limitations of our overall approach (esp. the survey) are discussed in Section 5. In Section 6, we iterate over the requirements on DSML DR documentation as indicated by the survey. Related work is discussed in Section 7 before Section 8 concludes the paper.

2. Background and Preliminaries

2.1. Documenting Design Rationale on DSMLs

A domain-specific language (DSL) is a tailor-made software language for a (narrow) application domain. Thus, a DSL is based on corresponding domain abstractions and provides at least one concrete syntax. A domain-specific *modeling* language (DSML) is a DSL that provides a graphical concrete syntax for the primary purpose of diagrammatic modeling in a particular application domain [20, 32]. A DSML is commonly deployed as part of a model-driven development (MDD) toolkit (e.g. as part of the Eclipse Modeling Framework, EMF). For the scope of this paper, we look at DSMLs which are *internal* to or *embedded* into the Unified Modeling Language version 2.x (UML 2.x [9, 12, 43]).

DR [22, 23] on DSML development includes reasoning and justification of decisions made when designing, creating, and using the core artifacts of a DSML (e.g. the abstract and concrete syntax, behavior specification, metamodeling infrastructure, or the MDD tool chain). Documenting DR explicitly aims at assisting software engineers by providing and explaining past decisions (e.g. in a design-space analysis) and by improving the understanding of a particular DSML design choice during development and maintenance (e.g. as a kind of design-process documentation).

For the purposes of this paper, we distinguish two kinds of DSML DR [27]: 1) DSML-*specific* DR reflects on the reasoning over different decision options during a particular design process for a single DSML. Examples of such explicitly documented, specific DR may be found in artifacts created in source-configuration management tools, development-issue trackers, and open-standards artifacts. 2) DSML-*generic* DR includes knowledge obtained through developing multiple DSMLs, for one or several application domains. Generic DR is commonly found only as implicit knowledge of experienced DSML engineers. For example, software patterns have been used in software-language engineering to explicitly document generic DR (see, e.g., [33, 44]).

In general, the DSML development process involves a number of characteristic development activities [20]. From a decision-making perspective, each development activity also marks a *decision point*, i.e. a point in time at which particular design-decisions must be addressed. In particular, this means that different design solutions as well as their effects on subsequent design decisions have to be assessed. From the DR documentation perspective, a decision point is a point in time for recording an on-going decision-making process.

In our study, DR on a given decision point is captured from multiple DSML projects and represented as a reusable option for decision making (see Section 2.2).[1] In particular, we consider seven concerns of UML-based DSML development and, therefore, seven reusable decisions (D1–D7, hereafter; see Section 4).

Survey Design. In order to assess the importance of DR for the development of DSMLs, we conducted a Web-based survey with researchers and practitioners (see also Section 6) [5]. The target population were peers in the field of designing and developing scientific/industry DSMLs. We applied a non-probabilistic sampling method by contacting MDD researchers and practitioners identified via dedicated scientific venues (e.g. authors of research papers, program committee members of conferences, associate editors of journals) to take part in the survey (i.e. convenience sampling [45]). Venues included premier outlets for researchers and practitioners in the field of MDD and DSMLs, such as, the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS)[2] or the International Journal on Software and Systems Modeling (SoSyM)[3]. The prospective participants were invited to take part in the Web-based survey via email. Prior to sending out the invitation emails, we pretested our questionnaire and adjusted the content as well as its length so that it took approx. 15–20 minutes for the participants to complete it.

The questionnaire was divided into four main parts:

1. An introductory text and an agreement concerning the participation in the survey;
2. Questions concerning the participants' experiences with developing DSMLs (e.g. number of developed scientific/industry DSMLs, job description while contributing to the DSMLs);
3. Questions about characteristics of the developed DSMLs (e.g. application domains, metamodeling languages) and about aspects of documenting and using DR when developing these DSMLs (e.g. DR documentation activities/barriers, DR reuse);
4. Demographic questions (e.g. country of residence);

Moreover, the participants were asked to indicate whether they like to receive a copy of the research report as well as their availability for a possible follow-up survey. We also provided a text box for additional (optional) feedback.

For the purposes of our study, we defined an industry DSML as a language that has been developed as part of one or several predominantly industry-driven software-development projects with the primary aim to create or to improve a commercial software product. In contrast, we defined a scientific DSML as a language that has been developed as part of one or several predominantly research-driven software-development projects which result in non-commercial software artifacts (e.g. research prototypes, experiment materials). In this context, a research-driven software-development project aims at exploring, collecting, systematizing, and validating knowledge on software engineering, in general, and DSML engineering, in particular. The survey especially targeted practitioners developing (UML-based) industry DSMLs. In case participants reported experience with industry DSMLs, we specifically asked them to answer additional questions referring to industry DSMLs [5].

Demographics. We contacted 399 researchers and practitioners and received 62 completed as well as 18 partially filled-out questionnaires (sample size: n=80, including partial answers); the participants resided in 22 different countries (n=62, because some participants did not indicate their country of origin). This results in a response rate (Response Rate2; RR2 [46]) of 20.1%.[4] Similar response rates have been reported for related studies (see, e.g., [47]).

Regarding expertise, two thirds of the respondents (53/80) have contributed to more than three DSML projects. In UML-based projects, 40.5% (17/42) have contributed to more than three DSMLs. This also reveals a considerable potential for reuse of DSML design decisions from past projects. 75% of the respondents (52/70) have six or more years of experience (up to 29 years). Moreover, our participants gained experience in diverse roles (multiple answers were allowed): 86.1% of the respondents worked in a non-profit organization (e.g. publicly funded university), 36.1% in a for-profit organization (e.g. private company), and 5.6% as freelancer/independent contractor (other employer: 4.2%). The job description for 76.4% of the participants included research aspects (e.g. research associate), tertiary education (e.g. university lecturer) for 37.5%, and software development (e.g. software architect, developer, tester) for 31.9% (other description: 5.6%). When the job description included software development, most of the participants filled one or more of the following roles in software-development projects involving DSMLs (in descending order): software architect/designer (87%), software developer/implementer (70%), project manager (56.5%), system analyst/requirements specifier (34.8%).

General Findings. In total, our survey participants contributed to 365 industry (per-participant mean: 4.6, sd: 15.7) and to 390 scientific (4.9/ 11.6) DSMLs (time frame: 1987–2015). Out of these, 67 (0.8/ 1.3) and 101 (1.3/ 2.1), respectively, were based on the UML. Note that these are only coarse aggregates across the participants which do not consider if different participants have been working on the

[1] Throughout the paper, we apply some notation conventions to refer to reusable design decisions and their content items such as decision options. D*i* denotes a reusable decision corresponding to some decision point *i*; O*i*.*j* refers to decision option *j* at decision point *i*.

[2] http://cruise.eecs.uottawa.ca/models2015/; last accessed: Feb 9, 2017.

[3] http://sosym.org/; last accessed: Feb 9, 2017

[4] For the definition of partial answers as well as more details on the outcome rates (e.g. response rate), please consult the survey report [5].

same DSML projects – 52.5% of the respondents (42/80) have contributed to at least one UML-based DSML, 40% (32/80) developed at least one UML-based DSML in an industrial setting.

The DSMLs created by our participants target diverse domains, such as software development techniques (reported by 44.3% of the participants; n=70), embedded systems (38.6%), model verification and validation (30%), or web applications (27.1%). Most of the participants used Ecore (a technology projection of the EMOF [4]) as metamodeling language to develop DSMLs (62.9%); followed by the MOF in versions 2.x (27.1%) and 1.4 (15.7%), respectively (n=70). These figures correspond to the result that 65.7% of the participants (n=70) employed an Eclipse-based MDD tool chain to integrate their DSMLs. All participants who developed UML-based DSMLs (n=42) used UML in version 2.x [3]; for example, 33.3% of the participants adopted UML in version 2.0 and 23.8% in version 2.4.1.

The majority of our participants (72.1%, sample size: n=68, due to partial answers) also believe that it is (extremely) important to use DR as part of DSML design documentation (a finding that is confirmed by related studies; see, e.g., [47]). Almost all survey participants (93.4%, n=61) (re)used DR available from arbitrary sources and documented in arbitrary formats (e.g. books, scientific publications, case-study reports) for making design decisions on at least one of their DSMLs. However, the participants also reported a limited usage of DR that has been explicitly documented in a generic and reusable form (e.g. pattern collections, design decisions).[5] For instance, only 59% of the participants (sample size: n=61) (re)used DR documented as design decisions (e.g. available from former projects), although 75% of them rated documented design decisions as being moderately to extremely useful (n=36). The above reasons serve as a strong motivation for compiling a catalog of generic, reusable design decisions.

2.2. Structure of Reusable Design Decisions

A *reusable design decision* documents two or more proven solutions, i.e. solutions that have been successfully applied to a generic and recurring problem in DSML development. Moreover, the problem described by a reusable decision must not only recur, i.e. be observable for many DSML development projects, but it must also have the quality of explicitly requiring a design-decision. However, no generally accepted format and notation for reusable design decisions exist [48]. Thus, the format we use in our work is based on existing proposals and represents a simplified and common core for documenting reusable design decisions, which can be extended to include additional documentation elements if required.

To systematically arrive at a suitable documentation format, we performed three steps: 1) we drafted a documentation format based on the state of the art in (architectural) design-decision documentation [48, 49] and based on our DSML documentation needs [38].[6] 2) based on the DR data obtained from a prior systematic literature review (see Section 2.3), we verified whether we could collect and distill actual decision data to populate all decision details. For example, stakeholder roles and decision-based actions cannot be extracted using a literature review alone. 3) in order to include an external assessment, we collected expert opinions on decision details deemed relevant for UML-based DSMLs ([5]; see also Section 2.1).

The survey data was used to confirm the importance of the content elements included in reusable design decisions. For example, we excluded the three elements that have been rated the least important ones in our survey: viewpoints (rated important to extremely important by 52.7%, sample size: n=55), stakeholders (47.2%, n=53), and status (32.1%, n=53). This way, we arrived at the format depicted on the left-hand side of Figure 2. This overview also highlights the ratings on each of the selected content elements, collected during step three (collection of expert opinions, see above).

The resulting documentation format for reusable decisions is divided into seven sections: Point, Problem, Driver, Consequence, Option, Application, and Sketch (see Figure 2). In the following, we introduce each section by referring to examples taken from an actual reusable design decision contained in our decision catalog (see Section 4.4 for the details).

A reusable design decision first describes a recurring design-*decision problem* that has been repeatedly observed for several DSML development projects. Our survey confirmed the importance of a problem section as 69.6% of the participants (56/80) rated it as (extremely) important. An exemplary problem statement frequently observed when deciding on the concrete-syntax style for a DSML is: "In which representation should the domain modeler create models using the DSML?".

This problem applies to a specific *decision context*. The decision context is, for example, established by one of the decision points characteristic for DSML development (e.g., decision making on language model, concrete syntax, and tooling). At each decision point, a particular DSML design concern (e.g., language-model definition, concrete-syntax styles) must be tackled. The majority of our survey participants rated an explicit context section as important to extremely important (56.4%). Decision points (concerns) can be addressed in varying order, with different typical orders denoting different DSML development styles used and the intention behind developing the DSML [20]. Besides, a particular metamodeling toolkit (e.g. MOF [4]), the application domain modeled by a DSML, and the corresponding software platform can contribute to establishing the decision context.

[5] For a discussion on the relation between architectural patterns and design-decision documentation for the process of software engineering see [27].

[6] See Section 3 for an example of a decision instance and how decision instances can reference reusable design decisions from the catalog.

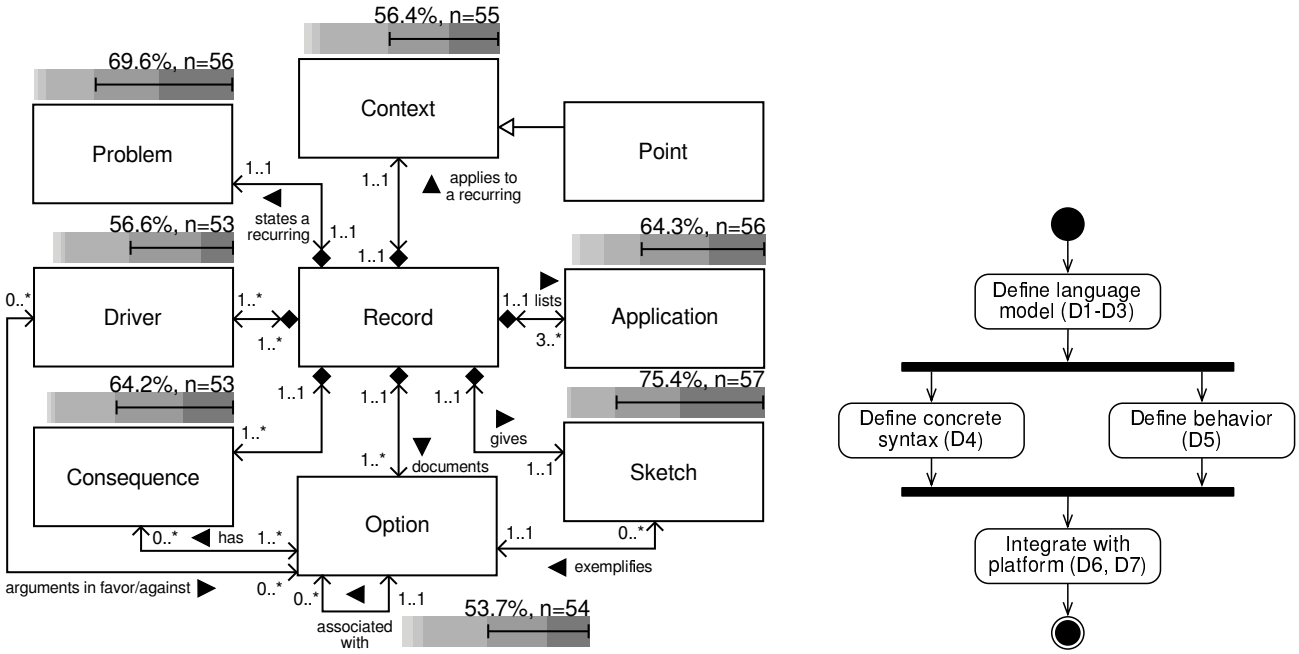

Figure 2: The *left-hand side* shows overview of the nine key concepts, their relationships, and their relative importance as rated by the survey participants (important to extremely important) [5]. The *right-hand side* depicts a typical flow in design-decision making in a "language-model-driven" DSML development style [20].

The main part of a reusable design decision is list of *decision options* (rated important to extremely important by 53.7% (n=54) of the respondents). Decision options describe proven solutions to the respective decision problem. For choosing a concrete-syntax, an exemplary option is *model annotation* (O4.1 in Section 4.4), which is about realizing a tailored concrete syntax by means of model annotations (e.g., via UML tags and structured comments). Furthermore, a reusable decision includes arguments in favor or against selecting a particular option (or a combination of options) in terms of *decision drivers*. Documenting drivers was considered important to extremely important by 56.6% of the respondents. An exemplary driver relevant for adopting the *model annotation* option is the cognitive expressiveness of the respective UML diagram elements. Drivers like these are likely to steer the DSML designer towards a particular option or combination of options. Moreover, the respective selection affects the solution space of subsequent decisions. For example, they can set a new decision context.

To scaffold follow-up decision making, a reusable decision makes the DSML designer aware of *decision consequences*. Documenting such recurring consequences was deemed important to extremely important by 64.2% of the survey respondents. Decision consequences can include the need to evaluate other decision options within the same reusable decision or in related reusable decisions, for example. However, consequences can also point to follow-up decision problems not covered by the design-decisions catalog alone.

To provide evidence that the different decision options are taken from observed practice, each reusable decision refers to example projects that implemented the respective option or option combination. This element of description was supported by 64.3% of the survey participants as important to extremely important. Each reusable decision is completed by replicating a concrete realization *sketch* of one decision option taken from an actual DSML project. Such sketches were rated important to extremely important by 75.4% of the participants.

The documentation format described above is capable of describing recurring *associations* between decision options in several ways (e.g., as decision drivers and decision consequences). Association types documented in the literature are, for example, different types of causal sequences between decisions [50]. A causal sequence groups decisions (resp. decision options) which are linked pairwise by depends-on, is-excluded-by, and/or relationships. Other relevant association types are: influences, refinedBy, decomposesInto, forces, isIncompatibleWith, isCompatibleWith, and triggers [28].

2.3. Material Corpus for Recovering DSML Design Rationale

In our previous work [21, 41, 51], we performed a systematic literature review (SLR) to collect and systematize a corpus of scientific publications on UML-based DSMLs and their companion material. This corpus contains 84 publications documenting *80* unique DSML designs, published in major MDD outlets (e.g. SoSyM, MoDELS). In addition, the SLR found 25 secondary studies on UML-based DSML development (e.g. [52, 53, 54]). To compile the catalog of design decisions (see Section 4), we performed a rigorous content analysis on the corresponding papers. This way, we identified and documented 35 reusable decision options for seven

decision points (D1–D7) (see Section 2.2).[7] In addition, the 80 DSMLs entered the catalog in terms of applications and solution sketches. This paper provides the first report on the resulting catalog of reusable design decisions (publicly available from [42]).[8]

2.4. Empirical Evidence on DR Reuse

Prior empirical research on DR reuse, involving software patterns and reusable design decisions [55, 56], has reported first evidence that it is possible to achieve the double objective of limiting the effort for documenting design decisions and of increasing documentation quality at the same time.

In two controlled experiments involving 171 software-architecture undergraduates, Lytra et al. [55] reported both an increase in effectiveness and in efficiency of design-decision making on two software architectures. In these experiments, the participants made and documented architecture-level decisions on predefined functional requirements for two software systems (an order-management system and a learning-management system) based on a design-documentation tool (CoCoADvISE) and collections of architectural patterns (16 and 40 pattern descriptions, respectively). The experiment groups were additionally equipped with five reusable architectural design-decision models which provided scaffolding for decision making and pre-structured references to the software-pattern descriptions. The experiment groups took less working time and documented more decisions than the control groups. Their decisions were also judged as being of higher quality by software-architecture experts.

In a project including one experiment and its replication, Heesch et al. [56] investigated the effect of software-pattern collections as reference material for recovering design decisions from a software architecture (JBoss J2EE application server). The experiments involved 34 participants of mixed proficiency in software patterns, with the experiment group being equipped with a pattern catalog (on remoting patterns such as BROKER). The participants documented the recovered decisions based on a predefined documentation template. Heesch et al. collected data on the number and the quality of recovered decisions, with quality being measured by ratings of independent software-architecture experts. Their findings indicate that the quality ratings obtained by the experiment groups (who have been working with the pattern catalog) were higher than those of the control groups. There was no significant increase in the number of recovered decisions though. However, the participants referencing the pattern catalog were more homogeneous with respect to the number of recovered decisions per participant.

Nevertheless, these findings do not originate from the field of UML-based DSML development and they suffer from a couple of limitations (e.g., missing third-party replications, emphasis on student or novice subjects). Nevertheless, such first evidence provides support for the basic claims regarding DR reuse (effort reduction, quality increase). In addition, the reusable DR material used in these experiments is comparable to our design-decision catalog. Therefore, these empirical studies strongly motivated our work.

3. Motivating Example: DSML Design-Process Documentation

Our catalog of reusable design decisions aims at collecting, systematizing, and documenting DSML design processes, comparable to the role of software patterns in documenting architectural design decisions [27]. The process of making design decisions is typically documented using structured text documents following an agreed upon format (document templates; see [48] for an overview). These structured text documents on design decisions and their details (alternatives, arguments) allow for *referencing* a collection of reusable and recurring design decisions. The objective is to reduce the time spent on these documentation tasks, by avoiding repetition. Consider an example taken from documenting PRDM [123, 124]—a DSML developed in support of model-driven role engineering in a business process context. By following a design-decision excerpt from this DSML, we show an example for design-decision making by using the decision catalog.

Select Development Style. At the beginning of a DSL project, we have to decide on the development style for this project [20]. A (tailored) development style can accommodate domain requirements (e.g. direct access to domain experts) and requirements of the overall software-development project (e.g. a software framework accessible via a DSML). For PRDM, a language-model-driven development style [20] was selected (see Figure 2, right-hand side). In this style, the (core) language model drives the subsequent activities in the sense that a draft model is defined and implemented early. Subsequently, it is continuously refined over a number of iterations. Once selected, the development style determines the order of subsequent decisions at specific decision points.

Identify Domain-specific Prototype Designs. Our design-decisions catalog documents different application areas of the collected DSMLs, based on the 2012 ACM Computing Classification System (CCS)[9]. Each DSML is assigned to one or several application areas (ACM CCS codes). The most frequently assigned CCS codes in our catalog, and relevant for PRDM, include security engineering (11), business-process modeling (10), and access control (7) [42]. Based on these categories, design-decision makers can approach the catalog

[7] Note that there are actually 40 decision codes/numbers. Five of those codes/numbers serve for coding pseudo-decision options; e.g., not taking any decision. Depending on the analysis requirements, they are either ignored or included as dedicated no-option codes.

[8] With the SLR being prior work, full details on the process of conducting the SLR are provided in earlier and companion publications (see [21, 41, 42]).

[9] http://www.acm.org/about/class; last accessed: Feb 9, 2017.

content in two ways: a) by reviewing prior DSML projects assigned to one or more relevant categories and/or b) by identifying so-called *DSML prototype designs* for the respective categories: Because reviewing prior DSMLs might incur substantial overhead in early iterations (for PRDM, this would have meant reviewing potentially 28 prior DSMLs), the catalog offers frequently recurring combinations of decisions, so called *prototype designs* [57]. At the time of writing, the catalog contained seven prototype designs. Each prototype design is characteristic for a significant subgroup (consisting of at least three DSMLs[10]) of the 80 DSMLs we examined in detail. Every prototype design includes the most frequently adopted design decisions (decision combinations) for these representative subgroups. Each of the seven prototype designs is also linked to the corresponding ACM CCS codes. This way, decision makers can consider a prototype design for given application areas, rather than having to review prior DSMLs. For PRDM, the corresponding prototype design is the combination of the following decision options: INFORMAL TEXTUAL DESCRIPTION (O1.1), PROFILE (RE-)/DEFINITION (O2.2), CONSTRAINT-LANGUAGE EXPRESSION (O3.1), MODEL ANNOTATION (O4.1), and DIAGRAM SYMBOL REUSE (O4.6; see Table 1 for a brief description). This combination of decisions was documented for 26 out of 80 DSMLs in the catalog.

Navigate Decision Associations. Starting from a prototype design, additional decisions will follow from the project-specific context and from unique project requirements. As for PRDM, for example, an earlier decision was to implement the DSML by extending the existing BusinessActivities framework [59] and the corresponding metamodel (see the decision reference `ExtendBusinessActivitiesFramework` in Table 2). This prompted PRDM to be realized as a metamodel extension (O2.3) combined with a metamodel modification (O2.4), adding to the UML profile (O2.2). Based on adopted decision options (either inspired by the prototype design or by project-specific factors), the DSML developer can study typical associations between decision options. For instance, for PRDM decision option PROFILE (RE-)/DEFINITION (O2.2), decision makers will find the association "constrained UML profiles" in our decision catalog.[11] This association provides rationale for capturing additional language-model constraints using OCL expressions (O3.1).

Document Design Decisions with Reuse. Table 2 shows an example of a structured document used to capture one particular decision on PRDM. The corresponding template [49] structures the document into predefined sections (e.g. name, status, problem statement, arguments leading to a decision). In order to annotate decision details within unstructured text fragments, dedicated mark-up elements («...») are provided. This way, the decision document and text elements in the document can be tagged to indicate a particular iteration in the decision-making process, the state of the decision, or stakeholder roles, for example.

Empirical work on generic design knowledge suggests that the effort of specifying such a decision document can be reduced by referencing reusable and generic decisions (see Section 2.4). In particular, such references allow for focusing on describing additional, DSML-specific decision knowledge only, rather than repeating what is described in the decisions catalog. A complete process documentation of a DSML design consists of a collection of such decision documents, which are interconnected by different relationship types (e.g. «`caused by`»).

4. A Catalog of Design Decisions for UML-based DSMLs

As mentioned above, we identified, described, and collected data on 35 decision options at seven different design-decision points (D1–D7, hereafter, see Figure 3). In this section, we present an overview of these reusable design decisions. Further details are provided in the publicly available decision catalog [42]. Note that the following overview is limited to decision options, decision drivers, and decision associations that we found for at least one DSML *and* that are also found in secondary studies on systematic DSML development. Decision details such as context, consequences, and applications are not reproduced in a structured manner (as in the catalog), but they are blended with the overview sections.

4.1. Language-Model Definition (D1)

One recurring design decision is whether or not one should define a platform-independent language model [20]. In general, a language model (also: abstract syntax) acts as a structured description of the captured domain (or domain fragments) and provides a domain definition, the domain vocabulary, as well as a catalog of domain abstractions and abstraction relations. It is platform independent in the sense of being independent from a particular implementation technique or software platform. In certain development styles, this can be the first decision point (see [20]). A prerequisite for defining a generic language model is a systematic analysis of the target domain. The process of analyzing the target domain includes collecting and evaluating relevant information (e.g., based on literature reviews, expert interviews, scenario descriptions, existing software systems) which provide input to generate a structured and technology-neutral description of the domain. The main challenge is how to document and how to organize the identified domain abstractions in order to arrive at a comprehensive and comprehensible language model. Figure 4 summarizes the key details of this decision point (D1).

[10] Here, we adopt a commonly followed rule of thumb from the software-pattern community. This rule mandates that a software-pattern description must provide at least three known uses of the pattern in existing software systems (see, e.g., [58]).

[11] The complete reusable decision on language-model formalization (D2) from [42], pp. 16–19, containing O2.2–O2.4 is available as an appendix to this paper.

Table 1: Thumbnail descriptions of selected (7 out of 35 total) decision options relevant for the discussion in Sections 2 and 3. See Section 4 for complete and comprehensive descriptions (incl. option details, decision drivers, and consequences).

	Problem statement	**Options (selected)**
D1	How should the domain (or domain fragment) be described?	O1.1 INFORMAL TEXTUAL DESCRIPTION Use informal text to identify and to describe domain abstractions and their relationships (e.g. domain-vision statements, domain-distillation lists).
D2	In which UML-compliant way should the domain concepts be formalized?	O2.2 PROFILE RE-/DEFINITION Implement the language model by creating (or by adapting existing) UML profiles (i.e. «`profile`» packages containing stereotype definitions). O2.3 METAMODEL EXTENSION Implement the language model by creating one or several metamodel extensions (i.e. «`metamodel`» packages containing new metaclasses and associations). O2.4 METAMODEL MODIFICATION Implement the language model by creating one or several metamodel extensions (i.e. «`metamodel`» packages containing redefining metaclasses and associations).
D3	Do we have to define constraints over the language model(s)? If so, how should these constraints be expressed?	O3.1 CONSTRAINT-LANGUAGE EXPRESSION Make language-model constraints explicit using a constraint-expression language (e.g. OCL, EVL).
D4	In which representation should the domain modeler create models using the DSML?	O4.1 MODEL ANNOTATION Attach UML comments as concrete-syntax cues to a UML model, containing complementary domain information such as keywords and narrative statements. O4.6 DIAGRAM SYMBOL REUSE Reuse built-in UML diagram symbols without modification.

4.1.1. Options

Domain abstractions are the basic building blocks of a language model and can be described using narrative as well as textual or diagrammatic specification formalisms.

Informal textual descriptions (O1.1) are primarily textual artifacts used to identify/define domain abstractions in an informal way; e.g., domain-vision (scoping) statements in narrative prose text, domain-distillation documents containing lists of core domain-abstractions and/or domain-definition and feature tables [60].

Formal textual models (O1.2) use textual formalism to identify and to unambiguously define domain abstractions and their relationships; e.g., mathematical expressions (e.g. universal algebra [2]) or formal grammars (e.g. extended BNF [61]).

Informal diagrammatic models (O1.3) are ad hoc diagrammatic representations not compliant to any standardized software modeling language and corresponding diagrammatic production rules; e.g., forms of visual concept modeling (e.g. early feature diagrams [60]) or pseudo UML diagrams (e.g. class diagram notations being used as re-composable drawing shapes).

Formal diagrammatic models (O1.4) are diagrams defined by a (formally) specified/standardized modeling language (e.g. MOF, UML, ER, STATEMATE) which adopt a graphical representation (e.g. UML class models, UML activity models, STATEMATE statecharts) to identify and to describe domain abstractions and their relationships. A combination of options may be beneficial, e.g., to facilitate communication about concepts. Diagrammatic models (O1.3, O1.4) can be used in support of a predominantly informal textual description (O1.1; see also related association O1.1 ⟷ O1.4 below). For explanatory purposes, normative and formal textual definitions (O1.2) are commonly supported by non-normative and informal textual descriptions (O1.1).

4.1.2. 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 association O1.4 ⟷ D2 below); for example, by perfective refinement (e.g. turning an informal into a formally correct diagram; O1.4). In general, the language-model definition is used as a communication vehicle for both, the domain experts and the DSML engineers.

Domain-expert audience: Different views and notations must be considered depending on the domain-expert audience. For example, in case of a DSML targeting software engineers (e.g. a DSML for defining software tests), the UML can be used to define the language model (O1.4). 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 implementation model (see D2 in Section 4.2).

Consistency preservation effort: Considering a combination of different options introduces the challenge of preserving the consistency between different domain-description artifacts (e.g. diagrams and textual descriptions). The negative effects of introducing inconsistencies, for instance, between

Table 2: An exemplary documented design decision named `UMLIntegration` for the DSML PRDM [123, 124] based on the document template from [49]. Decision options of the catalog (i.e. O2.2, O2.3, and O2.4) are referred to using the «see» tag in the *Decision* and the *Alternatives* sections. Drivers and consequences available from the catalog are referenced using «see» in the *Arguments* section.

Name	**`UMLIntegration`**
Current version	3 (MS2 «`Snapshot`»)
Current state	«`Approved`»
Decision group	None
Problem/issue	In which UML-compliant way should the domain concepts be formalized?
Decision	We opt for a combined strategy: First, a UML/BusinessActivities metamodel extension («see» `O2.3`) is created. The reuse of UML-based structural and behavioral features (duty associations to UML operations and properties) makes a slight modification of the UML metamodel necessary («see» `O2.4`). To bind standard UML metamodel elements (e.g. actions) to the extended duty-aware metamodel (e.g. as compensation actions), an auxiliary UML profile providing stereotypes to UML metaclasses is defined («see» `O2.2`).
Alternatives	Use either a UML profile («see» `O2.2`) or a UML metamodel extension/modification («see» `O2.3` and «see» `O2.4`) alone.
Arguments	There is a limited overlap between the constructed language model (i.e. the domain concepts) and existing, standard UML metamodel elements («see» `Domain space`). While, for example, concepts such as roles, subjects, and duties under different views (e.g. transition system for duty states, duty hierarchies) are not directly reflected in the UML metamodel, compensation actions for neglected duties can be modeled using standard UML actions. In addition, the BusinessActivities framework, as the basis for the DSML, deploys a UML metamodel extension. Compliance with the framework and its UML compatibility levels is a firm requirement. The integration into standard UML modeling tools is not a critical factor («see» `Tool integration`).
Related decisions	• This decision is «`caused by`» `DomainModel` • This decision is «`caused by`» `ExtendBusinessActivitiesFramework`
Related requirements	`Portability`, `MultipleViews`, `ProcessFlowMetaphor`
Related artifacts	[42]
History	Stakeholder / Action / Status / Iteration S. Schefer-Wenzl «`Developer`» / «`Propose`» / «`Tentative`» / MS1 S. Schefer-Wenzl «`Developer`» / «`Validate`» / «`Decided`» / MS1 M. Strembeck «`Domain expert`» / «`Confirm`» / «`Approved`» / MS2

a diagram and its textual description, can be mitigated by declaring either representation to be the normative one.

Cognitive effectiveness of a representation format: Another important driver is the cognitive load incurred by a representation choice, especially for formal textual (O1.2) or formal diagrammatic notations (O1.4). Irrespective of the target domain, diagrammatic representations benefit from their capacity to spatially group information bits that are spread in their textual form. Moreover, improved visual perception and visual reasoning facilitate processing and communicating domain abstractions (see [62] 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. Depending on the domain requirements (e.g., extensiveness of a domain, the domain experts' technical skills and preferences), textual (in support of visualizations) or graphical representations can be considered more or less adequate. For feature and variability modeling, both graphical, textual, and mixed concrete syntaxes are available, for example. Depending on the context (e.g., novices vs. experts) and different quality attributes (e.g., cognitive effectiveness, working time), each syntax style performs differently (see, e.g., [63, 64]). However, given the intentionally focused and thereby limited expressiveness of DSMLs (in terms of concepts covered), diagrammatic representations at the level of a generic domain description are suitable; especially when supported by (formal) textual descriptions to cover certain details. Besides, the perceptional bias of a target audience of domain experts (i.e., established legacy notations) might affect the cognitive effectiveness of the adopted representation type.

4.1.3. Associations[12]

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). We denote each association by a pairing of affected decision options and/or decision points (e.g. O1.2 ⟷ O3.1 or O1.4 ⟷ D2). An association between an option and a decision point shows a pairing 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 which is a short form of O1.4 ⟷ O2.1 ∨ O2.2 ∨ O2.3 ∨ O2.4).

At decision point D1, 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

[12] Filtered for and ordered by their relative support. Please note that some pairs of affected D1-related decision options (e.g. O6.2 ⟷ O1.4 ∧ O2.2) are described at other design decision points (i.e. in other sections of this paper; e.g. in Section 4.6) and are not re-iterated here. The same applies to association descriptions at other design decision points.

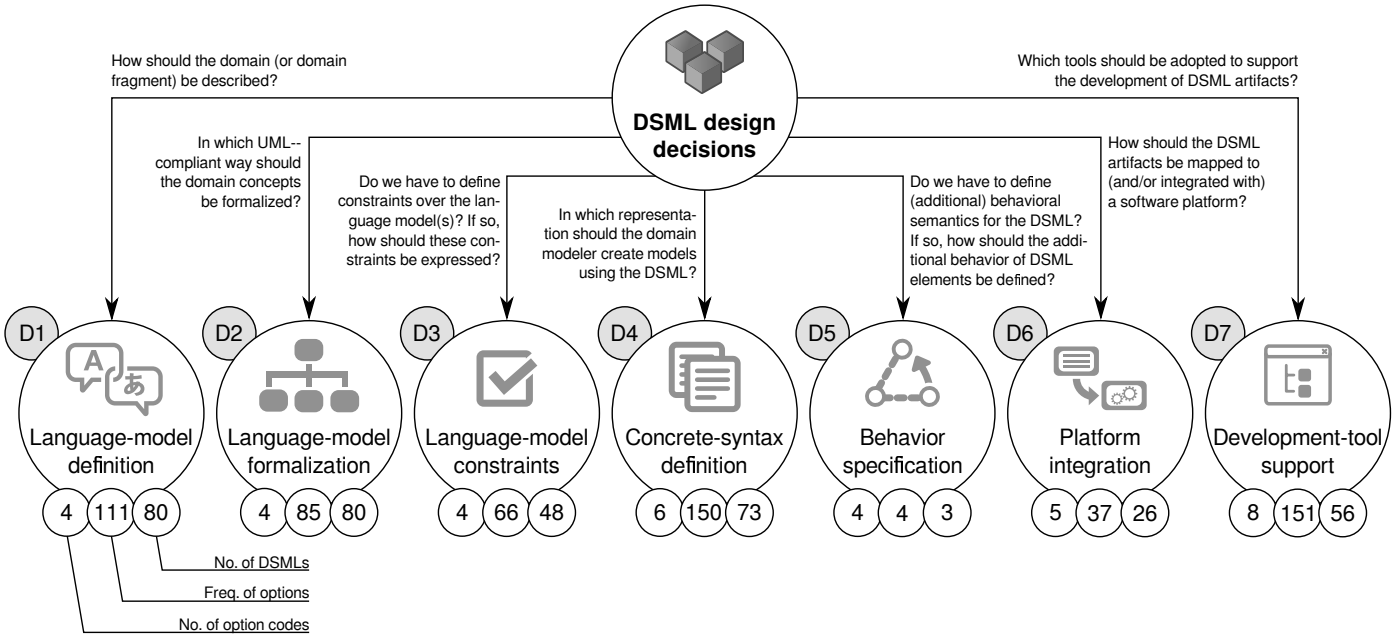

Figure 3: The problem statements leading to the seven DSML design-decisions we identified (D1–D7). The circles attached to each decision document the number of decision-options (without pseudo options), the number of occurrences of these options in the reviewed 80 DSML designs (absolute frequency), and the number of DSMLs choosing at least one of these options.

without further explanations (e.g. *textually accompanied formal models*; O1.1 ⟷ O1.4; see, e.g., [20, 42]). Textual descriptions (O1.1) were found for all of the DSML projects [42], particularly explaining the semantics of accompanying language models and rationale for design decisions (e.g., arguments on model and package designs, explanation of model elements, attributes, and associations).

If the domain description includes MOF or UML diagrams, a stepwise transition into a UML-based core language model (D2) is facilitated (i.e. *refined language-model formalization*). In particular, an association between options O1.4 and O2.2 (O1.4 ⟷ D2) is a candidate (see Section 4.2). Nevertheless, in some DSML projects found by our SLR, the definition of a MOF-based or modeling-language independent metamodel and the corresponding mapping to a UML profile was not explicitly documented. 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 though.

4.2. Language-Model Formalization (D2)

A formal language model (also: abstract syntax, *core* language model, or metamodel [20]) is an implementation of the language model using a well-defined metamodeling language such as the MOF. A metamodeling language is itself based on a well-defined and well-documented language model (CMOF for the UML metamodel [4]) 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). Here, formalization requires to make a decision on different UML/MOF implementation choices. Depending on the development style, this is typically performed after decision making at D1 (see Section 4.1). The card of this reusable design decision is depicted in Figure 5.

4.2.1. Options

M1 structural models (O2.1) implement the core language model using structural UML models at the M1 modeling level (e.g. via UML class diagrams [125]) rather than MOF expressed in class-diagram notation. In a class model, for instance, domain abstractions can be expressed as classes and their relationships via associations. Other examples are composite structure, component, and package diagrams.

Profile (re-)definitions (O2.2) implement the core language model by creating (or by adapting existing) UML profiles [3, 65]. A profile consists of a set of stereotypes which define how an *existing* UML metaclass may be extended.

Metamodel extensions (O2.3) implement the core language model by creating one or several extensions to an existing metamodel. A metamodel extension introduces new metaclasses and/or new associations between metaclasses to the UML metamodel or to other, pre-existing metamodel extensions [3, 4, 65]. The extension elements are typically organized into dedicated «`metamodel`» packages. The structure and semantics of existing elements of the UML metamodel are preserved.

Metamodel modifications (O2.4) 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

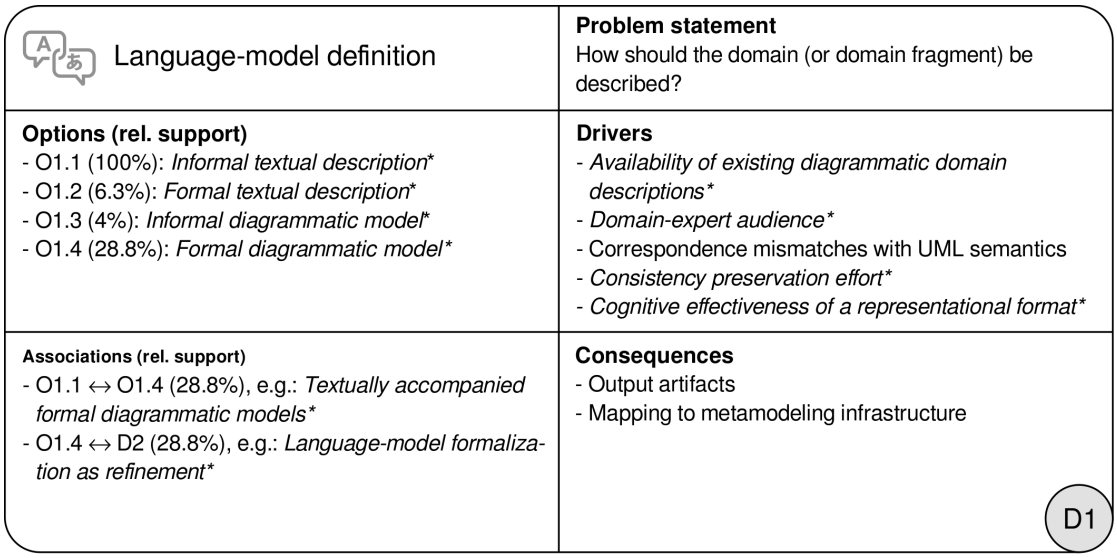

Figure 4: Design-decision card for D1: Defining a DSML's platform-independent language model. Associations are ordered by their support in the 80 reviewed DSMLs (relative occurrences). Details in italics and marked by an asterisk (*) are presented as part of the overview in this paper; the remainder is described in [42].

associations [3, 4, 65]. The extension elements are typically organized into dedicated «`metamodel`» packages.

A combination of options may include the definition of a metamodel extension as well as an equivalent profile definition. Similarly, stereotype definitions can be provided to accompany a metamodel extension/modification (see, e.g., [126]).

4.2.2. Drivers

Degree of DSML expressiveness: The expressiveness of a DSML is a major force in DSML development (see, e.g., [10, 36, 66]). 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., [10]). Therefore, profile constraints may only define well-formed rules that are more constraining (but consistent with) those specified by the metamodel [3] (see also association O2.2 ⟷ O3.1 ∨ O3.4). 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 association O2.2 – O2.4 ⟷ D4).

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 [65]. Re-usability of a UML extension is also affected by the extension's level of compliance with the UML standard (e.g. O2.2, 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 with respect to its compatibility and integration possibilities with other software artifacts (see also associations O4.6 ⟷ O2.2 and O6.2 ⟷ O1.4 ∧ O2.2 in Sections 4.4 and 4.6, respectively). 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, O2.2) is widely available (see, e.g., [10] and also Section 4.7).

4.2.3. Associations

A UML profile definition (O2.2) for the language-model formalization is typically observed in combination with a concrete syntax specification via annotating model elements (O4.1) and reusing diagram symbols (O4.6; see, e.g., [127, 128]). This association (O2.2 ⟷ O4.1 ∧ O4.6) can be explained by the *native 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" [3]. 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).

Constrained UML profiles (O2.2 ⟷ O3.1 ∨ O3.4): The specification of a UML profile (O2.2) was found accompanied by either formal (O3.1; see Section 4.3) or informal textual (O3.4) constraint definitions (or both; see, e.g., [129, 130]). The profile-specific part represents an extension to association O3.1 ⟷ O3.4 described in Section 4.3 and may indicate a demand for the definition of dedicated constraints besides native UML profile semantics.

Extending the UML metamodel (O2.3) without an explicit concrete syntax definition (O4.6)—even without anno-

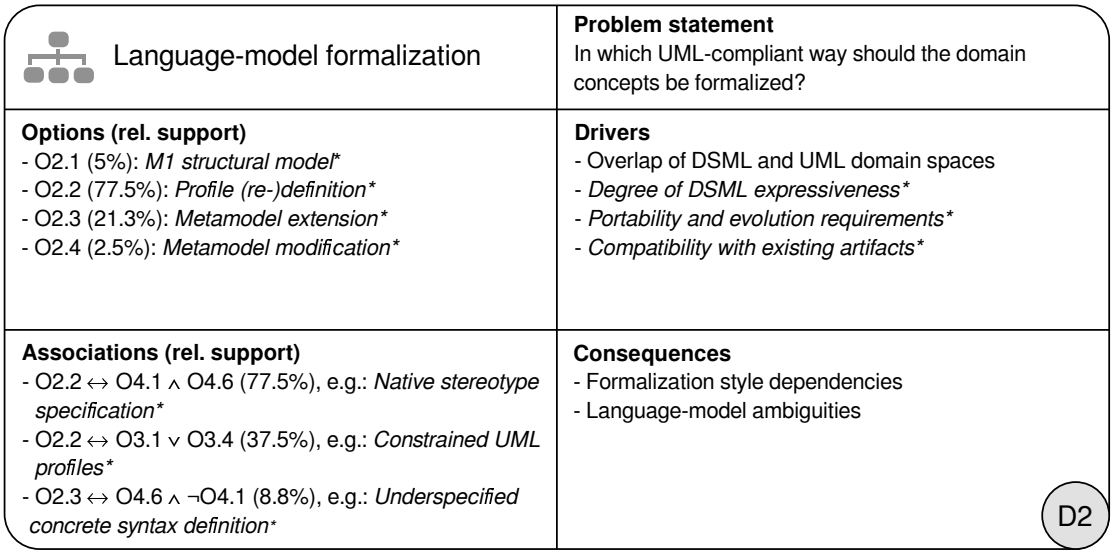

Figure 5: Design-decision card for D2: UML compliant formalization of a DSML's language model. Associations are ordered by their support in the 80 reviewed DSMLs (relative occurrences). Details in italics and marked by an asterisk (*) are presented as part of the overview in this paper; the remainder is described in [42].

tating model elements (O4.1)—was an observed association (O2.3 ⟷ O4.6 ∧ ¬O4.1; see, e.g., [131, 132]). The authors of such DSMLs implicitly assume that symbols defined for UML metaclasses (in the UML specification [3]) are inherited by the DSML-specific extensions (e.g. via a generalization relationship). However, this is in contrast to the practice applied in the UML specification itself (see O4.6 in Section 4.4) and results in an *underspecified concrete syntax definition*.

4.3. Language-Model Constraints (D3)

The language model (also: core language model or abstract syntax) has been implemented using either a UML metamodel extension/modification, a UML profile, or a UML class model. Depending on the requirements of the language-model definition and the capabilities of the modeling language, the language-model implementation might not capture all structural and behavioral semantics. For example, a structural UML model cannot (or only insufficiently) capture certain categories of constraints on domain abstractions, such as invariants for domain abstractions, pre-/post-conditions, or guards. As a result, the (graphical) metamodel alone could be incomplete or ambiguous. Thus, there might be the need for specifying additional model constraints (see also association O2.2 ⟷ O3.1 ∨ O3.4 in Section 4.2). 60% of the reviewed DSMLs specify additional constraints (O3.5). See Figure 6 for the corresponding design-decision card.

4.3.1. Options

Constraint-language expressions (O3.1) make language-model constraints explicit via a constraint-expression language, for example the Object Constraint Language (OCL [67]) or the Epsilon Validation Language (EVL [68]).

Informal textual annotations (O3.4) 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). Regarding the combination of options, textual annotations (prose text) can be used in addition to constraint-language expressions in order to provide natural-language constraint descriptions for readers not familiar with a specific constraint language, for example (see association O3.1 ⟷ O3.4).

4.3.2. Drivers

Constraint-formalization requirements: One decision driver steering a DSML designer towards an option are requirements on the formalization style of constraints (see, e.g., [20, 42, 69]). 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; see, e.g., [70]).

Language-model checking time: If tool integration for constraint checking on models is a requirement, we have to choose one or more of the options O3.1–O3.3 (see also associations O3.4 ⟷ O3.1 – O3.3 and O3.2 ⟷ O6.6 as well as, e.g., [51]). A driver towards either option is the intended time of language-model checking. Relevant points in time follow from the model formalization option adopted (e.g. class model vs. metamodel-based) and the platform support

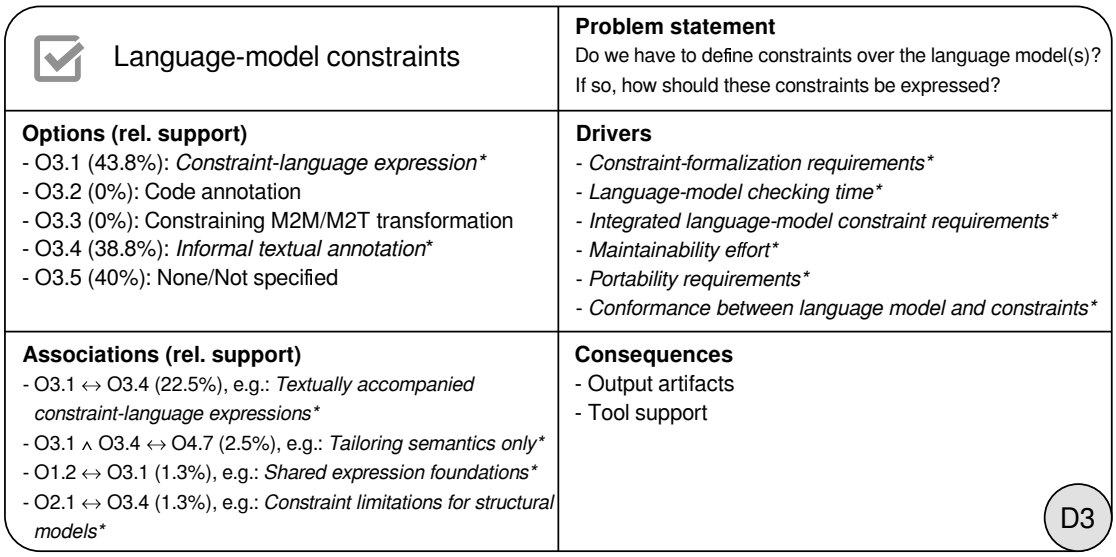

Figure 6: Design-decision card for D3: Defining constraints for a DSML's language-model. Associations are ordered by their support in the 80 reviewed DSMLs (relative occurrences). Details in italics and marked by an asterisk (*) are presented as part of the overview in this paper; the remainder is described in [42].

(model-level or instance-level checks; see, e.g., [71]).

Integrated language-model constraint requirements: Constraint-language expressions (O3.1) are developed with the purpose of integrating the constraints with the (meta)model representations (see, e.g., [72]). Examples are standard-compliant or vendor-specific OCL expressions for the UML. Models and constraints can also be integrated, for instance, via programming-language-based expressions, for example via natural-language UML comments (O3.4). Note, however, that O3.4 lacks support for automatic evaluation (constraint definitions would need to be transformed into evaluable expressions; see, e.g., [70]).

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 [73]). 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 (see, e.g., [72]). Each strategy affects the artifact complexity and the effort needed to keep the constraints and the models synchronized. See [74] for an approach to assist in constraint adaptation during metamodel evolution.

Portability requirements: If constraints should be portable between different MDD toolkits—such as, Eclipse Model Development Tools (MDT), IBM Rational Software Architect, No Magic MagicDraw—, platform-dependent options (e.g., code annotations) become infeasible. However, due to version incompatibilities and vendor-specific constraint-language dialects (e.g. Eclipse MDT OCL), even O3.1 cannot 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`; see, e.g., [75]).

Conformance between language model and constraints: When language models and their implementations evolve, constraints of each form must be adapted to match metamodel changes, such as OCL navigation expressions under O3.1 (see, e.g., [76]).

4.3.3. Associations

Similarly to the association O1.1 ↔ O1.4 (see Section 4.1), *constraint-language expressions are also found often annotated textually* (e.g., an OCL statement is additionally explained in natural language; see, e.g., [133, 134]). However, this is merely done to increase the readability of constraints as the reader may not be familiar with a certain constraint language (e.g., the OCL). This association (O3.1 ↔ O3.4) emerges also from the fact that not every language-model constraint can formally be described with a constraint language. 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., [3]). Such constraints have to be provided as text annotations in natural language.

Customizing the UML or any extensions of it (e.g., SoaML [77], SysML [78]) via explicit constraint expressions (O3.1, O3.4) without a concrete syntax definition (O4.7; see Section 4.4) to specify a DSML was another observed association (see, e.g., [134, 135]). This association (*tailoring semantics only*; O3.1 ∧ O3.4 ↔ O4.7) 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 associations O2.3 ↔ O4.6 ∧ ¬O4.1 and O4.6 ↔ O2.2 in Sections 4.2 and 4.4, respectively).

Thus, a corresponding DSML should only be used in isolation, without mixing concrete syntaxes of tailored and UML model elements.

Shared expression foundations: Adopting certain formal textual (e.g. set-theoretical) models affect the choice of a language (e.g. OCL [67]) for defining constraints over the core language model (O1.2 ⟷ O3.1). If there is a common (formal) 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., [79]), set theory and set algebras are a natural choice to define a generic language model.

Given a language model implemented at M1 (e.g. a class model), the language model is defined at the UML instance level (i.e. at the M1 layer [4]). 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 (*constraint limitations for structural models*; O2.1 ⟷ O3.4).

4.4. Concrete-Syntax Definition (D4)

The concrete syntax of a UML-based DSML serves as its user interface and can be defined in several ways. Multiple concrete-syntax styles are available and a DSML can be equipped with one or more concrete syntaxes. Different syntax types can be defined and tailored to the needs of the modeler (e.g., chosen depending on the modeler's domain and/or software-technical proficiency). A clear majority of DSMLs include a concrete-syntax decision, only a minority leave the concrete syntax undefined (*none/not specified*; O4.7). See Figure 7 for the corresponding design-decision card.

4.4.1. Options

Model annotations (O4.1) 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., [136]). 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 [3].

Diagrammatic syntax extensions (O4.2) extend one or multiple UML diagram types by creating novel symbols in addition 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., [80]). A notable example of a diagrammatic extension is the option to equip UML stereotype elements with dedicated icons which appear in addition to the standard notions of stereotyped elements (e.g. tags or nested icons in classifier rectangles [3]).

A *mixed syntax (foreign syntax)* (O4.3) creates a DSML's concrete syntax as either a non-diagrammatic syntax (textual, tree-based, or tabular) or as a diagrammatic syntax that is not integrated with the native UML syntax. 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 set. 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). Important candidates for non-diagrammatic foreign syntaxes are textual, tree-structured, and tabular notations (see, e.g., [32]).

A *textual* concrete syntax expresses DSML models in a text-based format [81]. Typically, textual grammars are used to define a textual concrete syntax (e.g. via the extended BNF [61]). 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 an 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 [82]). 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.

Diagram symbol reuse (O4.6) is a commonly applied option and means that no custom, DSML-specific extension to the standard UML symbol vocabulary is created. The UML has a concrete syntax that provides a visual notation, with its symbol set being organized into 14 diagram types [3]. The number of distinct graphical symbols applicable in these diagram types ranges from eight (in communication diagrams) to 60 (e.g. in class diagrams) [80]. With the UML specification [3] not being explicit about the case of undeclared notations (i.e. missing "Notation" sub clauses), the reuse of symbols that are defined for native UML metaclasses which are refined by the DSML must be stated explicitly (see also association O2.3 ⟷ O4.6 ∧ ¬O4.1 in Section 4.2). This resembles the practice applied in the UML specification itself

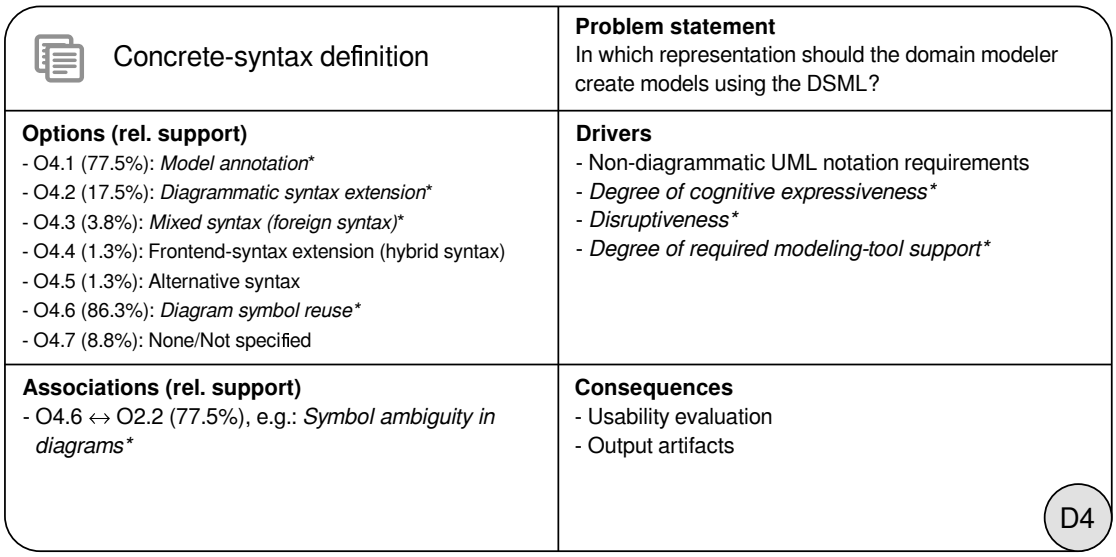

Figure 7: Design-decision card for D4: Defining a DSML's concrete syntax. Associations are ordered by their support in the 80 reviewed DSMLs (relative occurrences). Details in italics and marked by an asterisk (*) are presented as part of the overview in this paper; the remainder is described in [42].

(e.g. "A Class is shown using the Classifier symbol" [3]).

4.4.2. Drivers

Degree of cognitive expressiveness: UML stereotypes have a 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 define 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 a hierarchically structured DSML, but falls short in an adequate representation of process-flow constructs such as loops and sequences, for example.

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) [3]. 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 corresponding set of experiments [17] 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 which says 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" [18]. 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 has a long history of symbolic signs, a change may cause confusion and comprehension problems which may again lead to a decrease of DSML users' efficiency.

Degree of required modeling-tool support: A textual concrete syntax (O4.3) can be processed by a parser and (most often) does not need specific editor tools (in contrast to 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 can be serialized as 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).

4.4.3. Associations

When reusing existing UML symbols, the resulting "extended" diagrams risk becoming ambiguous. In particular, using the same symbol for two or more different concepts means that refining concepts cannot be distinguished from the refined ones (*symbol ambiguity in diagrams*; see O4.6 ↔ O2.2 and also, e.g., [65, 83]). 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 in order 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 [3], however, without adding to the abstract syntax and semantics of the language model.

4.5. Behavior Specification (D5)

The behavioral specification of a DSML (also: *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 [20]. Behavioral relationships may emerge from the language-model formalization (D2; see Section 4.2) or the language-model constraints (D3; see Section 4.3). Explicitly specified behavior allows for a correct mapping of the (platform-independent) DSML specifications to a certain software platform (see Section 4.6). If no behavioral specification exists (which is the case for nearly all of the 80 DSMLS investigated; see O5.5 in Figure 8), 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). See Figure 8 for the design-decision card.

4.5.1. Options

M1 behavioral models (O5.1) specify additional behavior of language-model elements using UML behavioral models at the M1 level (e.g. state machines, interaction diagrams, or activity diagrams). For instance, in the UML a classifier can reference "owned behavior" specifications. Behavior is then executed in the context of the directly owning classifier [3].

Formal textual specifications (O5.2) 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 with the purpose that its correctness can be checked (e.g. by using the Z notation [84]).

Informal textual specifications (O5.3) are used to informally specify the behavior of a DSML, for example via narrative prose text.

With regard to the combination of options, textual comments (O5.3) may be used to annotate models (O5.1) or to clarify formal specifications (O5.2), for example. Such combined uses were not found documented in the 80 DSMLs, though.

4.5.2. Drivers

Model consistency preservation: UML behavioral models (O5.1) allow for a native integration of behavioral semantics into UML-based DSMLs (see also association O5.1 ⟷ O3.1). For example, the behavior of a DSML element can be defined via an owned behavior specification [3]. 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 [85]).

Limited expressiveness: If it is not feasible or even impossible for some behavioral expressions to be sufficiently expressed via graphical models (O5.1) or formal (textual) 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 [85]. The respective execution model is specified as if all communications were perfectly reliable and deterministic (e.g. it is assumed that signals and messages are never lost or duplicated), which is not realistic, of course. 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 used to specify a particular behavior, the correctness of formal specifications (O5.2) and executable (i.e. well-formed) models (O5.4) can be (automatically) checked (see, e.g., [134] and [84, 86]). If the objective is to verify all artifacts in a DSML (such as, language model, language-model constraints, behavior specification, platform-specific artifacts), O5.2 is an option. 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 example, 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 the respective user requirements best (O5.2, O5.3). For instance, in case of the fUML, UML models can be represented using the action language ALF [87]. ALF acts as a textual surface representation for UML modeling elements that can be used to specify executable behavior.

4.5.3. Associations

UML M1 models can be attached to metamodel elements for behavioral specifications (e.g. via the `ownedBehavior` relation of a `BehavioredClassifier` [3]). In doing so, they are constraining/defining the behavior of metamodel elements (*M1 behavioral models as constraints*; O5.1 ⟷ O3.1). For example, in [123, 124] the authors make use of a UML state machine to define states (e.g. passive, pending, discharged) and transition options between those states for DSML elements.

4.6. Platform Integration (D6)

At this stage, decisions must be made on how to produce platform-specific executable models (esp. source code) by mapping DSML models (or an executable subset of the models) to a software platform (e.g. programming languages, frameworks, components, service applications). This platform integration is achieved via different types of model

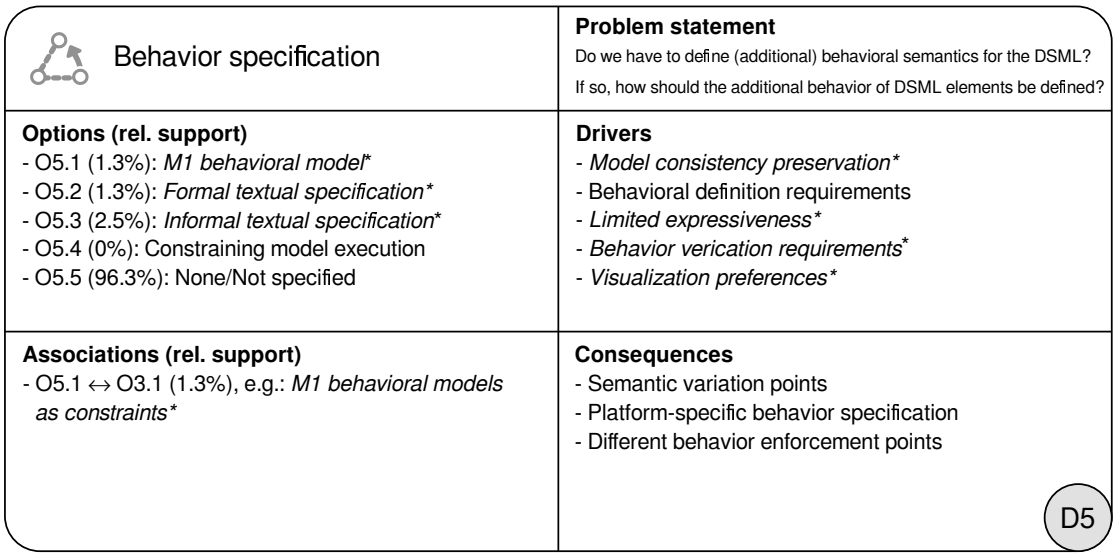

Figure 8: Design-decision card for D5: Specifying a DSML's behavior. Associations are ordered by their support in the 80 reviewed DSMLs (relative occurrences). Details in italics and marked by an asterisk (*) are presented as part of the overview in this paper; the remainder is described in [42].

transformations (see, e.g., [88, 89]) that convert a model into another platform-specific model (also: model-to-model transformation, M2M) or into textual/executable software artifacts (also: model-to-text transformation, M2T; see also association O6.2 ↔ O6.5). 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 [42]. Not that performing no platform integration at all is also a viable option (*none/not specified*; O6.6), for example, when the DSML should only serve for documentation purposes, for sketching a software design, or for analyzing requirements. Two thirds of the 80 reviewed DSMLs do not document or contain platform-integration decisions. The design-decision card is shown in Figure 9.

4.6.1. Options

Intermediate model representations (O6.1) provide for generating a second and intermediate model (i.e. the target model) based on a DSML model (i.e. the source model) using so-called model-to-model transformations. This intermediate model can be described via an own (separate) metamodel. The source and target models are also separate model entities. From the intermediate model, platform-specific artifacts/models can be 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 metadata (see, e.g., [90]).

Generator templates (O6.2) create transformation templates 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 EGL generator-template languages.

API-based generators (O6.3) 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 [90] or a mixin-based API [91]), 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., [92]). The resulting platform-specific artifacts are independent of the generator language or the generator implementation.

Model-to-model (M2M) transformations (O6.5) perform platform integration via (multiple) endogenous M2M transformations specified via M2M transformation languages (e.g. ATL [93], ETL [68]). 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 model-to-text (M2T) transformations (O6.2, O6.3).

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 [90]. In this

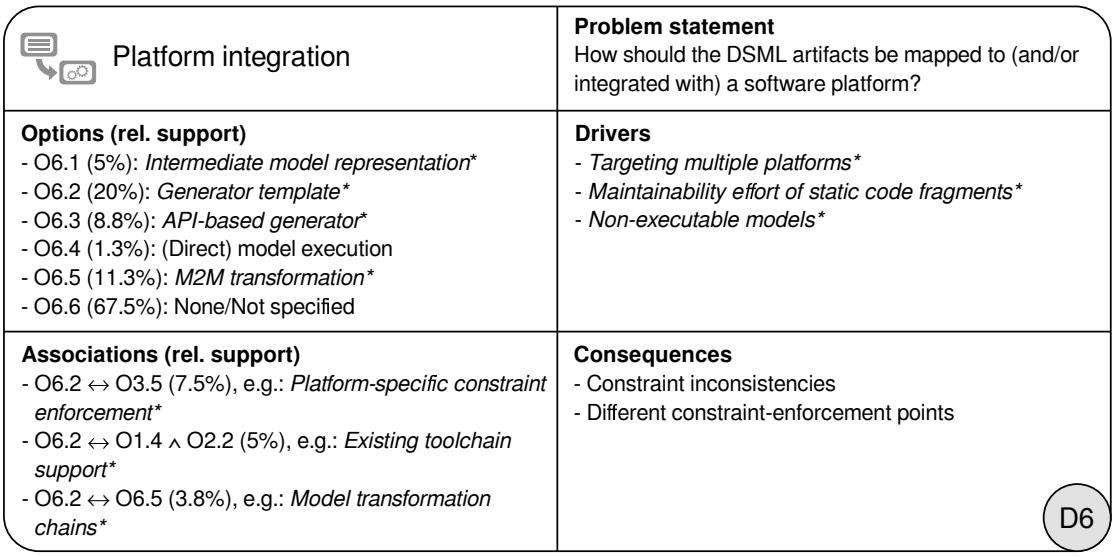

Figure 9: Design-decision card for D6: Defining a DSML's platform integration. Associations are ordered by their support in the 80 reviewed DSMLs (relative occurrences). Details in italics and marked by an asterisk (*) are presented as part of the overview in this paper; the remainder is described in [42].

way, transformation templates can operate on compressed and canonicalized DSML models (see, e.g., [90]), generators run against decorator models providing generation-specific metadata (e.g. an EMF generator model [72]), and a model interpreter finds a prefabricated and execution-oriented model representation (e.g. an unfolded control flow).

4.6.2. Drivers

Targeting multiple platforms: An intermediate model (O6.1) can act as a common, canonicalizing representation that can be mapped to multiple software 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 integration, an intermediary representation can increase the efficiency of subsequent M2T transformations.

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 [92]. Depending on the relative amount of static code fragments, an API-based generator involves extra maintenance efforts for managing the interwoven fragments of generative code and static code.

Non-executable models: If the DSML should only serve for modeling purposes, for example via the definition of a UML profile 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 association O3.3 ↔ O6.6 in Section 4.3) and might primarily serve as a communication vehicle between domain experts and software engineers.

4.6.3. Associations

The observed association O6.2 ↔ O3.5 (*platform-specific constraint enforcement*) is characterized by a late and platform-specific constraint enforcement point. Corresponding DSMLs do not define explicitly constraints for the language model (O3.5; see Section 4.3), but integrate them into (templates of) code generators (see, e.g., [137, 138]). As generation templates (O6.2) are applied to instances of the language model, constraints can basically be enforced. However, constraints are checked late in the DSML development process; i.e. at the time of executing model-to-text (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 an M2T transformation is not interpreted by its generator component).

Existing toolchain support (O6.2 ↔ O1.4 ∧ O2.2): Tools for editing UML models, including the definition and application of profiles (see O2.2 in Section 4.2), are nowadays frequently available (e.g. No Magic MagicDraw, Eclipse Papyrus, IBM Rational Software Architect). 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, EGL, JET, or Acceleo (see, e.g., [88, 94]). Several UML model editors provide combined tool support for M2T transformations in an MDD-based way, as well – for example based on EMF-compliant models in the Eclipse toolchain. Thus, the observed association is characterized by a high availability of

modeling tools and generator engines (see, e.g., [139, 140]). Nevertheless, a formal diagrammatic model not compliant with the UML specification (e.g., an ER model; see O1.4 in Section 4.1) 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 [4] (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. Moreover, a partially tool-supported approach for the semi-automatic transformation of MOF-based models into UML profiles is presented in [95] (see also association O1.4 ⟷ D2 in Section 4.1). Further tooling decisions related to the development of DSMLs are discussed in Section 4.7 (decision point D7).

Model transformation chains (O6.2 ⟷ O6.5) are characterized by endogenous M2M transformations (O6.5) prior to the code generation step (O6.2; see, e.g., [141, 142]). 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 this association being employed for analyzing models [142] as well as for generating test cases [141]. On the one hand, [142] provides an approach for analyzing OCL-constrained UML class models for inconsistencies via Alloy [96]. A UML class model is transformed into an instance model of the Alloy metamodel (both instantiating the MOF; O6.2). From the Alloy model, an 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, [141] 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 [142] and the platform-specific application and test models [141] all serve as intermediate representations (O6.1) for the creation of textual artifacts.

4.7. Development-Tool Support (D7)

DSML tool support requires important design decisions and, in turn, affects decision making on other DSML concerns (decision points). In MDD, the objective is to assist engineers in the creation of DSML language models as well as to automate the evaluation of language-model constraints, the transformation of models to platform-specific software artifacts (e.g. source code), and so forth. For instance, the generative nature of MDD makes model-transformation engines a key building block of most DSML approaches (see, e.g., [97, 98, 99]). At the same time, the choice of a particular MDD tool chain may affects other DSML design decisions because not all decision options (e.g. concrete-syntax options) might be supported by a given toolkit (see, e.g., [42, 100]). However, the variety of available MDD tools (e.g. IBM Rational Software Architect, Sparx Systems Enterprise Architect) makes the corresponding decision challenging. Researchers have discussed MDD tooling as a key barrier to MDD adoption (see [101] for a recent overview). The design-decision card is shown in Figure 10.

4.7.1. Options

Language-model editors (O7.1) are used to create, edit, and maintain the language model of the DSML. The editor can support the development of the language-model diagrammatically (e.g. Eclipse EcoreTools) or textually (e.g. Eclipse Emfatic).

Constraint evaluators (O7.2) are used to automatically analyze and validate conformance criteria for models. For example, language-model constraints defined as dedicated constraint-language expressions (e.g. OCL invariants evaluated via the OCL engine of the Eclipse MDT).

Generating diagrammatic-syntax editors (O7.3) support the representation of a DSML's graphical concrete syntax. Corresponding tools allow for creating, editing, and maintaining tailored editors for the domain-specific models in a given graphical concrete syntax (e.g. Eclipse Graphical Modeling Framework, GMF).

Generators for textual-syntax editors (O7.4) support the representation of a DSML's textual concrete syntax. Corresponding tools (e.g. Eclipse Xtext) allow for creating, editing, and maintaining tailored editors for the domain-specific models textually (i.e. textual DSLs).

Model-execution engines (O7.5) are used to interpret models directly without the need of additional transformation steps (e.g. the Moka module for Eclipse Papyrus includes an execution engine complying with fUML [85]).

M2M transformation engines (O7.6) take one or multiple models as input and generate one or multiple models as output. An editor supports creating, editing, and maintaining transformation specifications in a dedicated transformation language (e.g. ETL [68]).

M2T transformation engines (O7.7) take one or multiple models as input and generate one or multiple textual artifacts as output. An editor supports creating, editing, and maintaining transformation expressions in a dedicated transformation language (e.g. EGL [68]).

Orchestration engines (O7.8): As a DSML may consist of several tool-supported artifacts (e.g. language-model constraints, M2M/M2T transformation expressions etc.) for which the order of execution is important, orchestration engines (O7.8) can be used to coordinate the execution process as well as data input/output requirements of these artifacts providing an MDD-based tool chain for DSML development (e.g. Eclipse Modeling Workflow Engine, MWE).

4.7.2. Drivers

Availability of existing tools: One of the drivers towards adopting a specific toolkit or toolchain is the availability of existing tools and their suitability to support DSML development (e.g. to serve as an editor for the language model; O7.1). Porting existing (legacy) tools to fulfill requirements of a new DSML may be more efficient than adopting (and possibly adapting) a completely new tool set. However, whether

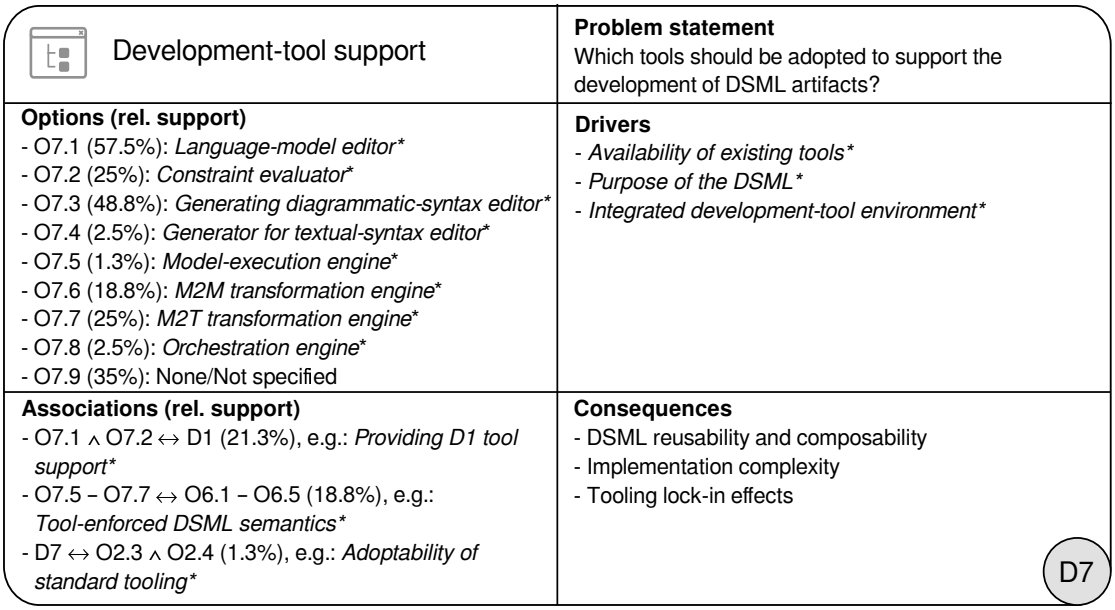

Figure 10: Design-decision card for D7: Tools supporting the development of a DSML. Associations are ordered by their support in the 80 reviewed DSMLs (relative occurrences). Details in italics and marked by an asterisk (*) are presented as part of the overview in this paper; the remainder is described in [42].

existing tools qualify for supporting the development of a DSML is dependent on a multitude of factors, for example, the capability of developing a DSML with existing software artifacts, the compatibility of different tooling license models, the maturity of available tools, or the portability, evolution, and maintainability effort needed in comparison to adopting a completely new tool set [21].

Purpose of the DSML: The adopted tools must also match the DSML's purpose, of course. For example, constraints defined in a format that cannot be validated automatically (e.g. O3.4) may make constraint evaluators (O7.2) useless. In the same way, models directly interpreted via a model-execution engine (O6.4, O7.5) may render any M2M/M2T transformation engines (O7.6, O7.7) unnecessary. In contrast, transformation engines may be essential when following a generative approach (see, e.g., [60]) to create (executable) platform-specific artifacts (e.g. source code).

Integrated development-tool environment (IDE): As a DSML consists of multiple, complex, and interrelated artifacts (models, model transformations etc.), the availability of an IDE becomes crucial (e.g. to fulfill compatibility and traceability requirements). Thus, DSML development tools must also be assessed regarding their ability to interoperate (e.g. to enable an orchestration engine to coordinate the execution order of interdependent tools; O7.8). For example, interoperability between different Eclipse-based software tools is facilitated through utilizing standardized interfaces (e.g. export/import of XMI serialized models [73]) allowing, for instance, that model transformation chains (see, e.g., [102]) can be developed by cascading multiple M2M/M2T transformation engines (O7.6, O7.7).

4.7.3. Associations

All of the decision points involved in the DSML development process (D1–D6) may be supported by tools (e.g. a language-model editor or an M2T transformation engine; see the D7-related options above). Thus, the decision point on development-tool support (D7) is likely to have interdependencies with each of the other six decision points. The *effort of providing D1 tool support* (O7.1 ∧ O7.2 ⟷ D1) for the initial definition of a generic DSML language-model (e.g. language-model editors) depends on the chosen representation option. An informal textual description (O1.1) may not need a DSML-specific tool support. For formal textual and diagrammatic models (O1.2, O1.4)—when based on well-defined and/or standardized specifications—it is likely that some sort of (reusable) development-tool support exists already (e.g. mathematical formula or UML diagram editors, model validators). In contrast, informal diagrammatic models (O1.3) may lack any tool support (e.g. an underspecification of the semantics of ad hoc modeling languages may render constraint evaluation impossible). The degree of tool support at decision point D1 also influences the effort needed to (automatically) refactor language-model concepts into a UML-compliant format (D2). For example, it may be easier to define reusable mappings for a formally specified model (O1.2, O1.4) than for an informal diagrammatic model (O1.3) where the semantics of the modeling constructs are not clearly specified.

The definition of *tool-enforced DSML semantics* (O7.5 – O7.7 ⟷ O6.1 – O6.5) for the phase of platform integration can be distinguished into interpretative semantics (O7.5 for O6.4), which directly execute a model representation, and translational semantics, which compile a model into a model/program expressed in another language (O7.6, O7.7 for

O6.1–O6.3, O6.5) [103]. When transforming models, keeping track of a model's origin enables linking elements of the transformation result back to the original input model. Such traceability capabilities of tools are particularly important for debugging activities. Furthermore, to better understand the behavior of a model, it can be useful to have a view of the code the model compiles to. For this, tooling features that can display the model and the generated code side by side are beneficial [103].

In this context, the extent of UML compliance of a DSML's language-model formalization influences the *adoptability of standard tools* (D7 $\leftrightarrow$ O2.3 $\wedge$ O2.4). For example, if the DSML's language model is formalized via an extension to the UML metamodel (O2.3; e.g. via the introduction of new datatypes), standard language-model editors (O7.1) may not be able to handle the new modeling constructs or it may be difficult for standard generators of diagrammatic-syntax editors (O7.3) to visually integrate new syntax elements within the native UML syntax set.

5. Limitations

Design-Decisions Catalog. The catalog of reusable design decisions was deliberately narrowed down to DSMLs embedded into UML 2.x. We excluded DSMLs from the catalog that are based on UML 1.x and metamodeling infrastructures such as Kermeta, Ecore, XMF. While this appears, at first glance, as a barrier to generalizing the reusable design decisions, the focus on UML 2.x was necessary because important decisions taken for the UML 2.x are substantially different from those for UML 1.x, not to mention from other infrastructures. Moreover, there are important lines separating the UML 2.x and UML 1.x regarding their language architectures and the foundational semantics of the available extension techniques (e.g. profiles, package merge; see [16, 104, 105]). The survey also supported the relative importance of the UML 2.x as opposed to its predecessors: More than 50% of the respondents (42/80) indicated having adopted UML 2.x (versions 2.0 through 2.5) for their DSML projects. Note, however, that many reusable decisions can still be adopted in a broader sense to be compatible with DSMLs based on other metamodeling infrastructures and DSLs (e.g. concrete-syntax decisions).

Survey. The design of our questionnaire included four question types. *Crucial questions* were used to identify break-off, partial, and complete questionnaires. All crucial questions were also mandatory questions. A *mandatory question*, when presented to the participant, had to be answered in order to continue the questionnaire. However, note that not all mandatory questions needed to be presented to a participant because of filter questions. A *filter question* controlled the succession of the questions in our questionnaire (e.g. depending on an answer, a subsequent question was presented or not). An *optional question* could be left out by the participant. Regarding the outcome rates of the survey, we considered an attempt a break-off if the respective participant answered less than 50% of the crucial questions. If a participant answered 50% or more of the crucial questions but less than 100% of the mandatory questions, this was considered a partial response. If a participant answered 100% of both, the crucial and the mandatory questions, this was considered a complete response (for further details see [5]).

Closely following the guidelines from [106], we carefully designed the questionnaire to minimize any negative influence on participants and their replies. For example, we paid specific attention to develop value-free, non-suggestive wordings for the questions and items. Moreover, the questions have been devised in an exhaustive, unbiased manner providing mutually exclusive response categories. The questionnaire was pre-tested in two iterations. In the first iteration, four co-researchers with a general software-engineering background were asked to complete the questionnaire to provide feedback on the comprehensibility of the questions and to measure the time required to answer all questions. Second, and based on a revised version of the questionnaire, we invited another three participants to run another pre-test. This second iteration included two experts, one on design-decision documentation (Uwe van Heesch; see, e.g., [49]) and one on UML-based DSML designs (Sigrid Schefer-Wenzl; see, e.g., [123, 124])—the third again having a general software-engineer background. Finally, we consulted the WU Competence Center for Empirical Research Methods to review the survey's design and questionnaire.

Nevertheless, personal bias cannot be ruled out completely. For instance, certain answer options may have been interpreted differently by the respondents. As an example, the answer option "extremely important" [5] could mean different things to different subjects. The comparatively high effort per participant to complete the questionnaire (i.e. 15–20 minutes), and the expected substantial barriers to motivating invited researchers and practitioners to actually participate, did not allow us to provide for repeated measurement. Therefore, we have no means to quantify the internal consistency (reliability) of the responses (e.g., by having each participant complete the survey twice after some cool-down phase). As, to the best of our knowledge, there is no alternative or complementary data set on UML-based DSML design decisions available (i.e. a second measurement instrument), we could also not quantify the validity of the survey responses.

We asked MDD researchers and practitioners identified via carefully selected scientific venues to take part in the survey. Although the venues included premier outlets for researchers and practitioners in the field of MDD and DSMLs, our selection strategy (convenience sampling [45]) may have missed further peers who are professionally designing and developing DSMLs. Furthermore, non-response errors may have been introduced because some members of the sample did not respond to our invitation for taking part in the survey (although we sent out reminders). Thus, there is a possibility that those who do not believe in the benefits of documenting DR may have opted not to respond which would bias our results. Hence, and because of the non-probabilistic sampling

method [45], it is difficult to assess the representativeness of the sample.

The majority of our respondents indicated that their DSMLs are based on Ecore, MOF, and/or UML (see Section 2.1). However, our survey, the questions, and the data reporting in this paper are not specific to any infrastructure, modeling language, or MDD tool chain. This was mainly because we aimed at maximizing participation and responses. In most sections of the questionnaire, we explicitly asked participants to answer the questions based on their general experience with their DSMLs (whether UML-based or not) [5]. Given these generic, non-UML-specific questions, we would expect widely consistent responses when re-running the survey explicitly setting the context to UML-based DSMLs (i.e. we expect a good alternative-form reliability [45]).

Literature Review. Despite being prior work, because the documented design decisions are based on papers identified by our literature review, the work reported in this paper inherits some limitations of the original review. We closely followed established guidelines on designing and conducting SLRs available from research on evidence-based software engineering to avoid any pitfalls [41]. The DSML papers were subjected to a documentation analysis to extract design decisions from scientific publications and their companion material. We considered supporting material if reported by and available from the publication authors. A documentation analysis represents an indirect data-collection technique [107]. Therefore, information on the ordering of design decisions over time (decision sequences) often remained implicit and, therefore, unrecoverable for us. Even if documented, any indirectly observed order of decision options adopted by DSML engineers might have also followed from the presentation requirements of a scientific publication (i.e. the one reporting on a DSML); an order which does not necessarily correspond to the original one during decision making. Therefore, in our research setting, we could only study option sets in terms of decision associations. For the same reason, we focused on one process style of DSML development only (i.e. language-model-driven development). Thus, we might have neglected design-decisions details (e.g., associations) characteristic for other development styles (e.g. mockup-driven DSML development [20]).

There is a bias inherent to the SLR design in that by relying on scientific publications only, the reusable decisions on DSMLs could be specific to DSMLs developed as research-driven prototypes and proof-of-concept implementations. While there is an important scientific audience, with approx. 48% of the survey respondents (38/80) having contributed to at least one UML-based scientific DSML, design decisions during DSML development in the software industry might not necessarily be covered by the catalog. For example, we established a clear preponderance of UML profiles in 80% of the reviewed DSMLs. Whether this characteristic also holds for DSMLs reported in other, predominantly non-scientific venues cannot be answered at this point. However, we find it difficult to assess the severity of this bias. To begin with, the primary studies reviewed in our SLR did not disclose their industrial background. Similarly, while related empirical studies on UML usage certainly document the existence of UML extensions and UML-based DSML designs (see, e.g., [6, 7]), they do not discriminate between industry-driven and research-driven projects. Our survey also documents that it is likely that DSML developers have contributed to both scientific and industry projects over time, so that their design expertise (although documented in scientific publications) reflects industrial practices: Approx. a quarter of the total respondents (19/80) have contributed to both scientific and industry UML-based DSMLs. For all DSMLs, this share increases even to 57.5% (46/80).

6. Discussion

In our survey [5], we collected expert opinions from DSML researchers and practitioners on different aspects of documenting and using DR when developing DSMLs (see also Section 2.1). As for documenting DR, we asked the participants to indicate whether they performed certain documentation activities (known from DR literature) for at least one DSML, and, if the answer was yes, how they rate the usefulness of DR. Figure 11 shows that 92.6% of the respondents (sample size: n=68) documented DR in written form. Written documentation artifacts include source-code comments and changelog files, for example. This activity is followed by meeting protocols (83.8%; e.g. brainstorming sessions, focus groups) and conceptual diagrams (69.1%; e.g. decision-flow modeling). These three DR documentation activities were also the ones the participants perceived most useful with 73% (useful to extremely useful; n=63), 71.9% (n=57), and 72.3% (n=47), respectively.

The combined levels of high usage and high perceived usefulness of written DR documentation, as part of more general documentation artifacts, can be explained straightforwardly since such practices (code commenting, source-code and model-management systems) are among the most established ones in software-development projects. This also holds for meeting protocols, which are often seen as central documentation artifacts in collaborative software development and project management. The importance of diagrams as form of decision documentation, whether ad hoc or based on a dedicated modeling language, has been stated before (e.g., QOC diagrams [108], UML activity diagrams for tailored development processes [20]) and is confirmed by our survey results. In Section 3, we showcase how the catalog can support written DR documentation in general as well as meeting documentation by referencing elements from the catalog. The use of the catalog for QOC diagramming is exemplified in [41].

It is noteworthy that some participants commented (via freetext comments in the questionnaire) that they have performed some of these activities before, but did not explicitly or systematically documented them (e.g. because of customers not wanting to become recorded) or they documented not the complete rationale leading to a design decision, but

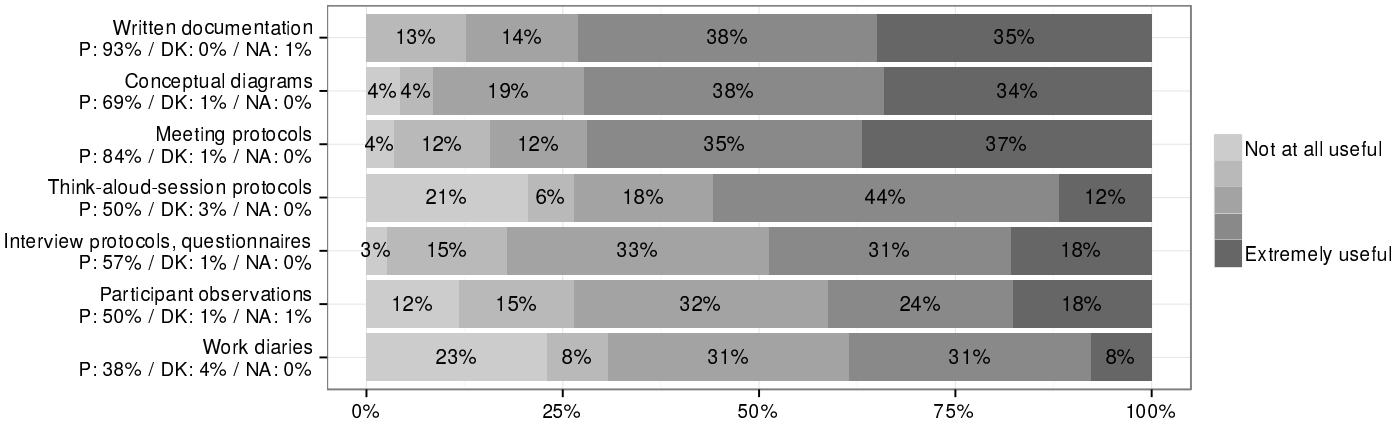

Figure 11: Number of respondents having performed selected activities (performed at least once) and perceived usefulness of the performed DR documentation activities (n=68), ordered by decreasing combined levels 4 and 5 support (useful to extremely useful). P = Performed; DK = don't know; NA = not answered.

the decision (solution) only. However, being aware of such undocumented activities is important because they exhibit potential to introduce routine documentation practices and they can be influential in the overall decision-making process (e.g. because they contribute to eliciting requirements; see [5]).

Based on the opinions of participating MDD researchers and practitioners, we derived a documentation format (decision records) for our setting (UML-based DSMLs). In Section 3, a structured document to capture one project-specific decision (referencing the catalog) is introduced by example. Looking at the actual practice of documenting DR (i.e., without any document templates as a scaffold), the survey participants indicated that they have documented the following details for at least one DSML (in descending order of respondent counts): issues (76.9%, sample size: n=65), alternatives (69.2%), criteria (67.7%), project context (64.6%), decision-making context (58.5%), and activity context (46.2%; see Figure 12). The frequency of documenting these details is widely mirrored by their perceived usefulness: criteria (81.8%, n=44), issues (78%, n=50), alternatives (64.4%, n=45), decision-making context (55.2%, n=38), project context (54.8%, n=42), and activity context (53.3%, n=30). To the extent these details are relevant for the description of a reusable design decision, the top-ranked items are reflected by the current documentation format (criteria: driver, consequences; alternatives: options; see Section 2.2). Issues, which are specific to a particular decision-making step, are out of the scope for reusable decisions.

The overall picture is that major problems, listing alternative solutions, and reasons for/against a solution are most often documented and perceived as most useful. Contextual information appears less frequently documented and considered comparatively less useful. In this context, the participants' comments can shed light on the corresponding figures: Some survey participants articulated that contextual details "are useful, but in practice the results are used only rarely", that "the industry projects did not really document design decisions", and that "these things were documented in research papers, but more as a result of the dissemination obligation, not so much to support the development/design process" [5].

Finally, our survey covered the participants' opinions on selected forces (e.g., time and budget constraints) in their DSML development projects which they encountered at least once, and whether these forces constituted actual barriers to DR documentation (i.e. their criticality; see Figure 13). Candidate barriers we proposed to the participants reflect the state of research on the DR capture problem [22, 23]. Four-fifth of the participants encountered (at least one of) the candidate barriers, in decreasing order by frequency: absence of tool support to document decisions (80.6%, sample size: n=62), time/budget constraints (79%), lack of standards/requirements to document design decisions (79%), and missing justification for extra work of documenting design decisions (79%). These forces were also perceived as being actual and the most critical barriers. Less frequently encountered barriers were the absence of prior, reusable decisions (72.6%), an uncertainty of what to document exactly (66.1%), missing benefits of reusing documented design decisions (66.1%), the disruption of the decision-making process (62.9%), and the risk of seeing decisions challenged at a later point in time (59.7%). While observed by comparatively many respondents (clearly more than the half), these forces were deemed less critical barriers.

Our quantitative results, which highlight the role of organizational and contextual barriers to DR documentation (e.g. time/budget constraints, extra work not justified) are also supported by comments of the participants. Examples are that "spending time on documenting design decisions rather than user documentation was not appreciated" and that there existed "deadlines to deliver products not docs" [5]. Furthermore, the absence of organization/project-wide standards (e.g. which DR documentation activities to perform, which details to document; see also Figures 11 and 12) as well as the lack of adequate tool support (for capturing, organizing, and retrieving design decisions; see also [109]) impede the systematic collection of DR in a reusable format.

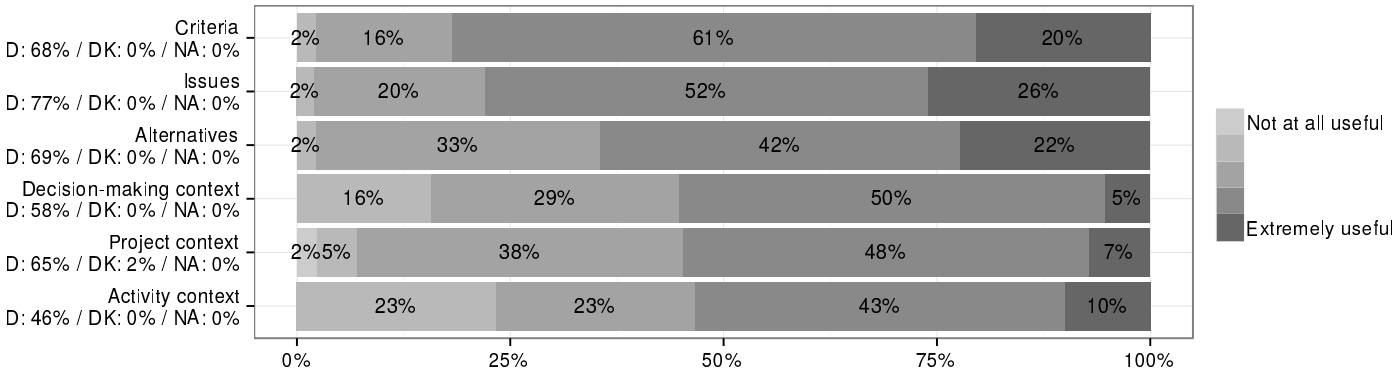

Figure 12: Number of respondents having authored selected DR documentation details (documented at least once) and the perceived usefulness of the documented details (n=65), ordered by decreasing combined levels 4 and 5 support (useful to extremely useful). D = Documented; DK = don't know; NA = not answered.

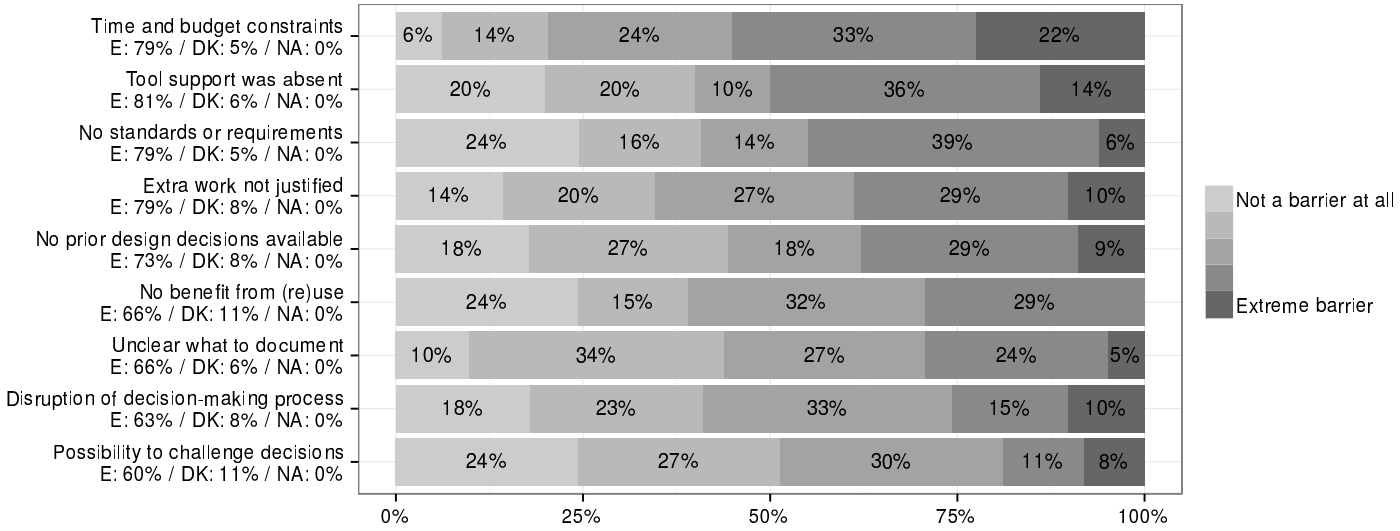

Figure 13: Number of respondents having encountered selected forces (at lease once) and the criticality of these encountered forces as perceived barrier to documenting DR (n=62), ordered by decreasing combined levels 4 and 5 support (barrier to extreme barrier). E = Encountered; DK = don't know; NA = not answered.

However, it is also obvious that DR can only pay off when it becomes available (i.e. explicitly documented) to be reused in later development projects (e.g. by saving time deciding on the best design variant). Our decision catalog is intended to serve as a reference source to be used when creating and maintaining project-specific DR. Moreover, it assists engineers by pre-structuring the design-decision space for a systematic exploration in the context of an actual DSML project (see Section 3).

7. Related Work

The related work in the fields of DR documentation and of reusable (architectural) design decisions has been elaborated on in Sections 1 and 2. In addition, our effort relates closely to a body of research describing systematic procedures for developing DSLs. Each of these approaches is based on experiences drawn from actual DSL engineering projects and provides insights into the DSL development process, into certain aspects of DSL design, or into DSL-related design decisions. For example, in [20], different DSL development activities are discussed and it is described how these activities can be combined to tailor a DSL engineering process.

In a complementary contribution, Zdun and Strembeck [32] document three main decisions to be made when applying the DSL development process from [20]. These decisions relate to the choices of a specific type of DSL development process, of a concrete syntax style, and of developing an external vs. an embedded DSL. To render these decision descriptions reusable, a pattern-like format is applied [32]. In software engineering, a pattern is a time-proven solution to a recurring design problem. A pattern description includes (at least) a "problem description", a description of the "context" in which the respective problem occurs, and one or more (alternative) "solutions". Typically, pattern descriptions also include different "forces" that may influence the choice of a certain solution, "consequences" that arise from a solution, as well as "known uses" of a particular solution. In this way, the description format we chose for the reusable design decisions resembles a pattern

format to a certain degree. However, reusable decisions are not identical to or variants of software patterns, since, for example, they list multiple solution propositions (decision options) rather than one.

While prior work on patterns for DSL development [20, 32] aims at describing generic procedures and decisions for DSL development projects, our contribution in this paper provides detailed insights into design decisions for UML-based DSLs. In this way, our work complements [20, 32], as well as other DSL development approaches such as [92, 110]. This is because our work provides a systematic and in-depth documentation of the follow-up decisions that DSL engineers face after they decided to develop a UML-based DSL.[13]

A number of other patterns and pattern languages exist that can be applied in DSL development and are thereby complementary to our work. This includes patterns for the design and implementation of DSLs [33], patterns for evolving frameworks into DSLs [111], and approaches for pattern-based DSL development [34]. Often, DSL-related patterns do not only describe how a DSL is developed, but also why it is developed in a specific way. In addition, pattern languages also describe potential sequences in which the patterns can be applied [58]. Pattern sequences compare with our notion of sets of co-adopted decision options in the sense that (ordered) option sets can represent sequences of adopted decision options.

In [36], Mernik et al. used the patterns from [33] to conduct a survey on decision factors affecting the analysis, design, and implementation phases of DSL development. These decision factors can be considered during DSL development. For example, the decision factor *Notation* deals with the consideration whether the DSL should provide a new or an existing domain notation. For a few decision factors, Mernik et al. suggest implementation guidelines. The work of [36] is complementary to ours as it focuses on general issues of design-decision making and implementation, rather than on design decisions for a specific (host) language environment such as the UML.

Another group of related work reports observations from developing DSLs in (industrial) practice. For example, Luoma et al. [37] conducted a study including 23 industrial projects for the definition of DSMLs. Similar to our approach, a number of DSLs are systematically compared. However, in contrast to our paper, Luoma et al. provide a high-level description only and do not describe specific DSL design decisions or decision-option sets in detail. Similar to patterns, lessons learned have been used as a vehicle to preserve best practices of DSL development. For example, Wile [112] reports on twelve lessons learned from three DSL experiments. For each lesson, he introduces a respective rule of thumb and gives an overview of the experiences that are the origin of the corresponding rule. Despite Wile's lessons learned being described at a comparatively high level of abstraction, they can, in general, also be observed in our work and are hence reflected in parts of the design-decisions catalog. Kelly and Pohjonen [100] present a report on worst practices found by reviewing 76 DSL development projects, and Karsai et al. [83] proposes 26 general guidelines for DSL development derived from their own experiences.

A UML-based DSL uses UML as its host language and extends the UML with domain-specific language elements and, therefore, qualifies as an embedded DSL (also: internal DSL). Related work on developing embedded DSLs includes the contributions by Günther et al. which describe a process and corresponding patterns for the development of internal DSLs on top of dynamic programming languages, such as Ruby or Python [44, 113]. Other related contributions describe how to develop DSLs from component building blocks that can be incrementally designed and composed (see, e.g., [114]). This idea originates from approaches such as keyword-based programming [115], in which so called "keywords" serve as building blocks for DSLs. In particular, a number of (universal) keywords are suggested which are then glued together to compose DSLs. This approach was first envisioned in [116] and is akin to building embedded DSLs in dynamic languages (such as Ruby, Perl, Python, or Tcl for example).

In the UML context, some authors propose approaches that define domain-specific UML extensions via UML profiles (see, e.g., [52, 53, 54]). While each of these approaches is related to our work, none of them documents reusable decisions for UML-based DSLs. The authors of [117] give an overview of standard compliant ways to define domain-specific UML extensions, while Atkinson and Kühne [118] discuss potential issues with UML profiles and suggest a solution to address these problems. Bruck and Hussey [65] present different techniques for tailoring the UML (e.g. lightweight profile or middleweight metamodel extension). In particular, Bruck and Hussey define a catalog of options and characterize different extension mechanisms accordingly. The authors also discuss pros and cons of using one approach or the other. However, Bruck and Hussey focus on UML customization techniques in general and do not integrate design decisions in the process of DSML development (e.g. no development phases are distinguished, language-model constraints as well as platform integration are not considered).

In addition, knowledge on DSL design decisions can also be gained from analyzing toolkits for DSL development. For example, Tolvanen and Kelly [119] present a tool for the definition and usage of integrated DSMLs. Similarly, Zdun [91] presents a tool suite for textual DSL-based software and provides a discussion of architectural decisions for DSL development. However, most existing contributions have a strong focus on textual domain-specific programming languages. To the best of our knowledge, there is no report reflecting on design decisions embodied in toolkits for UML-based DSML development.

In summary, the related work on patterns, best practices, and lessons learned in DSL development has in common with our approach that all are based on experiences from actual

[13] Remember that each UML-based DSL is an embedded DSL and that UML-based DSLs usually have a graphical concrete syntax or a mixture of graphical and textual concrete syntaxes.

DSL projects and contain some information on DSL design decisions and DR. Our work provides a systematic and detailed description of decision options for building UML-based DSLs. In this way, our contribution is complementary to those other approaches and can be combined with them.

8. Conclusion

In this paper, we adopt a *decision-centric perspective* on UML-based domain-specific modeling languages (DSMLs). Our focus was on providing a *tailorable* documentation [25] of *generic* and *reusable* design decisions [27]. Our work is based on a long-term research effort and complements other approaches for systematizing DSML development [20, 43], which put forth a development-process perspective.

Using a Web-based survey among MDD researchers and practitioners [5], we collected 80 expert opinions on the current practice of documenting and (re)using design rationale (DR) on UML-based DSMLs. Among others, the survey helped us to validate the description format that we used for our design-decision catalog. In particular, the *design-decision catalog* for UML-based DSMLs [21, 42] includes 35 decision options for seven decision points, covering design aspects from UML-based language-model specification to development-tool support. The reusable decisions include descriptions of positive and negative assessments of the considered options (decision drivers) as well as positive and negative effects on subsequent design decisions when adopting one or several options (decision consequences). The catalog was compiled from secondary studies as well as 80 unique DSML designs. To the best of our knowledge, our work is the first attempt to document DR on UML-based DSML development on a broad empirical basis. Thus, the catalog complements existing contributions via an evidence-based source of documented DR. As such, the design rationale documented in the catalog becomes available for reuse in new DSML projects. For example, our catalog offers a building block for documentation guidelines on extending the UML and can be used when documenting a DSML design in a systematic manner.

In our future work, we will incorporate further design rationale found on additional DSMLs. Moreover, we will proceed in engaging more MDD researchers and practitioners to collect further qualitative evidence on design-decision making (e.g., on the order of decisions) as well as to allow for a validation of documented drivers and consequences by performing interviews and participant observations based on our catalog. To this end, we will investigate ways of incorporating the reusable design decisions into design-support software. This includes design-knowledge management [120] for UML-based DSML development using DR-aware design-support tools (e.g., MetaEdit++ plus Debate Browser and Link Ability [121], Collaboro [122]).

Acknowledgement

This work was partly funded by the Austrian research funding association (FFG) under the scope of the DLUX project within the funding programme *ICT of the Future* (4th call 2015) of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), contract # 855465.

References

[1] C. Atkinson, T. Kühne, A Tour of Language Customization Concepts, Adv. Comput. 70 (2007) 105–161.

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

[3] Object Management Group, OMG Unified Modeling Language (OMG UML), available at: http://www.omg.org/spec/UML, version 2.5, formal/2015-03-01, 2015.

[4] Object Management Group, OMG Meta Object Facility (MOF) Core Specification, available at: http://www.omg.org/spec/MOF, version 2.5, formal/2015-06-05, 2015.

[5] B. Hoisl, S. Sobernig, A Survey on Documenting and Using Design Rationale when Developing Domain-specific Modeling Languages, Tech. Rep. 2016/01, WU Vienna, available at: http://epub.wu.ac.at/4920/, 2016.

[6] J. Hutchinson, J. Whittle, M. Rouncefield, Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure, Sci. Comput. Program. 89, Part B (2014) 144–161.

[7] L. Nascimento, D. L. Viana, P. A. M. S. Neto, D. A. O. Martins, V. C. Garcia, S. R. L. Meira, A Systematic Mapping Study on Domain-Specific Languages, in: Proc. 7th Int. Conf. Softw. Eng. Adv., IARIA XPS Press, 179–187, 2012.

[8] J. Hutchinson, J. Whittle, M. Rouncefield, S. Kristoffersen, Empirical Assessment of MDE in Industry, in: Proc. 33rd Int. Conf. Softw. Eng., ACM, 471–480, 2011.

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

[10] M. Staron, C. Wohlin, An Industrial Case Study on the Choice Between Language Customization Mechanisms, in: Proc. 7th Int. Conf. Product-Focused Softw. Process Improv., vol. 4034 of *LNCS*, Springer, 177–191, 2006.

[11] B. Selic, What will it take? A view on adoption of model-based methods in practice, Softw. Syst. Model. 11 (4) (2012) 513–526.

[12] G. Giachetti, B. Marín, O. Pastor, Using UML as a Domain-Specific Modeling Language: A Proposal for Automatic Generation of UML Profiles, in: Proc. 21st Int. Conf. Adv. Inform. Syst. Eng., vol. 5565 of *LNCS*, Springer, 110–124, 2009.

[13] S. Sen, N. Moha, B. Baudry, J.-M. Jézéquel, Meta-model Pruning, in: Proc. 12th Int. Conf. Model Driven Eng. Lang. Syst., vol. 5795 of *LNCS*, Springer, 32–46, 2009.

[14] A. Blouin, B. Combemale, B. Baudry, O. Beaudoux, Kompren: Modeling and Generating Model Slicers, Softw. Syst. Model. 14 (1) (2015) 321–337.

[15] X. Burgués, X. Franch, J. M. Ribó, Improving the accuracy of UML metamodel extensions by introducing induced associations, Softw. Syst. Model. 7 (3) (2008) 361–379.

[16] J. Dingel, Z. Diskin, A. Zito, Understanding and improving UML package merge, Softw. Syst. Model. 7 (4) (2008) 443–467.

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

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

[19] J. Pardillo, A Systematic Review on the Definition of UML Profiles, in: Proc. 13th Int. Conf. Model Driven Eng. Lang. Syst., vol. 6394 of *LNCS*, Springer, 407–422, 2010.

[20] M. Strembeck, U. Zdun, An Approach for the Systematic Development of Domain-Specific Languages, Softw. Pract. Exper. 39 (15) (2009) 1253–1292.
[21] B. Hoisl, S. Sobernig, Open-Source Development Tools for Domain-Specific Modeling: Results from a Systematic Literature Review, in: Proc. 49th Hawaii Int. Conf. Syst. Sciences, IEEE, 5001–5010, 2016.
[22] J. E. Burge, J. M. Carroll, R. McCall, I. Mistrík, Rationale-Based Software Engineering, Springer, 2008.
[23] A. H. Dutoit, R. McCall, I. Mistrík, B. Paech, Rationale Management in Software Engineering: Concepts and Techniques, in: Rationale Manag. in Softw. Eng., chap. 1, Springer, 1–48, 2006.
[24] R. Capilla, F. Nava, C. Carrillo, Effort Estimation in Capturing Architectural Knowledge, in: Proc. 26th IEEE/ACM Int. Conf. Automat. Softw. Eng., IEEE CS, 208–217, 2008.
[25] D. Falessi, L. C. Briand, G. Cantone, R. Capilla, P. Kruchten, The Value of Design Rationale Information, ACM Trans. Softw. Eng. Methodol. 22 (3) (2013) 21:1–21:32.
[26] J. Horner, M. Atwood, Effective Design Rationale: Understanding the Barriers, in: Rationale Manag. in Softw. Eng., chap. 3, Springer, 73–90, 2006.
[27] N. Harrison, P. Avgeriou, U. Zdun, Using Patterns to Capture Architectural Decisions, IEEE Softw. 24 (4) (2007) 38–45.
[28] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, N. Schuster, Managing architectural decision models with dependency relations, integrity constraints, and production rules, J. Syst. Softw. 82 (8) (2009) 1249–1267.
[29] O. Zimmermann, U. Zdun, T. Gschwind, F. Leymann, Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method, in: Proc. 7th Working IEEE/IFIP Conf. Softw. Archit., IEEE, 157–166, 2008.
[30] I. Lytra, S. Sobernig, U. Zdun, Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study, in: Joint Proc. 10th Working IEEE/IFIP Conf. Softw. Archit. & 6th Europ. Conf. Softw. Archit., IEEE, 111–120, 2012.
[31] I. Lytra, H. Tran, U. Zdun, Supporting Consistency Between Architectural Design Decisions and Component Models Through Reusable Architectural Knowledge Transformations, in: Proc. 7th Europ. Conf. Softw. Archit., vol. 7957 of *LNCS*, Springer, 224–239, 2013.
[32] U. Zdun, M. Strembeck, Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development, in: Proc. 14th Europ. Conf. Patt. Lang. Prog., ACM, 1–37, 2009.
[33] D. Spinellis, Notable Design Patterns for Domain-specific Languages, J. Syst. Softw. 56 (1) (2001) 91–99.
[34] C. Schäfern, T. Kuhn, M. Trapp, A Pattern-based Approach to DSL Development, in: Worksh. Proc. of Conf. Syst., Prog., and Appl.: Softw. for Hum., ACM, 39–46, 2011.
[35] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, R. Pretorius, Empirical Evidence about the UML: A Systematic Literature Review, Softw. Pract. Exper. 41 (4) (2011) 363–392.
[36] M. Mernik, J. Heering, A. Sloane, When and How to Develop Domain-specific Languages, ACM Comput. Surv. 37 (4) (2005) 316–344.
[37] J. Luoma, S. Kelly, J. Tolvanen, Defining Domain-Specific Modeling Languages: Collected Experiences, in: Proc. 4th OOPSLA Worksh. Domain-Specific Model., no. TR-33 in Computer Science and Information System Reports, University of Jyväskylä, 1–10, 2004.
[38] B. Hoisl, S. Sobernig, S. Schefer-Wenzl, M. Strembeck, 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., 303–314, 2012.
[39] E. Filtz, Systematic Literature Review and Evaluation of DSML-Design Decisions, Bachelor Thesis, WU Vienna, 2013.
[40] B. Hoisl, S. Sobernig, S. Schefer-Wenzl, M. Strembeck, 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.
[41] S. Sobernig, B. Hoisl, M. Strembeck, Extracting reusable design decisions for UML-based domain-specific languages: A multi-method study, J. Syst. Softw. 113 (2016) 140–172.
[42] B. Hoisl, S. Sobernig, M. Strembeck, A Catalog of Reusable Design Decisions for Developing UML/MOF-based Domain-specific Modeling Languages, Tech. Rep. 2014/03, WU Vienna, available at: http://epub.wu.ac.at/4815/, 2016.
[43] F. Lagarde, E. Huáscar, F. Terrier, C. André, S. Gérard, Leveraging Patterns on Domain Models to Improve UML Profile Definition, in: Proc. 11th Int. Conf. Fund. Appr. to Softw. Eng., vol. 4961 of *LNCS*, Springer, 116–130, 2008.
[44] S. Günther, Development of Internal Domain-specific Languages: Design Principles and Design Patterns, in: Proc. 18th Conf. Patt. Lang. of Prog., ACM, 1:1–1:25, 2011.
[45] B. Kitchenham, S. L. Pfleeger, Principles of Survey Research – Part 5: Populations and Samples, SIGSOFT Softw. Eng. Notes 27 (5) (2002) 17–20, ISSN 0163-5948.
[46] The American Association for Public Opinion Research, Standard Definitions: Final Dispositions of Case Codes and Outcome Rates for Surveys, AAPOR, 7th edn., 2011.
[47] A. Tang, M. A. Babar, I. Gorton, J. Han, A survey of architecture design rationale, J. Syst. Softw. 79 (12) (2006) 1792–1804, ISSN 0164-1212.
[48] M. Shahin, P. Liang, M. R. Khayyambashi, Architectural design decision: Existing models and tools, in: Joint Proc. 3rd Europ. Conf. Softw. Archit. and 8th Working IEEE/IFIP Conf. Softw. Archit., IEEE, 293–296, 2009.
[49] U. Heesch, P. Avgeriou, R. Hilliard, A documentation framework for architecture decisions, J. Syst. Softw. 85 (4) (2012) 795–820.
[50] P. Kruchten, P. Lago, H. van Vliet, Building Up and Reasoning About Architectural Knowledge, in: Proc. 2nd Int. Conf. Quality Softw. Archit., vol. 4214 of *LNCS*, Springer, 43–58, 2006.
[51] B. Hoisl, S. Sobernig, Consistency Rules for UML-based Domain-specific Language Models: A Literature Review, in: Proc. 1st Int. Worksh. UML Consistency Rules, vol. 1508 of *CEUR Worksh. Proc.*, CEUR-WS.org, 29–36, 2015.
[52] R. Paige, J. Ostroff, P. Brooke, Principles for Modeling Language Design, Inform. Softw. Tech. 42 (10) (2000) 665–675.
[53] S. Robert, S. Gérard, F. Terrier, F. Lagarde, A Lightweight Approach for Domain-Specific Modeling Languages Design, in: Proc. 35th EUROMICRO Conf. Softw. Eng. and Adv. Appl., IEEE, 155–161, 2009.
[54] B. Selic, A Systematic Approach to Domain-Specific Language Design Using UML, in: Proc. 10th IEEE Int. Sym. Object-Oriented Real-Time Distrib. Comput., IEEE, 2–9, 2007.
[55] I. Lytra, P. Gaubatz, U. Zdun, Two controlled experiments on model-based architectural decision making, Inform. Softw. Tech. 63 (2015) 58–75.
[56] U. van Heesch, P. Avgeriou, U. Zdun, N. Harrison, The supportive effect of patterns in architecture decision recovery: A controlled experiment, Sci. Comput. Program. 77 (5) (2012) 551–576.
[57] S. Sobernig, U. Zdun, Distilling Architectural Design Decisions and their Relationships using Frequent Item-Sets, in: Proc. 13th Working IEEE/IFIP Conf. Softw. Archit., IEEE, 61–70, 2016.
[58] F. Buschmann, K. Henney, D. C. Schmidt, Pattern-oriented Software Architecture – On Patterns and Pattern Languages, John Wiley & Sons, 2007.
[59] M. Strembeck, J. Mendling, Modeling Process-related RBAC Models with Extended UML Activity Models, Inform. Softw. Tech. 53 (5) (2011) 456–483.
[60] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[61] 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.
[62] B. C. Hungerford, A. R. Hevner, R. W. Collins, Reviewing Software Diagrams: A Cognitive Study, IEEE T. Softw. Eng. 30 (2004) 82–96.
[63] A. Jakšić, R. B. France, P. Collet, S. Ghosh, Evaluating the Usability of a Visual Feature Modeling Notation, in: Proc. 7th Int. Conf. Softw. Lang. Eng., no. 8706 in LNCS, Springer, 122–140, 2014.
[64] A. Classen, Q. Boucher, P. Heymans, A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL, Sci. Comput. Program. 76 (12) (2011) 1130–1143.
[65] J. Bruck, 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: Feb 9, 2017., IBM, 2008.
[66] G. Kahraman, S. Bilgen, A framework for qualitative assessment of domain-specific languages, Softw. Syst. Model. 14 (4) (2015) 1505–1526, ISSN 1619-1374.
[67] Object Management Group, Object Constraint Language, available at: http://www.omg.org/spec/OCL, version 2.4, formal/2014-02-03, 2014.
[68] D. Kolovos, L. Rose, A. García-Domínguez, R. Paige, The Epsilon Book, available at: http://www.eclipse.org/epsilon/doc/book/, 2017.
[69] A. Demuth, R. E. Lopez-Herrejon, A. Egyed, Supporting the Co-evolution of Metamodels and Constraints through Incremental Constraint Management, in: Proc. 16th Int. Conf. Model Driven Eng. Lang. Syst., vol. 8107 of *LNCS*, Springer, 287–303, 2013.
[70] B. Hoisl, S. Sobernig, M. Strembeck, Natural-language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages, in: Proc. 2nd Int. Conf. Model-Driven Eng. Softw. Dev., SciTePress, 356–367, 2014.
[71] D. S. Kolovos, R. F. Paige, F. A. Polack, Aligning OCL with Domain-Specific Languages to Support Instance-Level Model Queries, Electron. Commun. EASST 5.
[72] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, Addison-Wesley, 2nd edn., 2008.
[73] Object Management Group, XML Metadata Interchange (XMI) Specification, available at: http://www.omg.org/spec/XMI, version 2.5.1, formal/2015-06-07, 2015.
[74] K. Hassam, S. Sadou, V. L. Gloahec, R. Fleurquin, Assistance System for OCL Constraints Adaptation during Metamodel Evolution, in: Proc. 15th Europ. Conf. Softw. Maint. and ReEng., IEEE, ISSN 1534-5351, 151–160, 2011.
[75] D. Chiorean, V. Petraşcu, D. Petraşcu, How My Favorite Tool Supporting OCL Must Look Like, Electron. Commun. EASST 15.
[76] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, M. Wimmer, Systematic Co-Evolution of OCL Expressions, in: Proc. 11th Asia-Pacific Conf. Conceptual Model., vol. 165, ACS, 33–42, 2015.
[77] Object Management Group, Service oriented architecture Modeling Language (SoaML) Specification, available at: http://www.omg.org/spec/SoaML, version 1.0.1, formal/2012-05-10, 2012.
[78] Object Management Group, OMG Systems Modeling Language (OMG SysML), available at: http://www.omg.org/spec/SysML, version 1.4, formal/2015-06-03, 2015.
[79] M. Richters, M. Gogolla, OCL: Syntax, Semantics, and Tools, in: Object Model. with the OCL, vol. 2263 of *LNCS*, Springer, 447–450, 2002.
[80] D. Moody, 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., vol. 5452 of *LNCS*, Springer, 16–34, 2009.
[81] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, Text-based Modeling, CoRR abs/1409.6623.
[82] M. Fowler, Language Workbenches: The Killer-App for Domain Specific Languages?, available at: http://martinfowler.com/articles/languageWorkbench.html, last accessed: Feb 9, 2017, 2005.
[83] G. Karsai, H. K. C. Pinkernell, B. Rumpe, M. Schindler, S. Völkel, Design Guidelines for Domain Specific Languages, in: Proc. 9th OOPSLA Worksh. Domain-Specific Model., 7–13, 2009.
[84] 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. Last accessed: Feb 9, 2017, iSO/IEC 13568:2002, 2002.
[85] Object Management Group, Semantics of a Foundational Subset for Executable UML Models (fUML), available at: http://www.omg.org/spec/FUML, version 1.2.1, formal/2016-01-05, 2016.
[86] 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. Last accessed: Feb 9, 2017, iSO/IEC 13568:2002/Cor 1:2007, 2007.
[87] Object Management Group, Action Language for Foundational UML (ALF): Concrete Syntax for a UML Action Language, available at: http://www.omg.org/spec/ALF, version 1.0.1, formal/2013-09-01, 2013.
[88] K. Czarnecki, S. Helsen, Feature-based Survey of Model Transformation Approaches, IBM Syst. J. 45 (3) (2006) 621–645.
[89] T. Mens, P. v. Gorp, A Taxonomy of Model Transformation, Electron. Notes Theor. Comput. Sci. 152 (2006) 125–142.
[90] K. Czarnecki, S. Helsen, Classification of Model Transformation Approaches, in: Proc. 2nd OOPSLA Worksh. Genera. Tech. in the Context of Model Driven Archit., 2003.
[91] U. Zdun, A DSL Toolkit for Deferring Architectural Decisions in DSL-based Software Design, Inform. Softw. Tech. 52 (9) (2010) 733–748.
[92] T. Stahl, M. Völter, Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2006.
[93] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, 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.
[94] L. M. Rose, N. Matragkas, D. S. Kolovos, R. F. Paige, A Feature Model for Model-to-Text Transformation Languages, in: Proc. 4th Int. Worksh. Model. in Softw. Eng., IEEE, 57–63, 2012.
[95] F. Lagarde, H. Espinoza, F. Terrier, S. Gérard, Improving UML Profile Design Practices by Leveraging Conceptual Domain Models, in: Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng., ACM, 445–448, 2007.
[96] D. Jackson, Software Abstractions: Logic, Language, and Analysis, MIT Press, 2012.
[97] P. Mohagheghi, V. Dehlen, Where Is the Proof? – A Review of Experiences from Applying MDE in Industry, in: Proc. 4th Europ. Conf. Model Driven Archit. – Found. and Appl., vol. 5095 of *LNCS*, Springer, 432–443, 2008.
[98] A. Khalaoui, A. Abran, E. Lefebvre, DSML Success Factors and their Assessment Criteria, Metrics News 13 (1) (2008) 43–51.
[99] R. Schaefer, A Survey on Transformation Tools for Model Based User Interface Development, in: Human-Comput. Interact. Interact. Des. Usability, vol. 4550 of *LNCS*, Springer, 1178–1187, 2007.
[100] S. Kelly, R. Pohjonen, Worst Practices for Domain-Specific Modeling, IEEE Softw. 26 (4) (2009) 22–29.
[101] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, R. Heldal, A taxonomy of tool-related issues affecting the adoption of model-driven engineering, Softw. Syst. Model. 16 (2) (2017) 313–331.
[102] A. Yie, R. Casallas, D. Deridder, D. Wagelaar, Realizing Model Transformation Chain Interoperability, Softw. Syst. Model. 11 (1) (2012) 55–75.
[103] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, Evaluating and comparing language workbenches: Existing results and benchmarks for the future, Comput. Lang. Syst. Str. 44, Part A (2015) 24–47, ISSN 1477-8424.
[104] S. Cook, Looking back at UML, Softw. Syst. Model. 11 (4) (2012) 471–480.
[105] B. Henderson-Sellers, C. Gonzalez-Perez, Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0, in: Proc. 9th Int. Conf. Model Driven Eng. Lang. Syst., vol. 4199 of *LNCS*, Springer, 16–26, 2006.
[106] B. Kitchenham, S. L. Pfleeger, Principles of Survey Research – Part 4: Questionnaire Evaluation, SIGSOFT Softw. Eng. Notes 27 (3) (2002) 20–23, ISSN 0163-5948.
[107] J. Singer, S. E. Sim, T. C. Lethbridge, Software Engineering Data Collection for Field Studies, in: Guide to Adv. Empir. Softw. Eng., Springer, 9–34, 2008.
[108] A. MacLean, R. M. Young, V. M. E. Bellotti, T. P. Moran, Questions, Options, and Criteria: Elements of Design Space Analysis, in: Design Rationale: Concepts, Techniques, and Use, chap. 3, Lawrence Erlbaum Associates, 53–106, 1996.
[109] W. C. Regli, X. Hu, M. Atwood, W. Sun, A Survey of Design Rationale Systems: Approaches, Representation, Capture and Retrieval, Eng. Comput. 16 (3) (2000) 209–235.
[110] M. Völter, DSL Engineering – Designing, Implementing, and Using

Domain-Specific Languages, Amazon Distribution, 2013.
[111] D. Roberts, R. Johnson, Patterns for Evolving Frameworks, in: Patt. Lang. of Program Design 3, Addison-Wesley, 471–486, 1997.
[112] D. Wile, Lessons learned from real DSL experiments, Sci. Comput. Program. 51 (3) (2004) 265–290.
[113] S. Günther, M. Haupt, M. Splieth, Agile Engineering of Internal Domain-Specific Languages with Dynamic Programming Languages, in: Proc. 5th Int. Conf. Softw. Eng. Adv., IEEE, 162–168, 2010.
[114] N. Allen, C. Shaffer, L. Watson, Building Modeling Tools that Support Verification, Validation, and Testing for the Domain Expert, in: Proc. 37th Winter Simul. Conf., IEEE, 419–426, 2005.
[115] T. Cleenewerck, Component-based DSL Development, in: Proc. 2nd Int. Conf. Genera. Prog. and Compon. Eng., Springer, 245–264, 2003.
[116] P. Landin, The Next 700 Programming Languages, Commun. ACM 9 (3) (1966) 157–166.
[117] I. Weisemöller, A. Schürr, A Comparison of Standard Compliant Ways to Define Domain Specific Languages, in: Worksh. Proc. 10th Int. Conf. Model Driven Eng. Lang. Syst., vol. 5002 of *LNCS*, Springer, 47–58, 2008.
[118] C. Atkinson, T. Kühne, Profiles in a strict metamodeling framework, Sci. Comput. Program. 44 (1) (2002) 5–22.
[119] J.-P. Tolvanen, S. Kelly, MetaEdit+: Defining and using integrated domain-specific modeling languages, in: Proc. 24th ACM SIGPLAN Conf. Companion on Object Oriented Prog. Syst. Lang. Appl., ACM, 819–820, 2009.
[120] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, M. A. Babar, A comparative study of architecture knowledge management tools, J. Syst. Softw. 83 (3) (2010) 352–370.
[121] H. Oinas-Kukkonen, Method rationale in method engineering and use, in: S. Brinkkemper, K. Lyytinen, R. J. Welke (Eds.), Method Engineering: Principles of method construction and tool support, Springer, Boston, MA, 87–93, 1996.
[122] J. L. C. Izquierdo, J. Cabot, J. J. López-Fernández, J. S. Cuadrado, E. Guerra, J. de Lara, Engaging End-Users in the Collaborative Development of Domain-Specific Modelling Languages, in: Proc. 10th Int. Conf. Cooperative Design Vis. Eng., vol. 8091 of *LNCS*, Springer, 101–110, 2013.

A. Excerpt from Decision Catalog

This is the actual reusable decision on implementation options for a language model based on the UML (D2) as found in [42], pp. 16–19. The motivating example in Section 3 refers to the details of this decision.

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., [65, 3]).

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 diagrams, 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 [3, 4]. 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 [3, 4]. 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., [143, 144]). Similarly, stereotype definitions can be provided to accompany a metamodel extension/-modification (see, e.g., [126]).

Decision drivers. An overview of positive and negative links between decision drivers and available options is shown in Table A.3.

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., [10]). Therefore, profile constraints may only define well-formed rules that are more constraining (but consistent with) those specified by the metamodel [3]. 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).

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. 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., [10]).

Table A.3: 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). 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).

Application. In all our DSML projects, we formalized the language models as metamodel extensions (O2.3). Additionally, profiles (O2.2) were employed in [145, 123, 124, 143, 144, 126, 146]. Therefore, we effectively adopted combined strategies. In related approaches, we also found the application of M1 structural models (O2.1, e.g., in [147]) and the modification of the UML metamodel (O2.4, e.g., in [148]) for the formalization of the language model. As an example for O2.4, [148] 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 [149], existing classes from the UML metamodel (e.g. `UseCase`) are associated with newly defined classes (e.g. `UseCaseDescription`). The metamodel definition in [149] 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 [3]. Furthermore, accompanying textual annotations can provide clarifying details.

Sketch. Figure A.14 depicts an excerpt from a UML extension (taken from [143, 144]). 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 [143, 150, 144]) with the SoaML specification [77].

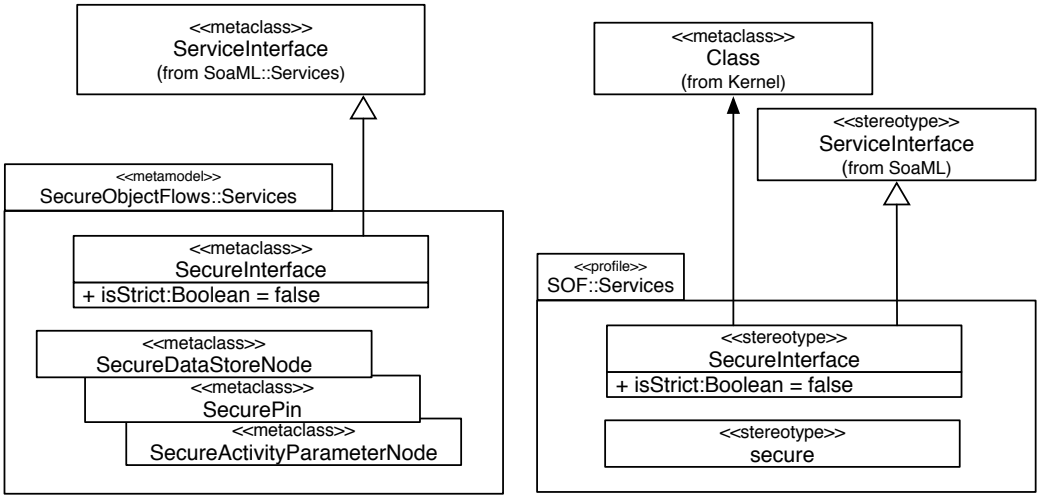

Figure A.14: Exemplary UML metamodel extension and profile definition [143].

B. DSML Papers

This is the subset of publications which were reviewed for documenting the reusable design decisions and which are referenced in this paper. The complete list is provided in a companion to this paper [42].

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

[124] S. Schefer-Wenzl, M. Strembeck, An approach for consistent delegation in process-aware information systems, in: Proc. 15th Int. Conf. Bus. Inform. Syst., Vol. 117 of LNBIP, Springer, 2012, pp. 60–71.

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

[126] B. Hoisl, M. Strembeck, A UML extension for the model-driven specification of audit rules, in: Proc. 2nd Int. Worksh. Inform. Syst. Secur. Eng., Vol. 112 of LNBIP, Springer, 2012, pp. 16–30.

[127] J. E. Pérez-Martínez, A. Sierra-Alonso, From analysis model to software architecture: A PIM2PIM mapping, in: Proc. 2nd Europ. Conf. Model Driven Archit. – Found. and Appl., Vol. 4066 of LNCS, Springer, 2006, pp. 25–39.
[128] F. Aoussat, M. Oussalah, M. Nacer, SPEM extension with software process architectural concepts, in: Proc. 35th Annu. IEEE Int. Conf. Comp. Softw. and Appl., IEEE, 2011, pp. 215–223.
[129] U. Zdun, P. Avgeriou, Modeling architectural patterns using architectural primitives, in: Proc. 20th Annu. ACM SIGPLAN Conf. Object-oriented Prog., Syst., Lang., Appl., ACM, 2005, pp. 133–146.
[130] I. Ivkovic, K. Kontogiannis, A framework for software architecture refactoring using model transformations and semantic annotations, in: Proc. 10th Europ. Conf. Softw. Maint. and ReEng., IEEE, 2006, pp. 135–144.
[131] R. Bendraou, M.-P. Gervais, X. Blanc, UML4SPM: A UML2.0-based metamodel for software process modelling, in: Proc. 8th Int. Conf. Model Driven Eng. Lang. Syst., Vol. 3713 of LNCS, Springer, 2005, pp. 17–38.
[132] A. Cicchetti, D. D. Ruscio, A. Pierantonio, A metamodel independent approach to difference representation, J. Object Technol. 6 (9) (2007) 165–185.
[133] K. Berkenkötter, U. Hannemann, Modeling the railway control domain rigorously with a UML 2.0 profile, in: Proc. 25th Int. Conf. Comput. Safety, Reliab., Secur., Vol. 4166 of LNCS, Springer, 2006, pp. 398–411.
[134] E. Cariou, C. Ballagny, A. Feugas, F. Barbier, Contracts for model execution verification, in: Proc. 7th Europ. Conf. Model Driven Archit. – Found. and Appl., Vol. 6698 of LNCS, Springer, 2011, pp. 3–18.
[135] A. Queralt, E. Teniente, A platform independent model for the electronic marketplace domain, Softw. Syst. Model. 7 (2) (2008) 219–235.
[136] J. Jürjens, Secure Systems Development with UML, Springer, 2005.
[137] I.-C. Hsu, Extending UML to model Web 2.0-based context-aware applications, Softw. Pract. Exper. 42 (10) (2012) 1211–1227.
[138] G. M. Kapitsaki, D. A. Kateros, G. N. Prezerakos, I. S. Venieris, Model-driven development of composite context-aware web applications, Inform. Softw. Tech. 51 (8) (2009) 1244–1260.
[139] S. Ali, L. C. Briand, H. Hemmati, Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems, Softw. Syst. Model. 11 (4) (2012) 633–670.
[140] G. N. Rodrigues, D. S. Rosenblum, S. Uchitel, Reliability prediction in model-driven development, in: Proc. 8th Int. Conf. Model Driven Eng. Lang. Syst., Vol. 3713 of LNCS, Springer, 2005, pp. 339–354.
[141] E. L. Alves, P. D. Machado, F. Ramalho, Automatic generation of built-in contract test drivers, Softw. Syst. Model. 13 (3) (2012) 1141–1165.
[142] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, On challenges of model transformation from UML to Alloy, Softw. Syst. Model. 9 (1) (2010) 69–86.
[143] B. Hoisl, S. Sobernig, Integrity and confidentiality annotations for service interfaces in SoaML models, in: Proc. Int. Worksh. Secur. Aspects of Process-aware Inform. Syst., IEEE, 2011, pp. 673–679.
[144] B. Hoisl, S. Sobernig, M. Strembeck, Modeling and enforcing secure object flows in process-driven SOAs: An integrated model-driven approach, Softw. Syst. Model. 13 (2) (2014) 513–548.
[145] M. Strembeck, U. Zdun, Modeling interdependent concern behavior using extended activity models, J. Object Technol. 7 (6) (2008) 143–166.
[146] U. Zdun, M. Strembeck, Modeling composition in dynamic programming environments with model transformations, in: Proc. 5th Int. Sym. Softw. Compos., Vol. 4089 of LNCS, Springer, 2006, pp. 178–193.
[147] C. Song, E. Cho, C. Kim, An integrated GUI-business component modeling method for the MDD- and MVC-based hierarchical designs, Int. J. Softw. Eng. Know. 21 (3) (2011) 447–490.
[148] A. M. R. da Cruz, J. a. P. Faria, A metamodel-based approach for automatic user interface generation, in: Proc. 13th Int. Conf. Model Driven Eng. Lang. Syst., Vol. 6394 of LNCS, Springer, 2010, pp. 256–270.
[149] S. S. Somé, A meta-model for textual use case description, J. Object Technol. 8 (7) (2009) 87–106.
[150] B. Hoisl, M. Strembeck, Modeling support for confidentiality and integrity of object flows in activity models, in: Proc. 14th Int. Conf. Bus. Inform. Syst., Vol. 97 of LNBIP, Springer, 2011, pp. 278–289.