Merlin-A: A tool to engineer adaptive modelling languages

Esther Guerra
Universidad Autónoma de Madrid
Madrid, Spain
Esther.Guerra@uam.es

Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain
Juan.deLara@uam.es

Abstract

Modelling languages are pervasive in software engineering to describe systems and processes. They are pivotal in model-driven engineering and low-code development, where automation solutions based on domain-specific languages (DSLs) need to be built for particular domains. However, today's DSLs are rigid, since a single language needs to accommodate a wide variety of users with diverse backgrounds, modelling purposes, and usage scenarios.

The new notion of *adaptive modelling language* attacks this limitation by enabling the adaptation of a language to cater for the user expertise and purpose of use. In this tool demo paper, we present the workbench Merlin-a for adaptive languages. It supports the creation of families of modelling language variants, migration transformations between the variants, language adaptation triggers, and recommenders of language variants. The tool website is http://miso.es/tools/merlin-adaptive/, and a demonstration video is available at https://youtu.be/fy171c7Afzk.

CCS Concepts

• **Software and its engineering → Domain specific languages**.

Keywords

Domain-Specific Languages, Product Lines, Adaptive Modelling Languages, Model-driven Engineering, Lowcode, IDEs

ACM Reference Format:
Esther Guerra and Juan de Lara. 2025. Merlin-A: A tool to engineer adaptive modelling languages. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728585

1 Introduction

Modelling languages are essential in software engineering, especially in paradigms like model-driven engineering (MDE) [3], where models are used to conduct most phases of the development. Modelling languages can be general-purpose, like the UML, or domain-specific, tailored to an application area. Recently, lowcode development [14, 26] proposes the combination of domain-specific languages (DSLs) and graphical forms to describe applications without the need to program. Lowcode approaches often run on cloud platforms where users – so called *citizen developers* – can define and deploy applications with zero-cost installation.

Both general-purpose and domain-specific modelling languages may have users with diverse background and expertise (especially in lowcode platforms), different roles and interests (e.g., clients, engineers, marketing), a variety of purposes (e.g., informal sketching vs formal analysis) and access means (e.g., tablets vs computers with large monitors). However, modelling environments usually enforce *visual monolinguism* [20], offering the same language version for every context of use.

Adaptive modelling languages [6] aim to overcome this limitation and facilitate modelling by considering the range of contexts where a language is used. Instead of being fixed, they accommodate a family of language variants, along with conditions triggering language reconfigurations, and mechanisms for adapting the model under construction to the language variant in use. This is useful in many scenarios. First, adaptive languages can adjust to the user background, which is especially interesting in educational settings. For instance, novice learners can use simple versions of class diagrams, and proficient ones more sophisticated versions (e.g., with aggregation or n-ary associations). Adaptive languages may also adapt to the modelling process and purpose. For example, early project phases benefit from informal modelling for discussion, but later phases may require precise models for analysis or code generation [10]. Thus, there may be a flexible variant of class diagrams for analysis, stricter variants for design, and other variants that consider the particularities of specific programming languages (e.g., C++ has multiple inheritance, but Java does not).

Overall, adaptive languages comprise: (i) a set of language variants, (ii) triggers for language reconfiguration, and (iii) mechanisms for adapting models from one language variant to another. Each of these elements poses challenges, as: (i) specifying each language variant individually is impractical; (ii) means are needed to identify when to reconfigure a language (e.g., detect when the user resorts to workarounds that a more expressive language variant makes unnecessary) and to help choose the most appropriate variant at any time; and (iii) defining a migration transformation for every pair of language variants is unfeasible.

This paper showcases Merlin-a, an Eclipse plugin for the creation of adaptive modelling languages and their editors, which resolves the previous challenges. It relies on MDE and product lines [21] for the definition of language variants. Language reconfigurations can be automated by means of monitoring hooks in the IDE and language variant recommenders. Moreover, it enables the compact, feature-wise specification of migration transformations between language variants by means of *adapters*, and the automated composition of migration steps depending on the source and target language variants [6].

FSE Companion '25, June 23–28, 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728585

2 Merlin-a

Merlin-a is a language workbench to engineer adaptive modelling languages. Next, we outline its architecture (Sec. 2.1), and explain how to create (Sec. 2.2) and use (Sec. 2.3) adaptive languages.

2.1 Architecture

Merlin-a is an Eclipse plugin. Fig. 1 shows its architecture, where the components are labelled with the sections that explain them.

The upper part of the figure contains the components enabling the definition of adaptive languages by language designers. Merlin-a uses the Eclipse Modeling Framework (EMF) [27] to define and manipulate models and meta-models; FeatureIDE [19] to specify the language variability space and the language configurations of interest; an Xtext [30] editor to define the adapters between language variants; and Henshin [2] to build the transformations to migrate models between language variants. Merlin-a is built atop Merlin [11], which provides means to validate the correctness of a language family. Moreover, a dedicated adapter analyser reports metrics and potential errors in the defined adapters. Sec. 2.2 introduces these elements on the basis of an example.

From an adaptive language definition, Merlin-a generates an editor where users can build models using the desired language variant (cf. bottom of Fig. 1). Changing the language variant triggers an automatic model migration to the new variant. The editor integrates language variant recommenders and can be extended with custom adaptation triggers. Sec. 2.3 details the editor capabilities.

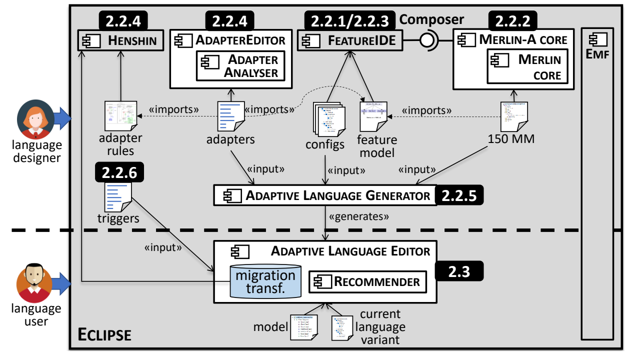

Figure 1: Architecture of Merlin-a.

2.2 Defining an adaptive language

Next, we show how to create an adaptive language with Merlin-a. As a running example, we define an adaptive class diagram language with variants for four phases of a software project: analysis, design, and detailed design for Java and C++. The analysis phase employs a simple language variant without methods, compositions or aggregations; the design phase uses a variant with these elements; the detailed design phase for Java uses single inheritance and interfaces; and the variant for C++ uses multiple inheritance and interfaces. Another scenario for adaptive class diagrams is education, with a beginner's version having only basic concepts, and progressively richer versions as users gain expertise.

Fig. 2 shows Merlin-a in action. The package explorer (label 1) depicts the workspace, where a project named AdaptiveClassDiagrams has been created. The next steps define the adaptive language.

2.2.1 **Language variability.** First, the language designer must define the language variability space, that is, the features that the language variants may have, and how they can be combined. This is specified as a feature model [18] using FeatureIDE.

Label 2 in Fig. 2 shows the feature model for the running example. It states that any variant of class diagrams must support classes and associations. However, it offers variability with respect to whether classes can define methods or not; the type of inheritance (single, multiple or none); whether interfaces are supported; the style of associations (unidirectional references or full associations); and the available decorations for association ends (composition, aggregation, navigation and cardinality).

Having an explicit feature model simplifies the process of selecting the features of a language variant. A valid feature selection is called a *feature configuration*. For instance, the example feature model admits a class diagram variant with just classes and unidirectional references and no decorations, but disallows having at the same time unidirectional references and full associations.

2.2.2 **Language family.** Next, the designer must define the abstract syntax of the adaptive language by means of a *150% meta-model* (150MM) [11]. This is a regular meta-model that declares all elements (classes, references, attributes, inheritance relationships, OCL constraints) that can be present in any of the language variants, superimposed [5]. The elements of the 150MM may declare *presence conditions* (PCs), which are boolean formulae that can use features of the feature model as variables. This way, the PCs determine if an element is present or not in a language variant, depending on the features the variant selects.

More in detail, to obtain a language variant definition from the 150MM, a feature configuration must be provided. Then, the features selected by the configuration are replaced by *true* in the PCs, and the unselected ones are replaced by $false$. Finally, the elements whose PC evaluates to $false$ after this replacement are deleted from the 150MM, and the remaining elements conform the meta-model of the language variant induced by the feature configuration. Merlin-a performs this process automatically and transparently.

The 150MM can be built with any Ecore editor, but we recommend OCLinEcore [23] as it simplifies editing annotations and OCL constraints. Label 3 in Fig. 2 shows an excerpt of the 150MM for adaptive class diagrams. Line 4 refers to the feature model in label 2, and allows Merlin-a to type check the PCs for incorrect feature names. Lines 5–15 show part of the definition of the meta-class Class. This meta-class and its attributes name and isAbstract have no PC, which entails that they will be present in every language variant regardless of the features the variant has. Then, properties parents and parent define PCs using the statement "annotation presence (condition='...')". The multi-valued property parents is used to represent multiple inheritance and will appear in language variants that select the feature Multi. Instead, the mono-valued property parent represents single inheritance and has Single as PC.

Merlin-a supports validating the correctness of the 150MM, both syntactically (e.g., no language variant has inheritance cycles, the features in PCs exist in the feature model) and semantically (e.g., all PCs are satisfiable, the OCL constraints in all variants are satisfiable). All semantic validations rely on SAT solving [11].

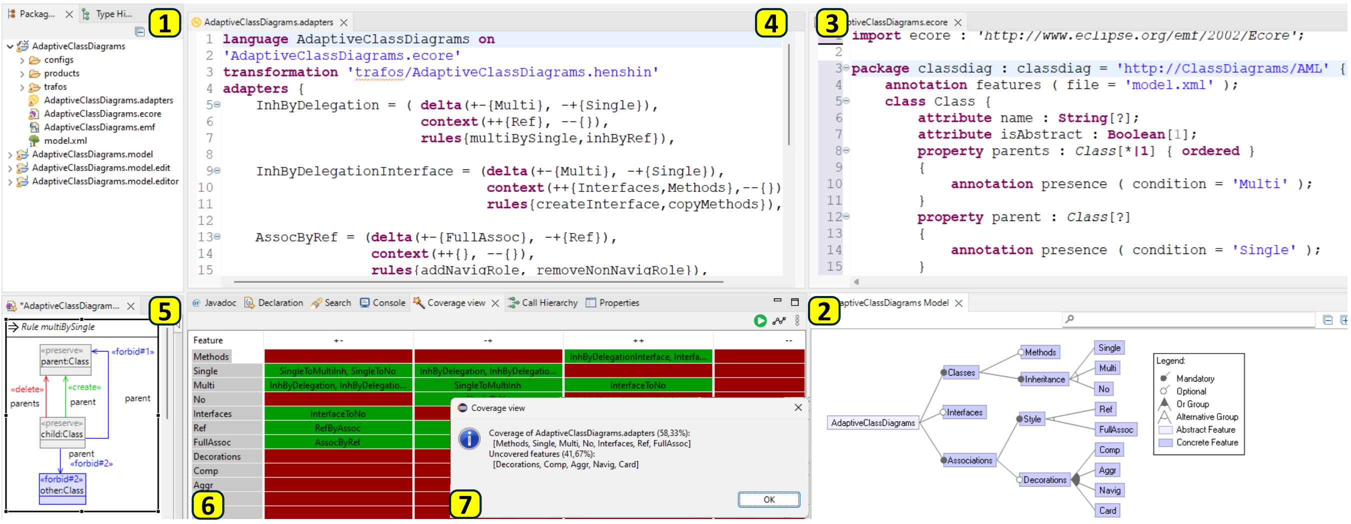

Figure 2: Merlin-A in action: (1) Workspace, (2) Feature model, (3) 150MM, (4) Adapters definiton, (5) Migration rules, (6) Feature coverage by migrations, (7) Feature coverage analysis.

2.2.3 **Configurations of interest.** By default, an adaptive language has as many variants as valid feature configurations (288 in our example). The designer may restrict this set of configurations (variants) to consider only those relevant or meaningful for the language. In the example, the language designer is only interested in four configurations: class diagrams for analysis, design, and detailed design for Java or C++. These configurations are defined using FeatureIDE, and stored as configuration files in folder configs.

2.2.4 **Migration adapters.** The last ingredient is the specification of how to migrate models between language configurations. When the user is building a model with a language variant and switches to another variant, this specification is used to adapt the model to the new variant.

However, defining a migration transformation for every two language variants is impractical. While our example with four variants would require specifying 12 transformations, covering all 288 configurations of the feature model requires 82.656 transformations, which is unfeasible. Moreover, many of them share rules.

To reduce this burden, Merlin-A permits the compact definition of migration transformation by small modules called *adapters* [6]. An adapter specifies how to adapt a model when a language feature (or a small set of them) becomes available or unavailable upon a language reconfiguration. Its definition comprises a set of feature differences (feature changes or invariants), plus graph transformation rules [8] to modify the models to accommodate those differences. This way, each adapter bridges the gap between a reduced set of language features. When moving from a source to a target language configuration, Merlin-A identifies their feature differences, and automatically builds a suitable migration transformation by composing adapters compatible with those differences [6].

Adapters are defined using a textual DSL, and transformation rules by means of Henshin [2]. Label 4 in Fig. 2 shows the definition of some adapters. The first three lines declare the adaptive language name, the file with the 150MM, and the file with the Henshin transformation rules. The subsequent lines define the adapters. Each one has a name, *delta* features that become available or unavailable, *context* features that remain available or unavailable, and names of rules to adapt the models when a language reconfiguration fulfils the delta and context conditions. Lines 5–7 define the adapter InhByDelegation, applicable when feature Multi was available but becomes unavailable (+-), Single was unavailable but becomes available (-+), and Ref was and remains available (++). This adapter changes from multiple to single inheritance, replacing inheritance by delegation. Label 5 shows rule multiBySingle, which is one of the rules to handle this adaptation replacing references of type parents by parent. The editor supports code completion of rule and feature names.

Merlin-A can analyse the correctness and compatibility of the adapters, their coverage of the language configurations and features, and the reachability of the language configurations by the induced transformations. As an example, label 6 in Fig. 2 shows the Coverage view with the adapters handling each feature deactivation (+-), activation (-+), availability (++) and unavailability (--). The window with label 7 reports the features that are not used by any adapter.

2.2.5 **Generation of adaptive editor.** At this point, the language designer can automatically generate an adaptive editor by just clicking a button. The generated editor is an Eclipse plugin and resembles the standard default tree editor for EMF models, making it intuitive and familiar for EMF users. However, unlike the default editor, it allows users to select the language variant in use, and models are automatically migrated to the chosen variant (cf. Sec. 2.3).

2.2.6 **Adaptation triggers (optional).** Adaptive editors allow selecting the language variant to be used on request. In addition, the language designer can configure adaptation triggers, i.e., conditions that trigger an automatic language reconfiguration when they are met. For this purpose, adaptive editors include *hook methods* that are invoked when saving or editing the model. The designer can customise them to define triggers for language reconfigurations, e.g., based on the user editing actions or the result of OCL [22] queries evaluated on the current model.

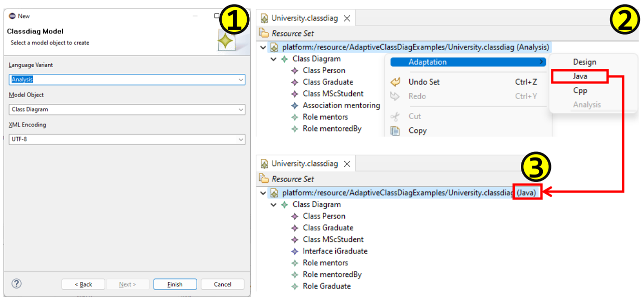

Figure 3: Generated editor for adaptive class diagrams.

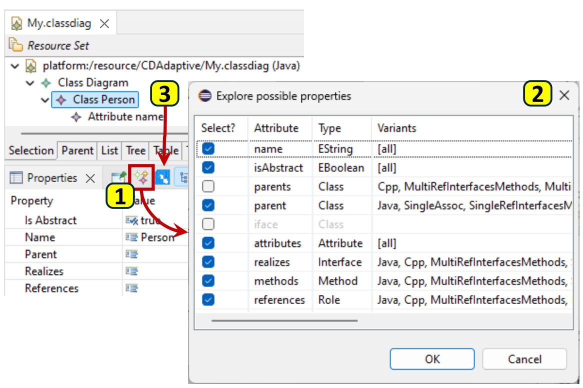

Figure 4: Property explorer for variant recommender.

In the example, we configured the OnEdit hook to detect a non-optimal workaround in the model being edited, which signals the need to use another language variant. In particular, we apply an OCL query to detect if two opposite unidirectional references exist within the model. If so, the editor suggests using a language variant with bidirectional associations to simplify their handling.

2.3 Using an adaptive language

Fig. 3 displays some screenshots of the generated tree editor for the example. Label 1 shows the model-creation wizard, which includes combo-boxes to select the initial language configuration (Analysis) and the type of the model root object.

Label 2 shows the tree editor, which is used in the standard way to create models of the selected configuration. The top node displays the file name (after platform:) and the current language variant (Analysis). The editor dynamically inspects the current language configuration and adapts its behaviour, hiding the menus and fields for creating and editing objects and properties unsupported in the current configuration, and omitting the check of the cardinality and OCL constraints absent from the configuration.

While modelling, the configured adaptation triggers are evaluated in the background when editing or saving the model, and may trigger language reconfigurations. In addition, the editor includes the contextual menu Adaptation (cf. label 2) to change the language configuration manually. Upon selecting a new language configuration, a migration is executed and the model is updated (label 3). The figure shows the adaptation from Analysis (label 2) to Java (label 3), which replaces associations by unidirectional references.

The editor also includes two built-in language variant recommenders, accessible from the Properties view of objects. The first one assists in selecting language variants that support objects with certain desired properties. Fig. 4 shows its trigger from the Properties view, where a Class object is being edited (label 1). The window with label 2 allows (de)selecting desired properties, and the recommender enables selecting among the variants that respect the selection of properties, performing the corresponding language reconfiguration and model migration. The second recommender is activated by the button with label 3. This lists all language variants in which the current model is valid, ranked by size, and enables reconfiguration and migration. This helps transition to a simpler language variant that is expressive enough for the current model.

3 Related Work

Several works propose product lines of modelling languages, either using composition (i.e., assembling components) [4, 7, 9, 15], transformation (i.e., transforming a core model) [12, 25] or annotations (i.e., extracting the variants from a superlanguage that superimposes them all) [5, 11, 24, 29]. We follow an annotative approach, but go beyond these works by considering language reconfigurations, model migration, adaptation triggers and recommenders.

Gradual programming introduces programming concepts step by step via language versions of increasing complexity. For instance, Hedy [13] is a gradual programming language (based on Python) for education. Adaptive languages can also encompass language variants in a learning process, and migrate models between variants, which is not possible in Hedy (Hedy programs are not transferred between languages). Storm and Hermans [28] define textual gradual languages by gradual extension of grammars with syntactic constructs in consecutive levels. We go beyond, considering migration between versions, adaptation triggers, and recommenders.

Self-adaptive languages [17] adapt their runtime semantics depending on the context to obtain some trade-off. Adaptivity is achieved by incorporating feedback loops within the virtual machine at runtime [16]. This run-time adaptation is orthogonal to our design-time adaptation, as it entails a modification of the language semantics but not of the language itself.

Finally, metamorphic languages [1] have several shapes, like internal, external, or using fluent APIs. Instead, Merlin-a enables language variants and their reconfiguration.

4 Conclusions and Future Work

This paper presented Merlin-a, an Eclipse plugin for building adaptive languages, i.e., families of language variants for specific contexts or user profiles, which can be reconfigured dynamically. The adaptive language editors permit switching between the variants and automate the migration of models between them.

Merlin-a works at the abstract syntax level, so we intend to extend it to have adaptive (graphical and textual) concrete syntaxes as well. We also plan to integrate conversational assistants able to recommend language versions, and perform a user study.

Acknowledgments

Work funded by the Spanish MICINN with projects TED2021-129381B-C21 and PID2021-122270OB-I00.

References

[1] Mathieu Acher, Benoît Combemale, and Philippe Collet. 2014. Metamorphic domain-specific languages: A journey into the shapes of a language. In *Onward!@SPLASH*. ACM, 243–253.

[2] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MoDELS (LNCS, Vol. 6394)*. Springer, 121–135. See also https://projects.eclipse.org/projects/modeling.emft.henshin.

[3] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-driven software engineering in practice, Second edition.* Morgan & Claypool Publishers, San Rafael, California (USA).

[4] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2020. A compositional framework for systematic modeling language reuse. In *MoDELS*. ACM, 35–46.

[5] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying feature-based model templates against well-formedness OCL constraints. In *Generative Programming and Component Engineering, GPCE*. ACM, 211–220.

[6] Juan de Lara and Esther Guerra. 2025. Adaptive modelling languages: Abstract syntax and model migration. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 66:1–66:54.

[7] Juan de Lara, Esther Guerra, and Paolo Bottoni. 2024. Modular language product lines: Concept, tool and analysis. *Softw. Syst. Model.* In press (2024). https://doi.org/10.1007/s10270-024-01179-9

[8] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of algebraic graph transformation.* Springer.

[9] Antonio Garmendia, Esther Guerra, and Juan de Lara. 2024. Product lines of graphical modelling languages. In *Proc. of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS*. ACM, 69–79.

[10] Esther Guerra and Juan de Lara. 2018. On the quest for flexible modelling. In *MoDELS*. ACM, 23–33.

[11] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. 2022. Property satisfiability analysis for product lines of modelling languages. *IEEE Trans. Software Eng.* 48, 2 (2022), 397–416.

[12] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. 2015. Systematic synthesis of delta modeling languages. *Int. J. Softw. Tools Technol. Transf.* 17, 5 (2015), 601–626.

[13] Felienne Hermans. 2020. Hedy: A gradual language for programming education. In *ICER*. ACM, 259–270.

[14] Martin Hirzel. 2023. Low-Code Programming Models. *Commun. ACM* 66, 10 (2023), 76–85.

[15] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software language engineering in the large: towards composing and deriving languages. *Comput. Lang. Syst. Struct.* 54 (2018), 386–405.

[16] Gwendal Jouneaux, Olivier Barais, Benoît Combemale, and Gunter Mussbacher. 2021. SEALS: A framework for building self-adaptive virtual machines. In *SLE*. ACM, 150–163.

[17] Gwendal Jouneaux, Olivier Barais, Benoit Combemale, and Gunter Mussbacher. 2021. Towards self-adaptable languages. In *Onward!@SPLASH*. ACM, 1–16.

[18] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[19] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering software variability with FeatureIDE.* Springer. See also https://featureide.github.io/.

[20] Daniel L. Moody. 2009. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Software Eng.* 35, 6 (2009), 756–779.

[21] L. Northrop and P. Clements. 2002. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[22] OCL. 2014. http://www.omg.org/spec/OCL/.

[23] OCLinEcore. (last accessed in December 2024). https://wiki.eclipse.org/OCL/OCLinEcore.

[24] Gilles Perrouin, Moussa Amrani, Mathieu Acher, Benoît Combemale, Axel Legay, and Pierre-Yves Schobbens. 2016. Featured model types: Towards systematic reuse in modelling language engineering. In *MiSE@ICSE*. ACM, 1–7.

[25] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL - A delta-based modeling framework for software product line engineering. In *ASE*. IEEE Computer Society, 852–857.

[26] Davide Di Ruscio, Dimitrios S. Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. 2022. Low-code development and model-driven engineering: Two sides of the same coin? *Softw. Syst. Model.* 21, 2 (2022), 437–446.

[27] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, Upper Saddle River, NJ.

[28] Tijs van der Storm and Felienne Hermans. 2022. Gradual grammars: Syntax in levels and locales. In *SLE*. ACM, 134–147.

[29] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt. 2009. Improving domain-specific language reuse with software product line techniques. *IEEE Softw.* 26, 4 (2009), 47–53.

[30] Xtext. 2025. http://www.eclipse.org/Xtext/. (last accessed in January 2025).

Received 13 January 2025; accepted 10 March 2025