Open-Source Development Tools for Domain-Specific Modeling: Results from a Systematic Literature Review

Bernhard Hoisl and Stefan Sobernig
Institute for Information Systems and New Media, WU Vienna
{bernhard.hoisl, stefan.sobernig}@wu.ac.at

Abstract

Domain-specific modeling languages (DSMLs) are key to empowering organizational experts in interacting with business information systems. For this task, developing DSMLs based on the Unified Modeling Language (UML) has become a popular option. However, a number of design decisions must be considered when developing UML-based DSMLs. In this paper, we extend a systematic literature review (SLR) to cover decisions with respect to model-driven development (MDD) tooling. We extract tooling information from the selected SLR papers and classify these software tools supporting DSML development. Using this classification, we report on usage frequencies for MDD-based tools as well as on their corresponding open- and closed-source license models. Results indicate that closed-source tools are mostly employed as editors for the language model and the diagrammatic syntax of a DSML only. For DSML development aspects other than these, primarily open-source tools are utilized (e.g., for constraint evaluation, model transformation).

1. Introduction

Business information systems are software systems in support of managerial decisions, organizational processes, and organizational capabilities. Emerging business information systems, which are based on enterprise wikis and enterprise mashups, build on the idea of incorporating non-technical business experts into analyzing, designing, and implementing software-based information systems—beyond mere spreadsheet programming [1]. Key to this idea are special-purpose software languages (*domain-specific languages*, DSLs) which empower organizational experts in interacting with a business information system in terms of a continued re-design.

Domain-specific modeling languages (DSMLs) are specialized modeling languages tailored primarily for graphical modeling tasks in a particular application domain, supporting the model-driven development (MDD) of software systems for this domain. As a special kind of DSLs, DSMLs provide end users with at least one graphical or diagrammatic concrete syntax; in contrast to textual or form-/table-based DSLs, for instance (see, e.g., [2], [3]).

In the context of MDD, developing DSMLs based on and integrated with the Unified Modeling Language (UML [4]) and the Meta Object Facility (MOF [5]) has become a widely used option (see, e.g., [6], [7]). There exists a large amount of software tools implementing the respective specifications of the Object Management Group (OMG), thereby fulfilling requirements for MDD and providing infrastructures for metamodeling, model transformation, model analysis etc. (e.g., IBM Rational Software Architect, Sparx Systems Enterprise Architect).[1] As in all MDD-related projects, supporting the development of MOF/UML-based DSMLs with a comprehensive tool environment is crucial (see, e.g., [7], [8]). In general, researchers have discussed MDD tooling as a key barrier to MDD adoption (see [9] for a recent overview).

When developing UML-based DSMLs, a number of design decisions must be considered. On the one hand, design decisions relate to specifics of a language design, for example, the language-model definition, concrete-syntax design, and platform integration [10]. On the other hand, design decisions are concerned with the application domain of the business information system, the architectural styles and patterns used to construct it, as well as adoption of software tooling and infrastructure [11]. To support design-decision making for DSML development, we conducted a systematic literature review (SLR) to extract design decisions and corresponding options from related work and documented them in a catalog for their generic reuse ([12]–[14]).

One important design decision when developing UML-based DSMLs is the choice of adopting appropriate MDD tools. When a DSML should not serve documentation purposes only, the objective is to automate the creation of its language model, the evaluation of language-model constraints, the transformation of models to platform-specific software artifacts (e.g., source code), and so forth. As a DSML consists of multiple interrelated artifacts (models, model transformations etc.), the availability of an integrated development-tool environment becomes crucial. Furthermore, the fundamental choice of selecting open- or closed-source tools has consequences regarding maintenance, support, and costs for the business information system containing the DSML. For example, Whittle et al. [9] present first evidence on the role of direct (licensing) and indirect costs of MDD tool adoption (e.g. tool training).

Prior empirical research on MDD tool usage (e.g., [9], [15]–

1. For example, a comprehensive list of UML/MOF-enabled tools is provided at https://www.dmoz.org/Computers/Programming/Methodologies/Modeling_Languages/Unified_Modeling_Language/Tools/ (accessed on 2015-09-15).

1530-1605/16 $31.00 © 2016 IEEE
DOI 10.1109/HICSS.2016.620

IEEE computer society

[17]) has not addressed UML-based DSML development, but rather MDD in general or for certain application domains (e.g., HCI). In addition, prior work has not assessed tool usage over a larger time window and in the context of closely related design decisions (e.g., platform integration).

This paper aims at closing this gap by evaluating related work found via the SLR with respect to MDD tooling support. The contribution of this paper are threefold. By extracting tooling information from the papers retrieved via the SLR, we (1) construct a classification system for software tools supporting DSML development. Using this classification, we report on (2) usage frequencies for MDD-based tools as well as on (3) corresponding open- and closed-source license models.

The paper is structured as follows. Section 2 summarizes our approach to systematically develop DSMLs, presents decision points involved in DSML development, and summarizes our SLR study. Results of captured tooling decisions are reported in Section 3. Section 4 discusses the findings of our study, especially with regard to software tooling and licensing usage frequencies. Related work is presented in Section 5 and Section 6 concludes the paper.

2. Background

2.1. DSML Development Process

DSML development is an exploratory, iterative process. A process view (such as [10]) treats DSML development as a complex flow of characteristic development activities. The interdependencies between the development activities result from their ordering and the input/output artifacts (e.g., modeling artifacts, behavior specification etc.) that serve as input and output to the different development activities. With such a process vocabulary, common development styles for DSMLs can be documented as different flows between these development activities. We discriminate between the following four development phases [10]:

Define DSML language model: One first defines an initial core language model and the corresponding language model constraints for the selected target domain. By following a domain analysis method, domain abstractions are identified and form the language model of a DSML. Because the language model often cannot capture all restrictions and/or semantic properties of the DSML elements, language model constraints are added, if necessary. This phase results in the *DSML language model* and accompanying *DSML language model constraints*.

Define DSML concrete syntax: In this phase, suitable notation symbols as well as composition and production rules are defined. One uses the DSML language model and the DSML language model constraints as input to produce the *DSML concrete syntax specification*.

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

DSML platform integration: All artifacts defined for a DSML are mapped to the features of a selected software platform by either extending an existing platform or by developing a new tool set. A common technique for platform integration is achieved by defining model transformations (see, e.g., [18]) to convert a model into another platform-specific model (model-to-model transformation, M2M) or into executable software artifacts (model-to-text transformation, M2T).

The order of these four activities performed during DSML development largely depends on the DSML's development context. In our work, we focus on a *language-model driven* process variant [10]. In this case, the DSML engineering process is driven by the language model definition. First, the language model is defined for the relevant domain concepts, then the concrete syntax and corresponding behavior are specified. Finally, the DSML is mapped to a platform.

The process view, however, abstracts from DSML development as a decision-making process, with each activity being a decision point which opens up a considerable decision space of analysis, modeling, and implementation instruments. Each decision point refers to a number of options (e.g., extension techniques). The specific decision context is set by the decision point's position in the overall activity flow (i.e. previously taken decisions) and the available/required input artifacts. This context and certain forces (design drivers) then lead to the adoption of one or several options. The decisions taken at each point have consequences by producing output artifacts as well as by widening or restricting the decision space at subsequent decision points ([12], [14]).

Given that development decisions must be taken repeatedly for each DSML development project against fairly similar decision spaces (e.g., tooling), a systematic reuse of previously gained process and decision knowledge (e.g., options, drivers, consequences) comes to mind. To sum up, a decision-making view on DSML development complements the process view [10] by documenting DSML development at a finer grained level of abstraction and by making decision interdependencies explicit—irrespective of the actual development process style adopted.

2.2. Decision Points in DSML Development

For the process of developing a DSML, until now, we empirically identified six different decision points ([12], [14]) which can be conducted in different sequences depending on the development style used and the intention behind developing the DSML [10]. This paper adds a seventh decision point (*tool support*; see Fig. 1) complementing our catalog of reusable design decisions for developing UML-based DSMLs [14]. The seven decision points are as follows:

Language-model definition: After a systematic analysis and a structuring of the respective language domain, one identifies the domain abstractions to be represented by a DSML. In this context, one of the main questions is how one describes these

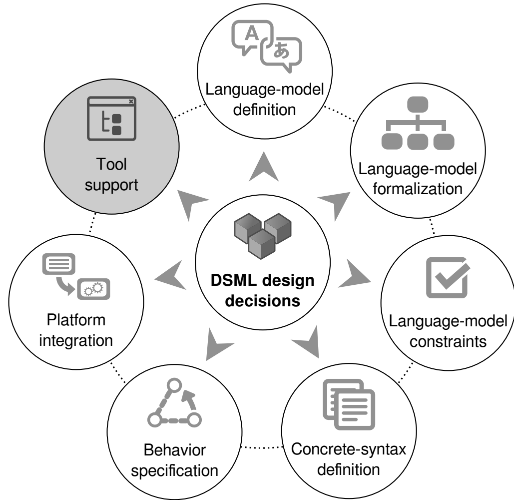

Fig. 1. Seven DSML design-decision points.

domain abstractions to arrive at a comprehensive and comprehensible language model to be used as the basis for developing the DSML. Corresponding options are the specification via a (formal) textual description, via formal or informal (graphical) models, or through a combination of these options.

Language-model formalization: At this decision point, it is determined how a language model defined informally or defined independently from the UML is turned into a *formal UML model*. By formal model, we refer to a realization of the language model using a well-defined metamodeling language such as the UML metamodeling infrastructure. A metamodeling language is itself based on a well-defined and well-documented language model (i.e. CMOF for the UML metamodel [5]) and provides at least one well-defined and well-documented concrete syntax to define an own language model (e.g., the CMOF diagram syntax to specify a UML metamodel extension). At this decision point, decision options are the definition of an M1 structural model, a UML profile, a UML metamodel extension, a UML metamodel modification, or a combination of these options.

Language-model constraints: 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 language-model formalization could be incomplete or ambiguous. To prevent this, one can specify special-purpose language-model constraints, for instance via a constraint-language (such as the OCL), code/textual annotations, M2M/M2T transformations, or a combination of these options.

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

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

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

Tool support: Adequate tool support for the development of DSMLs is essential. For instance, the generative nature of MDD makes model-transformation engines a key building block of most DSML approaches (see, e.g., [8], [16], [19]). In addition, the choice of a particular MDD tool chain may affect other DSML design decisions because not all decision options (e.g. concrete-snytax options) might be supported by a given toolkit (see, e.g., [12], [14], [20]). Decision options requiring direct tool support are language-model and diagrammatic-/textual-syntax editors as well as constraints-evaluation, model-execution, M2M/M2T transformation, and process-orchestration engines. Besides, MDD-tooling decisions incur direct (licensing) and indirect costs (e.g. tool training [9]).

Fig. 2 shows the timely ordering of decision points for the most common DSML development process variant—as experienced by us while developing our own DSMLs [21] and while extracting design decision from the included articles of the SLR ([12], [14]): a language-model driven DSML development activity (see also Section 2.1). For the definition of the DSML language model, decisions must be taken regarding the specification of the language model, its UML-based formalization, and accompanying constraints. Subsequent two decisions target the definition of a concrete-syntax style and the behavior of the DSML. The last activity in the process maps the DSML to a certain platform for which a decision for a dedicated platform-integration method must be made. All of the decision points involved in the DSML development process may be supported by tools (e.g., a language-model editor or a M2T transformation engine; for details see Section 3.1). Thus, the *tool support* decision point has interdependencies with all other six decision points at every phase of the DSML development activity (see Fig. 2).

2.3. Systematic Literature Review

This section summarizes our systematic literature review (SLR) conducted to distill generic DSML design decisions from UML-based DSML design documents (details are published in [12], [13]). The main goal of the SLR was to identify a maximum number of scientific publications which document design decisions on UML-based DSMLs as primary sources.

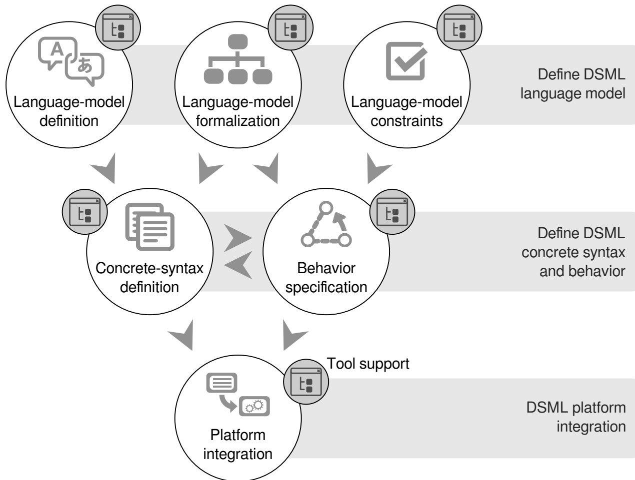

Fig. 2. Sequence of decision points in a language-model driven DSML development process.

The SLR was performed in three steps. First, to provide a basis for evaluation of the search procedure, we established a corpus of reference publications as quasi-gold standard (QGS [22]). Based on this reference corpus, we identified the search engines for an automated publication search. Second, we performed the actual engine-based publication search. Based on the bibliographical records extracted from publications selected up to this point, we then performed a backward-snowballing search. Backward snowballing is the practice of manually identifying additional publications for selection from the reference lists (citations) of a given set of publications [23]. We selected publications for inclusion and assessed their quality based on predefined criteria at each stage.

Quasi-gold standard (QGS): To guide publication search, and to report the search validity, we used a QGS corpus of publications [22]. In essence, a QGS is a set of systematically hand-picked publications considered relevant for a specific SLR. As a starting point for establishing our QGS, we considered all publications collected from the following three sources: Our own DSML publications [21], the third-party publications from our pilot review [24], and the third-party publications returned by a mapping study [6]. The three sources accounted for 159 papers and 119 publication venues in total. In a pair session, the 119 publication venues were filtered by two authors according to defined inclusion/exclusion criteria resulting in 17 venues fulfilling the conditions. In a next step, two authors manually screened the selected 17 venues. This involved 418 journal volumes and 80 proceeding issues published between 2005–2012. The screening result was a collection of 83 articles. Each article was rated for inclusion into the QGS corpus independently by two authors. 37 publications were positively rated by both authors and formed the final QGS corpus. Based on these QGS publications, the relevant search engines for the automated search were identified (SpringerLink, IEEE Xplore, Scopus, and ACM Digital Library) and a search string for the automated search was constructed. See [12], [13] for a list of the 37 QGS publications and all procedural details.

Main search: In this step, we conducted the automated publication search using the search string developed in the previous step on the four selected search engines. Details of conducting the main search (search execution, duplicate cleansing, validity computation, QGS-based capping, paper evaluation, quality assessment, publication-data extraction) are reported in [12], [13]. The search execution yielded 5,778 search hits split into four result sets, one for each of the search engines. After enforcing the QGS-based capping, having evaluated the papers based on our selection criteria and having completed the quality assessment, 73 papers representing 1.3% of the original search hits remained. For this final publication set, we extracted the publication-specific data. These items included bibliographical entries, selection decision for each hit as well as decision-mining entries (application domain(s), relevant UML diagram type(s), decision options with respect to each of the six decision points; see Section 2.2).

Backward snowballing: To incorporate prior work considered relevant by the authors of the 73 included papers, we performed a manual, citation-based search using the bibliographical references taken from these papers. Again, details of conducting the snowballing procedure (manual search, paper evaluation, quality assessment, publication-data extraction) are reported in [12], [13]. Via the backward snowballing search [23], we reviewed a total of 2,337 references. After evaluation and quality assessment of the papers, eight were included into the paper corpus. From the eight additional publications, we extracted bibliographical metadata and coded their design-decision data (in the same way as was done for papers retrieved by the main search).

We considered a total of 81 articles as relevant: 73 from main search plus eight from snowballing. To complete the paper corpus, we re-considered the QGS publications not retrieved by the main and the snowballing searches for inclusion based on the selection criteria. This way, we classified two QGS journal articles and one QGS conference article as relevant. We so arrived at a paper corpus of *84* publications. The corpus was composed of 54 conference articles (64%) and 30 journal articles (36%).

3. Results: Captured Tooling Decisions

In this section, we report on the results of the SLR regarding DSML tooling support. We group identified tooling support into eight categories (Section 3.1), present tooling and license information extracted from the papers (Section 3.2), describe tooling support for each of the design-decision points (Section 3.3), and discuss limitations of our study (Section 3.4).

3.1. Classification of Tooling Support

The included papers of the SLR supported, for example, the creation, the analysis, or the execution of different DSML

artifacts via dedicated tools. In total, after evaluating all 84 papers included in our corpus, we extracted 180 references to tools supporting the development of DSMLs. The eight categories of tooling support are the following, with each category found represented by a tool nomination in at least one of the 84 papers.

TO.1 Language-model editors: An editor is used to create, to edit, and to 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).

TO.2 Constraints evaluators: A constraints evaluator is used to automatically analyze and to 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 Model Development Tools, MDT).

TO.3 Generators for diagrammatic-syntax editors: The representation of a DSML's graphical concrete syntax is supported via an editor. The tool allows 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).

TO.4 Generators for textual-syntax editors: The representation of a DSML's textual concrete syntax is supported via an editor. The tool (e.g., Eclipse Xtext) allows for creating, editing, and maintaining tailored editors for the domain-specific models textually (i.e. textual domain-specific languages, DSLs).

TO.5 Model-execution engines: A model-execution engine is 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 [25]). This requires that the target software platform (and its DSML-specific functions) can be accessed through the same programming language which is used to represent the internal, programmatic DSML model structure. In particular, this option (re)uses and/or extends an existing interpreter or compiler infrastructure for the execution of DSML models.

TO.6 Model-to-model transformation engines: A M2M transformation engine takes one or multiple models as input and generates one or multiple models as output. An editor supports creating, editing, and maintaining transformation specifications in a dedicated transformation language (e.g., Epsilon Transformation Language, ETL).

TO.7 Model-to-text transformation engines: A M2T transformation engine takes one or multiple models as input and generates one or multiple textual artifacts as output. An editor supports creating, editing, and maintaining transformation expressions in a dedicated transformation language (e.g., Epsilon Generation Language, EGL).

TO.8 Orchestration engines: 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. An orchestration engine coordinates the execution process as well as data input/output requirements of these artifacts providing a MDD-based tool chain for DSML development (e.g., Eclipse Modeling Workflow Engine, MWE).

3.2. Extracted Tooling and License Information

In previous work ([12], [14]), for each DSML design found documented in the publications of our paper corpus, we identified and recorded the decision options for six decision points (see Section 2.2). This yielded one decision-option set per DSML which we now complement with data extracted from our paper corpus regarding the decision of tooling support for each DSML design (our seventh decision point).

To decide whether a particular decision option was present in a given DSML design, we read and studied the corresponding publication as the primary design document for cues on each decision option. In addition, we considered all auxiliary design-documentation artifacts contained (e.g., diagrams) or referenced by the publication at hand, if fully accessible. In particular, important artifacts to extract information about supporting tools included implementation artifacts, such as, language-model, constraint-language, and model-transformation specifications.

From the above sources, we extracted tooling information for each of the eight categories (see Section 3.1). 28 out of the total 84 DSML artifacts (33%) did neither explicitly document nor implicitly hint at any tool support. The frequency of supporting tools employed in the development of the remaining 56 DSML designs (67%) is shown in Table 1. In total, we extracted 180 occurrences in which tools supported the implementation of a DSML. 53 distinct tools were used, of which 33 (62%) are distributed under an open-source and 13 (25%) under a closed-source license. No license information was available for seven out of the 53 tools (13%).

As for the most frequently adopted tools, Table 1 shows that No Magic's MagicDraw ranks first with 27 occurrences in supporting the development of a DSML for any of the eight decision options. Thus, MagicDraw was applied in 15% of the cases in which a tool supported the development of a DSML. Ranked second, Table 1 lists IBM's Rational Software Architect (18 occurrences; 10%), followed by the open source Eclipse Modeling Framework (EMF) at the third position (17 occurrences, 9%).

We collected the license information under which each tool supporting the development of a DSML is distributed (see Table 2). In total, we extracted 181 license details (19 distinct licenses) for the 53 tools found.[2] Of these 19 distinct licenses, 10 are open-source (in different versions) and 9 closed-source licenses. Most of the tools found are distributed under the open-source Eclipse Public License (EPL) in version 1.0 (73 occurrences; 40%). The next three frequently used licenses are proprietary (i.e. closed source) end-user license agreements

2. The difference of 181 license information found for 180 tool occurrences results from the double licensing policy of the Saxon tool: The open-source versions are licensed under the Mozilla Public License (MPL) 2.0, the commercial versions under Saxonica's EULAs.

TABLE 1. Frequency of supporting tools employed in the development of the 84 DSML designs. An asterisk (*) denotes a tool distributed under an open-source license.

Frequency	Tool
27 (15%)	No Magic MagicDraw
18 (10%)	IBM Rational Software Architect
17 (9%)	Eclipse Modeling Framework (EMF)*
15 (8%)	UML2 project of the Eclipse Model Development Tools (MDT)*
12 (7%)	Sparx Systems Enterprise Architect
6 (3%)	IBM Rational Rose
5 (3%)	Eclipse Atlas Transformation Language (ATL)*, Gentleware Poseidon for UML, OCL project of the Eclipse Model Development Tools (MDT)*, TOPCASED*
4 (2%)	openArchitectureWare*
3 (2%)	Eclipse Graphical Editing Framework (GEF)*
2 (1%)	Apache Velocity*, Eclipse EMF Compare*, Eclipse Graphical Modeling Framework (GMF)*, Eclipse Papyrus*, Eclipse VIATRA2*, IBM Rational Rhapsody, IBM Rational Rhapsody Architect for Systems Engineers, IBM Rational Software Modeler, LPGM4EA, MDD4SOA*, MOFScript*, Objecteering, SiTra, SmartQVT*, StarUML*, Telelogic Tau (G2), WebRatio
1 (1%)	AGG*, CompSize, Eclipse Kernel MetaMetaModel (KM3)*, Eclipse Modeling Workflow Engine (MWE)*, Eclipse Process Framework (EPF)*, Eclipse Xpand*, EIS plug-in, Fujaba Tool Suite*, IBM Rose Extensibility Interface (REI), ITEM ToolKit, Jasmine-AOI, Kent Modeling Framework (KMF)*, Kermeta*, LTSA, mediniQVT*, MOCAS*, Oclarity, Octopus*, Saxon*, UCEd*, UML2 Tools project of the Eclipse Model Development Tools (MDT)*, UML Model Transformation Tool (UMT)*, VIENNA Add-In*, Xalan-Java*

TABLE 2. Frequency of licenses under which supporting tools are distributed. An asterisk (*) denotes an open-source license. EULA = end-user license agreement.

Frequency	License
73 (40%)	Eclipse Public License (EPL) 1.0*
31 (17%)	IBM EULA
27 (15%)	No Magic EULA
12 (7%)	Sparx Systems EULA
9 (5%)	No license information available
5 (3%)	Gentleware EULA
4 (2%)	Apache License 2.0*
3 (2%)	Common Public License (CPL) 1.0*
2 (1%)	GNU General Public License (GPL) 2.0*, modified GNU General Public License (GPL) 3.0*, Objecteering EULA, Telelogic EULA, WebRatio EULA
1 (1%)	Berkeley Software Distribution (BSD)*, GNU General Public License (GPL) 3.0*, GNU Lesser General Public License (LGPL) 2.0*, GNU Lesser General Public License (LGPL) 2.1*, ITEM Software EULA, Mozilla Public License (MPL) 2.0*, Saxonica EULA

(EULA) of the companies IBM (31 occurrences; 17%), No Magic (27 occurrences; 15%), and Sparx Systems (12 occurrences; 7%). No license information could be retrieved in nine out of the 181 cases (5%).

In Table 3, we differentiate between open- and closed-source license models per tooling category. We classified the license information found for the support tools of the 84 DSML designs accordingly. In total, we can report that open-source tools are used in 49% and closed-source tools in 46% of the collected cases to support the development of DSMLs. For the decision point of choosing a language-model editor (TO.1), a balanced distribution between the application of open- and closed-source licensed tools exists (51% and 49%). We found that more closed-source tools are employed (76%) as generators for the diagrammatic-syntax editors (TO.3). For all other decision points, open-source tools are more often or even exclusively used (constraints evaluators, textual-syntax editor generators, model-execution engines, M2M/M2T transformation engines, orchestration engines; TO.2, TO.4–TO.8; see Table 3).

3.3. Tooling Support at Design-Decision Points

While conducting our SLR study ([12], [13]), we evaluated the documentation quality of the UML-based DSML designs retrieved. We found a number of issues including missing metamodel and/or UML profile definitions, defects in metamodel and/or profile definitions as well as in constraint-language expressions, or missing mappings between DSML language models and UML profiles.

Given that broadly accepted or adopted documentation guidelines for DSML designs do not exist, the choice for a dedicated tool chain becomes relevant. Documentation quality and discipline often directly depend on a tool's implementation. For example, whether a language-model editor implements all syntax and semantics requirements from the UML specification [4] and visually renders the icons as well as enforces semantics constraints accurately while modeling or not, determines the well-formedness and completeness of a DSML's design documentation. Thus, the availability of support tools is relevant, on the one hand, for practical DSML development (modeling, automation) and, on the other hand, for improving the documentation quality of DSML designs.

In our study, we have evaluated the availability of tooling support with respect to the six design-decision points identified relevant for DSML development (see Section 2.2). Fig. 3 shows whether DSML design choices made in the 84 retrieved documents of our SLR study are actually covered by dedicated tooling details. Therefore, we relate the eight tooling categories (see Section 3.1) to their corresponding options of the remaining six decision points (e.g., whether a UML profile definition is covered by a language-model editor; TO.1).[3]

Table 4 indicates that a language-model formalization was found in all 84 DSML reports (e.g., UML profile definition). 49 (58%) of the publications formalize the DSML language model via language-model editors, 35 (42%) do not explicitly document nor implicitly hint at any tool support (TO.1). Language-model constraint expressions are specified and evaluated via a dedicated engine (TO.2) in eight out

3. Please note that we only evaluate whether *MDD-specific* tooling support is documented for the six decision points. We do not report, if, for example, a DSML's behavior specification is defined informally via textual descriptions which can be written with any common word processing software.

TABLE 3. Frequency of license models under which supporting tools are distributed per tooling category.

	TO.1	TO.2	TO.3	TO.4	TO.5	TO.6	TO.7	TO.8	Total
Open source	37 (51%)	10 (45%)	11 (24%)	2 (100%)	1 (100%)	14 (78%)	12 (71%)	2 (100%)	89 (49%)
Closed source	36 (49%)	8 (36%)	35 (76%)	0 (0%)	0 (0%)	1 (6%)	3 (18%)	0 (0%)	83 (46%)
Unknown	0 (0%)	4 (18%)	0 (0%)	0 (0%)	0 (0%)	3 (17%)	2 (12%)	0 (0%)	9 (5%)
Total	73 (100%)	22 (100%)	46 (100%)	2 (100%)	1 (100%)	18 (100%)	17 (100%)	2 (100%)	181 (100%)

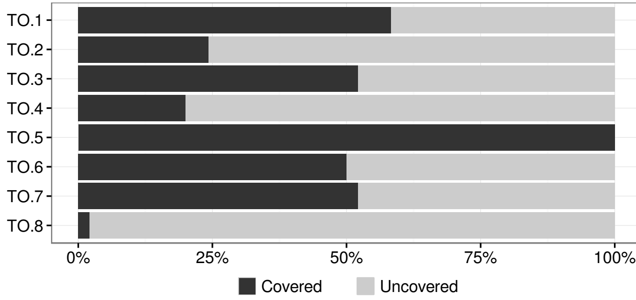

Fig. 3. Overview of tooling details covered in the DSML design reports.

TABLE 4. Frequency of tool support per tooling category.

Tool	Design-decision point	Covered	Uncovered	Total
TO.1	Language-model formalization	49 (58%)	35 (42%)	84
TO.2	Language-model constraints	8 (24%)	25 (76%)	33
TO.3	Concrete-syntax definition	37 (52%)	34 (48%)	71
TO.4	Concrete-syntax definition	1 (20%)	4 (80%)	5
TO.5	Behavior specification, platform integration	1 (100%)	0 (0%)	1
TO.6	Language-model constraints, platform integration	5 (50%)	5 (50%)	10
TO.7	Language-model constraints, platform integration	12 (52%)	11 (48%)	23
TO.8	Language-model constraints, behavior specification, platform integration	1 (2%)	46 (98%)	47
Total		114 (42%)	160 (58%)	274

of 33 cases (24%). Editor generators are used in 52% for diagrammatic syntax (TO.3) and in 20% for textual syntax (TO.4) definitions (only one source found). A model execution engine (TO.5) can support the phase of behavior specification and platform integration (e.g., constraining/direct model execution). The only tool-supported case found implements an engine for the execution of UML state machines. M2M and M2T transformations are employed for constraining language models as well as for integrating the DSML to a specific platform (e.g., via M2T generator templates). Automatic M2M transformations (TO.6) are covered in five out of ten DSML designs (50%), M2T transformation engines (TO.7) in twelve out of 23 (52%) DSML designs. Orchestration engines (TO.8) coordinate the automatic execution of a process, thus, they are employed, for example, to coordinate the evaluation of language-model constraints, the direct execution of models (behavior specification), and the transformation from one model representation to another (platform integration). For these activities, orchestration engines were found employed in one out of 47 DSML reports (2%). To summarize, with regard to the six design-decision points identified relevant for DSML development, in only 42% of the cases, a design choice was backed up by corresponding tooling support.

3.4. Study Limitations

Based on our SLR, we applied 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 (e.g., publicly accessible software artifacts). SLR studies have the major problem of finding a representative set of relevant primary studies. In general, we closely followed established guidelines on designing and conducting SLRs available from research on evidence-based software engineering to avoid any pitfalls ([22], [23], [26]). However, we may risk having missed further relevant primary studies on UML-based DSMLs. Note that we addressed this threat right from the beginning, by building our review procedure around the principle of continuous search validation and search refinement driven by a QGS as a recommended practice [22].

We only evaluated tools publicly available to date (e.g., via a URL). When no URL was provided or the resource had moved, we used a web search to find the implementation based on the title of the paper as well as on the name of the tool and of the author(s). When locating the tool failed this way, it was excluded from our study. Furthermore, we took only the latest version of a software tool into account (i.e. we did not scan code repositories for the version number at the time of publication of the paper). As version numbers and license models might also change over time, we risk having collected license information different from the one at the time of publication. For example, there might be the case that a tool is based on a software framework formerly licensed under the Common Public License (CPL) 1.0 which is now distributed under its successor Eclipse Public License (EPL) 1.0. In this case, we would have extracted the EPL 1.0 licensing information only.

Our classification of tooling support introduced in Section 3.1 focuses exclusively on MDD-based tools which sup-

port a designer in the development of DSMLs. Implementation characteristics of tools, such as, editors or engines as well as of generated platform-specific artifacts (e.g., source code) are beyond the scope of this paper. Therefore, we neither report nor differentiate between, for example, different language-model implementation languages used by language-model editors (e.g., whether Ecore or XML is used), different programming languages to implement supporting tools (e.g., whether a constraints evaluator is implemented in Java or Python), or different generated platform-specific artifacts (e.g., whether a M2T transformation produces source code in C++ or Ruby).

4. Discussion

In this section, we discuss implications of the results of our survey regarding development-tooling trends for DSMLs, integrated tool chains, license models, and tool-support quality.

Tool-support trends: As can be seen in Table 1, proprietary tools are common for the development of DSMLs. Especially prominent are proprietary tool-supported language-model and diagrammatic-syntax editors (TO.1, TO.3; see Table 3). These figures confirm an observation made during data extraction: DSML developers have a tendency to define UML profiles and implement the profile definition in UML models using proprietary tools. However, for further model-processing tasks (e.g., model transformations), they switch to open-source frameworks (e.g., Eclipse).

The following reasons may (partially) explain this pattern. Virtually all UML modeling tools (no matter whether open or closed source) support the export/import of XMI serialized models [27]. Commercial tools are developed on top of and integrated with the Eclipse IDE (e.g., IBM Rational Software Architect). Therefore, model-level interoperability makes switching between different tools all based on the same core infrastructure a feasible option. Furthermore, commercial tools may tend to provide more mature functionality than open-source tools. A study conducted in 2010 [28], examining the maturity of Eclipse-based MDD tools, slightly underlines this statement. Therein, Eclipse Papyrus and Eclipse GMF—two tools used for defining, on the one hand, UML-based language models (i.e. profiles) and, on the other hand, diagrammatic concrete syntaxes—are rated emerging.

In contrast, the same survey rates the core MDD-based components of the Eclipse platform as stable. Nevertheless, a five year old study may not reflect current Eclipse-based MDD implementations best. Another reason may be that the development of research prototypes (e.g., constraints evaluation, model execution, and model transformation engines) tend to be provided as open-source implementations. These prototypes demand a language model which may be specified using commercial tools and which is serialized via XMI for further processing. Last, proprietary tools—while possibly being more mature—may not provide the same amount of functionality than, for example, the integrated tool chain of all MDD-based projects of the Eclipse framework. This is also indicated by the lack of closed-source license tools for decision points TO.4–TO.8 in Table 3.

Open-source tools for DSML development seem to provide a more diverse range of functionality employed by the 84 DSML designs (see Table 3). For every tool-supported decision point (TO.1–TO.8), at least one open-source tool was utilized. The Eclipse Modeling Framework (EMF) represents the core for many of these MDD-based open-source projects (e.g., Eclipse UML2/MDT, GEF, GMF etc.; see Table 1). The majority of Eclipse-based developments is also reflected by the high frequency of EPL 1.0 licensed software tools (see Table 2).

Integrated tool chains: By analyzing the extracted tooling information from our SLR study, we identified a predominance of Eclipse-based tools. Eclipse is used as a core infrastructure for both, commercial as well as open-source software solutions. Interoperability between different Eclipse-based software tools is facilitated through utilizing standardized interfaces (e.g., XMI export/import, EMF-based model representation). The Eclipse foundation provides the infrastructure for its user and developer communities (e.g., Eclipse community forum, code repository, issue tracking system). Eclipse-based tools can be combined in a mix-and-match strategy to build an integrated tool chain. At each tooling decision point, a DSML designer can choose from a variety of different implementations. For example, a choice for an Eclipse-based M2T transformation engine can be made from the following three template-based code generators: Acceleo, EGL, and Xpand.

License models: Most commercial supporting tools extracted from the data retrieved via our SLR study (e.g., IBM Rational Rose, Gentleware Poseidon for UML, No Magic MagicDraw) distribute their products with two different licensing options: a single-user (seat, individual/authorized/single/named user) and a multi-user (floating, site, concurrent user) license model (see, e.g., [29]). Further commercial license models include evaluation/trial and redistribution licenses (e.g., Saxon), rental licenses (e.g., WebRatio), combinations of open- and closed-source (commercial) licenses (e.g., Saxon), and combinations of closed-source (commercial and free of charge) licenses (e.g., Gentleware Poseidon for UML). Therefore, the application of the tool is limited to the license model purchased with the software.

In contrast, open-source licenses allow—under different terms and conditions—that the source code of the tool be used (e.g., in conjunction with an enterprise system), modified (e.g., tailored to be integrated with an enterprise system), and distributed (e.g., via a newly created software product). Prominent example of reusing code parts which are open-source licensed (EPL 1.0) for commercial products is the Eclipse-based IBM Rational Software Architect.

The 53 different tools retrieved via our SLR are distributed under 19 different licenses (10 open source, 9 closed source). The license model of a software tool has implications on the tool's utilization, for example, with respect to the integration in existing tool chains (e.g., compatibility of different licenses,

multi-licensing) or the distribution policy of a newly created software product (e.g., whether commercial distribution is permitted).

Tool-support quality: Our study revealed that from a third of the 84 DSML reports (28) tooling information was completely missing (i.e. no hints given in the paper). In some publications, the tooling information was outdated (e.g., a URL was provided, but the software artifact had moved) and we resorted to manually searching for the implementation. In total, we could only successfully access 15 references (i.e. URLs) to software prototypes available to date (18%). Furthermore, in nine cases, tool support was mentioned, but no license information was provided (see Tables 2 and 3). These figures are also backed up by frequency counts of the tool support per DSML design-decision point (see Table 4). Only 42% of the choices for a design decision were supported by corresponding tools (according to the data extracted from our paper corpus).

Regarding the quality of a tool's implementation (e.g., with respect to its error-free functionality), open-source and commercial products differ in their degree of support provided (e.g., community-based vs. purchased software maintenance plan, bug-fixing intervals). Furthermore, customer-related training and help services as well as availability of accompanying documents (e.g., installation instructions, user manuals, software documentations) may not always be available for open-source tools—especially for projects which are not frequently maintained and used by a rather small number of people (see also [9]).

5. Related Work

For this paper, we identified two categories of related work: (1) work on DSL design procedures and DSL design-decision making and (2) reports on empirical MDD research.

Systematic development of DSLs: A group of related work reports observations from developing DSLs in (industrial) practice. For example, [30] conducted a study including 23 industrial projects for the definition of DSLs. Similar to our approach, a number of DSLs are systematically compared. However, in contrast to our paper, the authors of [30] provide a high-level description only and do not describe specific DSL design decisions or decision options in detail. In addition, [31] reports on twelve lessons learned from three DSL experiments. Despite that these lessons learned being described at a comparatively high level of abstraction, they can, in general, also be observed in our work.

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). From this perspective, some authors propose systematic approaches that define domain-specific UML extensions via UML profiles (see, e.g., [32], [33]). While all related, none touches on tool-related design decisions for UML-based DSMLs explicitly.

In addition, knowledge on DSL design decisions can also be gained from analyzing toolkits for DSL development. For example, the authors of [34] present a tool for the definition and usage of integrated DSMLs. Similarly, [35] 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 tools for UML-based DSL development.

Empirical evidence on MDD-based tool usage: Pérez-Medina et al. [15] present a survey of existing MDD tools in support of model management in the HCI domain. The paper's goal is to find criteria that could help HCI designers to select a MDD tool for a given task. 14 tools are evaluated regarding their capabilities in metamodeling, modeling, model transformations and weaving, repository management, as well as tool interoperation. As for these criteria, our work provides complementary quantitative evidence. However, the report in [15] is specific to the HCI domain and does not result from a systematic but rather an ad-hoc procedure for defining criteria and for selecting tools for comparison.

Schaefer [16] reviews several MDD-based transformation tools/languages (e.g., ATL, XSLT) and discusses their applicability for model-based user-interface development. Criteria for the comparison of transformation tools include the support for model/XML/code transformations or extension/parameterization possibilities. Again, the report is based on an ad-hoc selection of tools and on an ad-hoc definition of evaluation criteria. Another limitation is that Schaefer does not consider MDD tooling other than transformation engines.

Whittle et al. [9] discuss the impact of tools on MDD adoption based on two sets of 39 qualitative interviews conducted in different companies. The paper proposes a taxonomy of tool-related considerations which can be used to reason about future MDD-based tooling decisions. The taxonomy encompass technical (e.g., tool features, complexity), internal/external organizational (e.g., processes, external influences), and social factors (e.g., control, trust). The tool-feature subcategories have some similarities with our tooling decision options (e.g., UML profiles, code generation templates) and are a source to complement the choices of our decision point. However, the focus of [9] was clearly different, that is, to develop and to validate the proposed taxonomy via two sets of interviews.

Giese and Henkler [17] identify a number of general requirements for the model-driven specification of software-intensive systems. Their work is limited to graphical MDD of adaptable software-intensive systems with emphasis on blending software engineering with concepts taken from control and safety engineering. In the survey, the authors identify requirements (support for modeling, MDD, and model analysis) and use them to classify and to characterize nine approaches. While the evaluation criteria (i.e. requirements) have some overlap with our tooling options, their selection procedure remains ad-hoc and reflects only a narrow domain. Furthermore, neither tool-usage frequencies nor details of their usage in MDD are reported.

6. Conclusion and Future Work

In this paper, we presented an extension to a SLR study to include an additional decision point of tooling support for UML-based DSML development. In particular, we extracted tooling decisions from 84 DSML designs retrieved via the SLR. Based on this data, we created a classification for MDD-based support tools for DSML development. Using this classification, we compiled a comprehensive list of commonly used open- and closed-source tools as well as of corresponding licenses. Results of our study show that closed-source tools are in the majority of cases employed solely as editors for the language model and the diagrammatic syntax of a DSML. Open-source tools are utilized for different, special-purpose DSML development activities (e.g., for constraint evaluation, model transformation).

These empirical findings complement recent, qualitative research on MDD tool adoption [9]. The collected and aggregated data also provides a source of documented design rationale for DSML developers. As part of our decision-record catalog [14], developers can review lists of employed tools including license models and documented interdependencies between tools and other design decisions (e.g. concrete syntax) when making their project-specific design decisions.

In future work, we will repeat data extraction to be fully compliant with the quality-assessment standard of our SLR (multiple raters [12]). Furthermore, we will contact DSML authors to fill in tooling information missing from their publications. These results will be reported in a revision of our catalog [14].

References

[1] G. Neumann, S. Sobernig, and M. Aram, “Evolutionary business information systems: Perspectives and challenges of an emerging class of information systems,” *Bus. & Inform. Syst. Eng.*, vol. 6, no. 1, pp. 33–38, Feb. 2014.

[2] D. Spinellis, “Notable design patterns for domain-specific languages,” *J. Syst. Softw.*, vol. 56, no. 1, pp. 91–99, 2001.

[3] S. Kelly and J. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[4] Object Management Group, “OMG unified modeling language (OMG UML),” Available at: http://www.omg.org/spec/UML, Jun. 2015, version 2.5, formal/2015-03-01.

[5] ——, “OMG meta object facility (MOF) core specification,” Available at: http://www.omg.org/spec/MOF, Jun. 2015, version 2.5, formal/2015-06-05.

[6] L. Nascimento, D. L. Viana, P. A. M. S. Neto, D. A. O. Martins, V. C. Garcia, and S. R. L. Meira, “A systematic mapping study on domain-specific languages,” in *Proc. 7th Int. Conf. Softw. Eng. Adv.* IARIA, 2012, pp. 179–187.

[7] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, “Empirical assessment of MDE in industry,” in *Proc. 33rd Int. Conf. Softw. Eng.* ACM, 2011, pp. 471–480.

[8] P. Mohagheghi and V. Dehlen, “Where is the proof? – A review of experiences from applying MDE in industry,” in *Model Driven Archit. – Found. Appl.*, ser. LNCS. Springer, 2008, vol. 5095, pp. 432–443.

[9] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, “A taxonomy of tool-related issues affecting the adoption of model-driven engineering,” *Softw. Syst. Model.*, 2015.

[10] M. Strembeck and U. Zdun, “An approach for the systematic development of domain-specific languages,” *Softw. Pract. Exper.*, vol. 39, no. 15, pp. 1253–1292, 2009.

[11] A. Jansen and J. Bosch, “Software architecture as a set of architectural design decisions,” in *Proc. 5th Working IEEE/IFIP Conf. Softw. Arch.* IEEE, 2005, pp. 109–120.

[12] S. Sobernig, B. Hoisl, and M. Strembeck, “Extracting reusable design decisions in UML-based domain-specific languages: A multi-method study,” submitted.

[13] ——, “Protocol for a systematic literature review on design decisions for UML-based DSMLs,” Available at: http://epub.wu.ac.at/4467/, WU Vienna, Tech. Rep. 2014/02, Feb. 2015.

[14] B. Hoisl, S. Sobernig, and M. Strembeck, “A catalog of reusable design decisions for developing UML/MOF-based domain-specific modeling languages,” Available at: http://epub.wu.ac.at/4466/, WU Vienna, Tech. Rep. 2014/03, Feb. 2015.

[15] J.-L. Pérez-Medina, S. Dupuy-Chessa, and A. Front, “A survey of model driven engineering tools for user interface design,” in *Task Models Diagr. User Interf. Des.*, ser. LNCS. Springer, 2007, vol. 4849, pp. 84–97.

[16] R. Schaefer, “A survey on transformation tools for model based user interface development,” in *Human-Comput. Interact. Interact. Des. Usability*, ser. LNCS. Springer, 2007, vol. 4550, pp. 1178–1187.

[17] H. Giese and S. Henkler, “A survey of approaches for the visual model-driven development of next generation software-intensive systems,” *J. Vis. Lang. Comput.*, vol. 17, no. 6, pp. 528–550, Dec. 2006.

[18] T. Mens and P. v. Gorp, “A taxonomy of model transformation,” *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, 2006.

[19] A. Khalaoui, A. Abran, and E. Lefebvre, “DSML success factors and their assessment criteria,” *Metrics News*, vol. 13, no. 1, pp. 43–51, 2008.

[20] S. Kelly and R. Pohjonen, “Worst practices for domain-specific modeling,” *IEEE Softw.*, vol. 26, no. 4, pp. 22–29, 2009.

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

[22] H. Zhang, M. A. Babar, and P. Tell, “Identifying relevant studies in software engineering,” *Inform. Softw. Tech.*, vol. 53, no. 6, pp. 625–637, Jun. 2011.

[23] S. Jalali and C. Wohlin, “Systematic literature studies: Database searches vs. backward snowballing,” in *Proc. ACM-IEEE Int. Sym. Empir. Softw. Eng. Meas.* ACM, 2012, pp. 29–38.

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

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

[26] B. Kitchenham, “Procedures for performing systematic reviews,” Keele University & National ICT Ltd., Joint Tech. Rep. (Keele University Tech. Rep., NICTA Tech. Rep.) TR/SE-0401, 0400011T.1, Jul. 2004.

[27] Object Management Group, “XML metadata interchange (XMI) specification,” Available at: http://www.omg.org/spec/XMI, Jun. 2015, version 2.5.1, formal/2015-06-07.

[28] K. Hussey, B. Selic, and T. McClean, “An extended survey of open source model-based engineering tools,” Zeligsoft Inc., Tech. Rep. Revision E, May 2010.

[29] M. L. Rustad, *Software Licensing: Principles and Practical Strategies*. LexisNexis, 2014.

[30] J. Luoma, S. Kelly, and J. Tolvanen, “Defining domain-specific modeling languages: Collected experiences,” in *Proc. 4th OOPSLA Worksh. Domain-Specific Model.*, Oct. 2004.

[31] D. Wile, “Lessons learned from real DSL experiments,” *Sci. Comput. Program.*, vol. 51, no. 3, pp. 265–290, 2004.

[32] S. Robert, S. Gérard, F. Terrier, and F. Lagarde, “A lightweight approach for domain-specific modeling languages design,” in *Proc. 35th Euromicro Conf. Softw. Eng. Adv. Applicat.* IEEE, 2009, pp. 155–161.

[33] 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, 2007, pp. 2–9.

[34] J.-P. Tolvanen and S. Kelly, “MetaEdit+: Defining and using integrated domain-specific modeling languages,” in *Proc. 24th ACM SIGPLAN Conf. Object-Oriented Program. Syst. Lang. Applicat.* ACM, 2009, pp. 819–820.

[35] U. Zdun, “A DSL toolkit for deferring architectural decisions in DSL-based software design,” *Inform. Softw. Tech.*, vol. 52, no. 9, pp. 733–748, 2010.