Universidad Autónoma de Madrid | Escuela de Doctorado

Doctoral Thesis

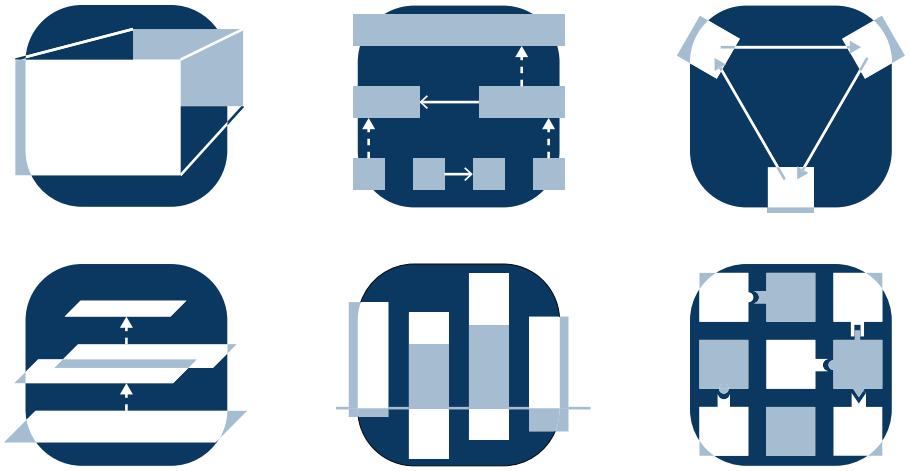
Model-Driven Engineering of Low-code Development Platforms

Francisco Martínez Lasaca

Doctorate in Computer and Telecommunication Engineering

Supervision:
Juan de Lara Jaramillo
Esther Guerra Sánchez

Madrid, 2026



Model-Driven Engineering of Low-code Development Platforms
2026, Francisco Martínez Lasaca

Abstract

The software industry faces a scalability challenge as the supply of skilled developers cannot meet the rising demand for more and faster software. Low-code development platforms (LCDPs) have been proposed as a solution, empowering domain experts to build software via visual programming. However, these platforms suffer from drawbacks that limit their flexibility and scalability, such as rigid code generation pipelines, limited customization, and vendor lock-in. While Model-Driven Software Engineering (MDSE) offers solutions to these issues, its focus on professional developers has so far limited the symbiotic integration of the two paradigms.

This thesis presents an MDSE-based approach for engineering low-code platforms that overcomes these limitations. The approach grounds LCDPs on a model-driven foundation, allowing them to scale to models with hundreds of thousands of elements, provide mechanisms for understanding extensive modeling ecosystems, offer configurable degrees of model formality, and support every component, from user management to code generation workflows. To enhance platform usability, this thesis also introduces a method for automatically generating chatbots for managing these platforms using natural language.

The approach was validated through the evaluation of prototype tools in an industrial setting with UGROUND, a software company that develops enterprise no-code platforms. The evaluations assessed the scalability, model comprehension, and development of real-world models with hundreds of thousands of elements, using quantitative and qualitative measures. The results show that the tools are scalable, help users understand large modeling ecosystems, and improve industrial development processes. These findings validate a novel method for building low-code platforms, effectively bridging the gap between low-code development and Model-Driven Software Engineering.

Keywords low-code platforms, Model-Driven Software Engineering, graphical languages, workflow languages, chatbots

Resumen

La industria del *software* se enfrenta a un desafío de escalabilidad: la oferta de desarrolladores cualificados no llega a cubrir la creciente demanda de más *software* y de desarrollarlo con mayor rapidez. Las plataformas de desarrollo *low-code* (*LCDP*) se han propuesto como una solución, ya que permiten a diferentes expertos de dominio construir *software* mediante programación visual. Sin embargo, estas plataformas adolecen de limitaciones de flexibilidad y escalabilidad, como procesos de generación de código rígidos, opciones de personalización limitadas y dependencia del proveedor (*vendor lock-in*). Por su parte, aunque el campo de la ingeniería de *software* dirigida por modelos (ISDM) lidia con problemas similares, su enfoque centrado en el desarrollo profesional ha limitado hasta ahora la integración de ambos paradigmas.

Esta tesis presenta un enfoque basado en ISDM para construir plataformas *low-code* que supera estas limitaciones. En particular, las LCDP se erigen sobre una base de ISDM, lo que les permite escalar a modelos con cientos de miles de elementos, dar soporte a mecanismos para la comprensión de ecosistemas de modelado extensos, ofrecer grados configurables de formalidad en los modelos y manejar toda clase de componentes: desde la gestión de usuarios hasta la definición de flujos de trabajo de generación de código. Además, esta tesis introduce un método para generar automáticamente chatbots para gestionar estas plataformas mediante lenguaje natural.

La propuesta se ha validado mediante la evaluación de prototipos desarrollados en un entorno industrial con UGROUND, una empresa de *software* que desarrolla plataformas *no-code* empresariales. Las evaluaciones midieron la escalabilidad, la comprensión de modelos y el desarrollo de modelos realistas con cientos de miles de elementos mediante medidas tanto cuantitativas como cualitativas. Los resultados demuestran que las herramientas desarrolladas son escalables, ayudan a los usuarios a comprender grandes ecosistemas de modelado y mejoran los procesos de desarrollo industrial. Se valida así un método para la construcción de plataformas *low-code*, cerrando la brecha entre el desarrollo *low-code* y la ingeniería de *software* dirigida por modelos.

Palabras clave plataformas *low-code*, ingeniería de *software* dirigida por modelos, lenguajes gráficos, lenguajes de flujos de trabajo, chatbots

Agradecimientos

A quien, por sus obras, los merezca.

— Jesús G. Maestro

x

Contents

Part I

Preliminaries

Chapter 1

Introduction

This chapter motivates the thesis (1.1), summarizes its technical contributions (1.2) and associated publications (1.3), and outlines its overall structure (1.7). Additionally, it describes the industrial collaboration that supported this work (1.4), a research visit undertaken during the PhD (1.5), and the financial support received (1.6).

1.1 Motivation

Software underpins nearly every aspect of modern society: from smartphones, social media, and video games to public and private organizations, engineering practice, and safety-critical systems such as intensive care units, railway control systems, or nuclear power plants [1], [2]. As society becomes increasingly digital [3], the demand for software has grown substantially, driving the search for faster and more scalable development processes [4].

Software engineering is the discipline that addresses this demand by providing methods and tools for developing complex software systems. Advances in cloud computing have enabled highly scalable, web-based development environments [5]. More recently, the advent of large language models (LLMs) [6] in the field of Artificial Intelligence has opened up new possibilities for software development [7]. For example, it is now possible to create simple applications using natural-language prompts —a trend colloquially known as '*vibe coding*'— and deploy them directly to cloud platforms [8]. However, these approaches are currently unreliable for complex projects, often contain errors, and ultimately require expert supervision [9], [10], [11]. Moreover, most of these advances primarily benefit technical users, limiting their accessibility to a broader audience.

As a result, there is a need for tools that enable citizen developers —individuals without a technical background— to participate in software production. This is feasible because many non-programming professionals

now routinely use complex digital tools, allowing them to easily become proficient with them [12], [13]. Such tools are called low-code development platforms (LCDPs) [14], and they let users build applications in the browser without installation. Citizen developers drag and drop components in visual editors and deploy applications to the cloud with one click [15]. However, LCDPs face well-known limitations, such as rigid code generation that hinders complex projects, and closed formats that limit interoperability and foster vendor lock-in [16].

Beyond low-code, Model-Driven Software Engineering (MDSE) focuses on a very similar objective: raising the level of abstraction to speed up development. In particular, MDSE investigates the development and transformation of models (i.e., purposeful abstractions) to automatically generate applications or other artifacts [17]. This approach is similar to most other engineering disciplines, where blueprints guide the stages that transform a specification into a final artifact [18]. However, MDSE distinguishes itself from other engineering disciplines because all involved artifacts are digital; models are stored, manipulated, and interpreted by computers [19]. This renders models particularly powerful, as they serve not only as specifications but also as execution drivers exploitable through model transformations [20].

A closely related field is Language Engineering [21], which focuses on building domain-specific languages (DSLs) that are tailored to solve a collection of domain-focused problems [17]. In particular, DSLs narrow the gap between the problem and solution spaces by defining a limited set of relevant primitives and constructs [22], [23]. Furthermore, DSLs enable domain experts to develop software without requiring programming knowledge, as technical concepts (e.g., loops, variables, or functions) are replaced by a domain-specific language that becomes a shared vocabulary and facilitates communication [24], [25]. Due to the universal expressivity of languages and models, language engineering and MDSE overlap significantly, with DSLs often being implemented using models [17]. Overall, the goal is to express a problem in a DSL and encode it in a model that is interpreted or transformed to generate a solution [18]. However, DSLs face multiple issues. First, they are usually designed for professional development environments and technical users (e.g., within the Eclipse IDE) [26], [27], which increases friction in their adoption. Additionally, dependence on desktop tooling and file-based storage limits scalability [28], [29]. Finally, the reliance on compiled languages can hinder rapid development [30].

Recently, researchers have proposed integrating low-code platforms with MDSE to leverage their strengths and mitigate their limitations. Broadly speaking, these disciplines are understood as two sides of the same coin —each with distinct strengths and weaknesses [31]. Their combination has been coined as low-code engineering platforms (LCEPs; cf. LCDPs), which systematically ground low-code design and implementation in MDSE [32].

Following this approach, MDSE underpins the storage, management, and manipulation of models (domain-specific data) while remaining transparent to end users. Ultimately, this approach retains the benefits of low-code development, allowing for agile software creation, while leveraging MDSE's battle-tested model management infrastructure [32].

However, integrating these paradigms poses several challenges. First, scalability remains an ongoing problem in MDSE, as most of the tooling, such as IDEs or formats (e.g., XMI), is not designed to handle large models [29]. Second, traditional MDSE technologies are not well adapted for low-code development scenarios. Low-code platforms manage permissions, users, files, and workflows —entities for which existing MDSE technologies lack expressiveness [33]. Moreover, existing integrations often lack flexibility for informal modeling and user-driven code generation. Finally, mechanisms for understanding large models —which are common in industrial low-code platforms— and for specifying application behavior are limited [34], [35].

To validate that MDSE enhances both low-code tooling and development processes, this thesis pursues the following overarching objectives:

O1. Provide model-driven low-code tooling capable of handling very large models (with hundreds of thousands of elements) without compromising usability in industrial contexts.

O2. Provide graphical, efficient mechanisms for rapidly understanding very large models.

O3. Enable the automated construction of low-code platforms with most features found in mainstream low-code platforms, including those with large user bases and supported by major vendors.

O4. Explore the integration of LLM-driven chatbots that let users manage and query low-code platforms via natural language, leveraging the ontological nature of low-code platforms.

This thesis presents Dandelion and Dandelion+, scalable low-code engineering platforms with a model foundation. Dandelion and Dandelion+ explore complementary techniques, including model flexibility, model-based diagramming, and application behavior modeling. Moreover, the thesis introduces LowcoBot, a model-based chatbot generator for low-code platforms, including one for Dandelion+. These tools were developed in collaboration with UGROUND [36], a company specializing in industrial low-code platforms.

1.2 Technical contributions

The technical contributions of this thesis are:

1. Dandelion (cf. Chapter 4): a cloud-based graphical language workbench. The platform allows users to define concrete syntaxes directly in a web browser, introduces a harmonizing meta-model that bridges heterogeneous technologies, supports configurable, scalable exploration and visualization of large-scale models, and offers flexible modeling to control the desired degree of formality in artifacts created by citizen developers.
2. Model sensemaking strategies (cf. Chapter 5): a model-driven mechanism to define purposeful (meta-)model visualizations. It is implemented atop Dandelion, including a catalog of the most commonly used strategies out of the box.
3. Dandelion+ (cf. Chapter 6): a model- and workflow-driven approach for engineering domain-specific low-code platforms and applications. Specifically, Dandelion+ provides a model-driven architecture and PlatFlow, a workflow language that defines low-code application behavior.
4. LowcoBot (cf. Chapter 7): a model-driven tool that generates chatbots tailored to low-code platforms. Using it, a chatbot has been developed for Dandelion+, allowing users to manage the platform directly from a textual prompt.

These contributions directly map to the previously outlined objectives: Dandelion tackles O1, model sensemaking strategies address O2, Dandelion+ focuses on O3, and LowcoBot targets O4. In particular, objectives O1–O3 have been validated using large-scale industrial models with hundreds of thousands of elements in the context of Dandelion and Dandelion+, while O4 has been explored through a case study implementing a chatbot for Dandelion+ using LowcoBot.

An online instance of Dandelion, documentation, and experiment data for Dandelion, Dandelion+, and sensemaking strategies are available at `https://miso.es/tools/Dandelion.html`. LowcoBot is open-source and its implementation is available at `https://github.com/LowcoBot/lowcobot`.

1.3 Publications

This thesis has led to the following publications:

Journals

1. Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara (2023). *Dandelion: A scalable, cloud-based graphical language workbench for industrial low-code development*. Journal of Computer Languages, 76, 101217 [37]. (JCR 2023: IF 1.7, Q3). See Chapter 4.
2. Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara (2025). *A Model and Workflow-Driven Approach for Engineering Domain-Specific Low-Code Platforms and Applications*. Software and Systems Modeling [33]. (JCR 2024: IF 3.2, Q2). See Chapter 6.

Conferences

3. Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara (2023). *Model sensemaking strategies: Exploiting meta-model patterns to understand large models*. In Proceedings of the 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pages 261–272. IEEE [34]. (Core A). See Chapter 5.
4. Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara (2023). *Engineering low-code modelling environments with Dandelion*. In Proceedings of the 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems [38]. (Core A). See Chapter 4.

Workshops

5. Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara (2024). *LowcoBot: Towards chatting with low-code platforms*. In STAF 2024 Workshops, volume 3727 of CEUR Workshop Proceedings, pages 66–76. CEUR-WS.org [39]. See Chapter 7.

1.4 Industrial collaboration

This thesis is situated within the context of Lowcomote (2019–2023), a Marie Skłodowska-Curie European Training Network.[1] It was established to train early-stage researchers to develop scalable low-code engineering platforms (LCEPs) that integrate model-driven engineering, cloud computing, and machine learning, fostering collaboration between academia and industry. Specifically, this thesis is a joint effort between the Modelling & Software Engineering (MISO) research group at the Autonomous University

[1] https://lowcomote.eu/

of Madrid and UGROUND. The company leverages ROSE [40], a patented ontology-based platform, in its products. This model-based approach enabled a synergistic collaboration between the university and the company. UGROUND provided real industrial-scale models, enabling realistic evaluations of the developed approaches; in turn, the author identified and helped improve parts of the company's development process.

1.5 Research visit

In June 2023, the author undertook a secondment at the Department of Computer Science at the University of York under the supervision of Prof. Dimitris Kolovos. The visit was planned in the context of Lowcomote, where the author gained a deeper understanding of Eclipse Epsilon [41], a central technology in the development of Dandelion+ (cf. Chapter 6).

1.6 Financial support

The first half of the doctorate was supported by Lowcomote [32] (2019–2023) under the Marie Skłodowska-Curie grant agreement No. 813884. Subsequently, UGROUND provided financial support. This work also received funding from the Spanish MICINN (grants PID2021-122270OB-I00, TED2021-129381B-C21, PID2024-155231OB-I00, and RED2022-134647-T).

1.7 Thesis structure

This dissertation is structured as illustrated in Figure 1.1:

I. Preliminaries

1. Introduction. Introduces the thesis, its contributions, and structure.

2. Background. Covers the concepts of Model-Driven Software Engineering and low-code development necessary to understand the thesis.

3. Related Work. Reviews the state of the art in cloud-based graphical editors, model scalability, and model visualization.

II. Contributions

4. Definition of Graphical Language Workbenches in the Cloud. This chapter details Dandelion, a scalable, cloud-based graphical language workbench.

5. Purposeful Visualizations for Understanding Models. This chapter explores the concept of model sensemaking strategies (SMSs) and their implementation in Dandelion.

6. Modeling Low-code Platforms: Structure and Behavior. Dandelion+, a model and workflow-driven approach for engineering low-code platforms, is presented in this chapter.

7. AI Assistants for Low-code Platforms. Describes LowcoBot, a tool for generating LLM-based chatbots for low-code platforms.

III. Evaluations

8. Evaluations. Validates the contributions from the previous part through several large-scale experiments.

IV. Conclusions

9. Conclusions and Future Work. Summarizes the findings of the thesis and outlines avenues for future research.

10. Conclusiones y Trabajo Futuro. *(Spanish version of Chapter 9).*

V. Bibliography References cited throughout the thesis.

VI. Appendices

A. Catalog of Model Sensemaking Strategies Presents an exhaustive catalog of model visualization techniques introduced in Chapter 5.

B. Addenda to PlatFlow Complements some technical details about the implementation of PlatFlow in Chapter 6.

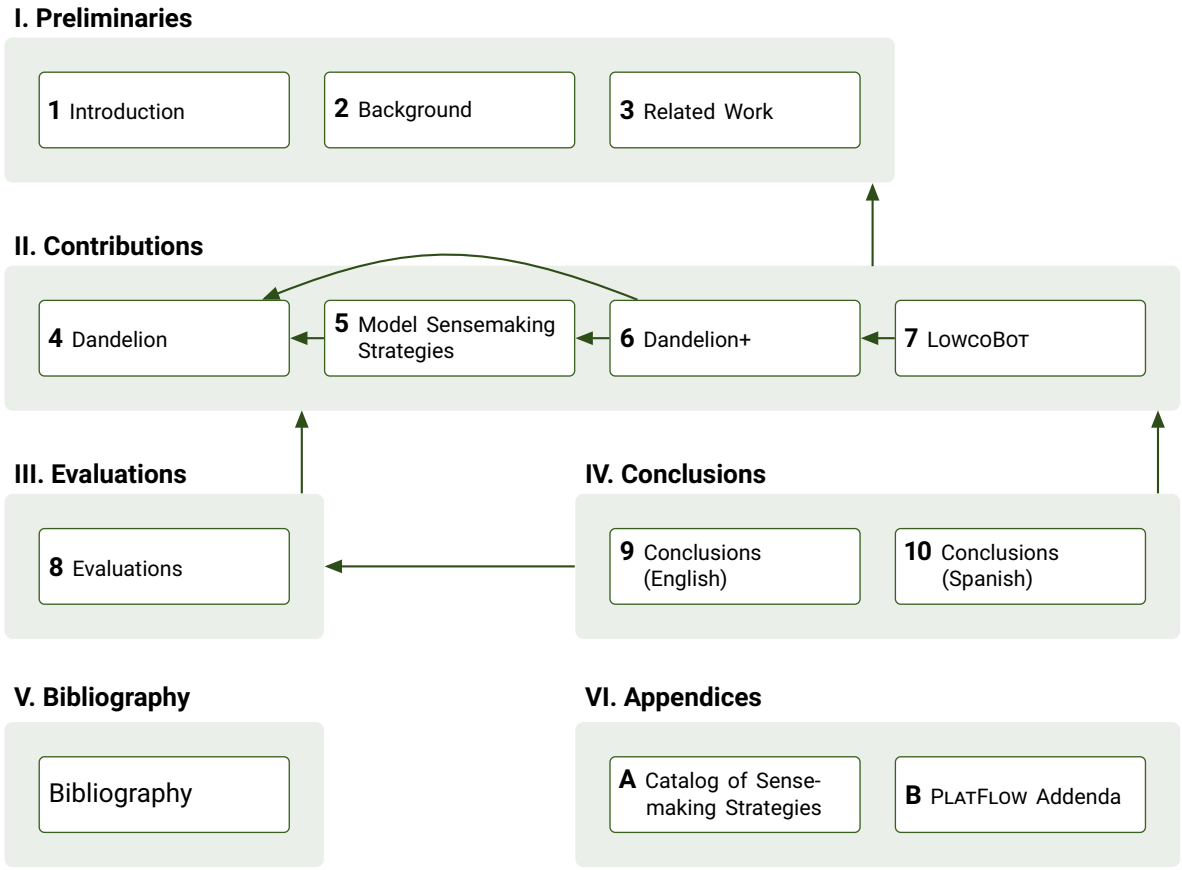


Figure 1.1 Structure of the thesis: parts, chapters, and dependencies.

Chapter 2

Background

This chapter introduces Model-Driven Software Engineering (2.1) and low-code development (2.2), and discusses their synergies and challenges (2.3) to provide the necessary context to understand the thesis.

2.1 Model-Driven Software Engineering

This section defines Model-Driven Software Engineering, its architecture, terminology, and key concepts, including domain-specific languages, multi-level modeling, flexible modeling, model transformations, and model management tooling.

2.1.1 Introduction

All scientific disciplines rely on models, albeit in different ways, to tackle the complexity of different aspects of reality. From Newton's classical mechanics in physics [42] to climate models in meteorology [43] and blueprints in architecture, models serve as domain-focused artifacts that allow us to represent, understand, and manipulate complex systems through appropriate abstractions [44].

Model-Driven Software Engineering (MDSE) is a subfield of Software Engineering that relies on the systematic use of models to build software systems and support development activities via automated transformations [17]. While classical software development often treats diagrams and other semi-formal artifacts as documentation —often ending up as *post-mortem* documentation— MDSE treats models as core artifacts in the development process [20]. The contrast is even clearer when compared to *ad hoc* programming, where the entire development process is handcrafted and unstructured [45]. Finally, unlike other engineering approaches, MDSE has a distinctive feature: its artifacts are digital (cf. disciplines like mechanical

engineering or architecture), allowing a seamless transition between models and software through automated transformations [19].

Historically, MDSE has its roots in the 1960s–1980s, with the emergence of formal modeling and abstraction methods. Notable examples are structured analysis and design [46], entity-relationship modeling [47], and computer-aided software engineering (CASE) tools [48]. Later, the Unified Modeling Language (UML) was introduced in 1997 [49], becoming a standard of the Object Management Group (OMG). This marked a milestone in the discipline, as UML provided a common modeling notation on which to build model-based tooling. However, a common vocabulary was insufficient, as proper semantic foundations, architectural support, and transformation technologies were missing [50]. To mitigate these limitations, the OMG developed the Model-Driven Architecture (MDA; 2001) [51], a UML-based approach for developing software in a model-driven fashion. However, MDA achieved only partial success, as UML was never intended to encode full executable system logic, and its behavioral notation proved inadequate for supporting large-scale model execution. More recently, the focus has shifted towards Domain-Specific Languages (DSLs), which bridge the gap between the problem space and the solution space [52]. In this setting, meta-modeling languages are used to define DSLs, on top of which code generators can be defined and executed, rather than forcing modeling languages such as UML into all stages of software development.

Although MDSE offers numerous benefits (e.g., in development automation, stakeholder participation, complexity reduction) [18], it does not suit all scenarios. For instance, while designing DSLs, available tooling may not provide all the necessary features for rapid development (e.g., debuggers, syntax highlighters, type systems, and refactoring tools). Moreover, exposing model-centric constructs to end users may be intimidating. This thesis addresses some of these issues by incorporating low-code elements into an MDSE-based development process and, conversely, by grounding low-code development in MDSE principles.

2.1.2 Terminology and concepts

A constellation of acronyms Several acronyms are commonly used in the literature to refer to model-related approaches, as presented in Table 2.1.

Two dimensions can be observed: model-based vs. model-driven and engineering vs. software engineering. Regarding the model-based vs. model-driven distinction, model-driven approaches specifically refer to those where models play a central role in defining and generating the final product, while, in model-based ones, models may not be as central. The consensus is that model-based practices encompass model-driven ones. As per the engineering vs. software engineering dimension, the *software* qualifier simply makes the

	Model-Based...	**Model-Driven...**
Engineering	MBE	MDE
Software Engineering	MBSE	MDSE

Table 2.1 Summary of the most common acronyms in model-related approaches.

scope specific to software systems, to distinguish it from other engineering disciplines.

Ultimately, real-world boundaries are often blurred, leading to these acronyms sometimes being used interchangeably in the literature. This thesis, however, strives for a systematic use of models throughout the tools it presents and, thus, mainly adheres to the Model-Driven Software Engineering (MDSE) category.

Models and meta-models Models are domain-specific abstractions of a system that capture relevant aspects about it [17]. Models can be classified as descriptive, if they describe a system that already exists, or as prescriptive, if they describe systems that do not exist yet. The latter can be placed on a spectrum of formality, whether they act as sketches, blueprints, or programs. This thesis mainly focuses on prescriptive, formal models that work as programs and, therefore, unambiguously define the part of the system they encode. However, thanks to the flexible modeling mechanisms, which are explained later, this thesis also covers the possibility of using models as sketches.

Models are not isolated. Quite the contrary, the basis of MDSE lies in relating models to each other. In particular, meta-models (from the Greek prefix "μετα-" meaning *beyond* or *about*) specify the syntax of the models that they describe. Specifically, models *conform to* (or are *instances of*) meta-models. Models *consist of* model elements which, in turn, are *instances of* the types defined in meta-models. The elements of a meta-model are typically called meta-classes. The process of meta-modeling consists of applying the aforementioned concepts and relationships to build one or more conformance layers of models and meta-models. Moreover, as abstraction increases, this is reflected in the number of "metas" (e.g., *meta-meta-model*).

Concepts Consider the example from Figure 2.1, showcasing a simple task management meta-model (e.g., a Kanban board) and an instance of it.

The left-hand side displays the meta-model, which contains meta-classes like Project, Stage, or Task. Meta-classes have attributes, like the name or description of an Item. They are typed, meaning that their values must belong to a specific primitive type, such as String, Integer, or Boolean. Meta-classes have relationships between them. For instance, a project may define several steps, a fact that is represented via a containment relationship

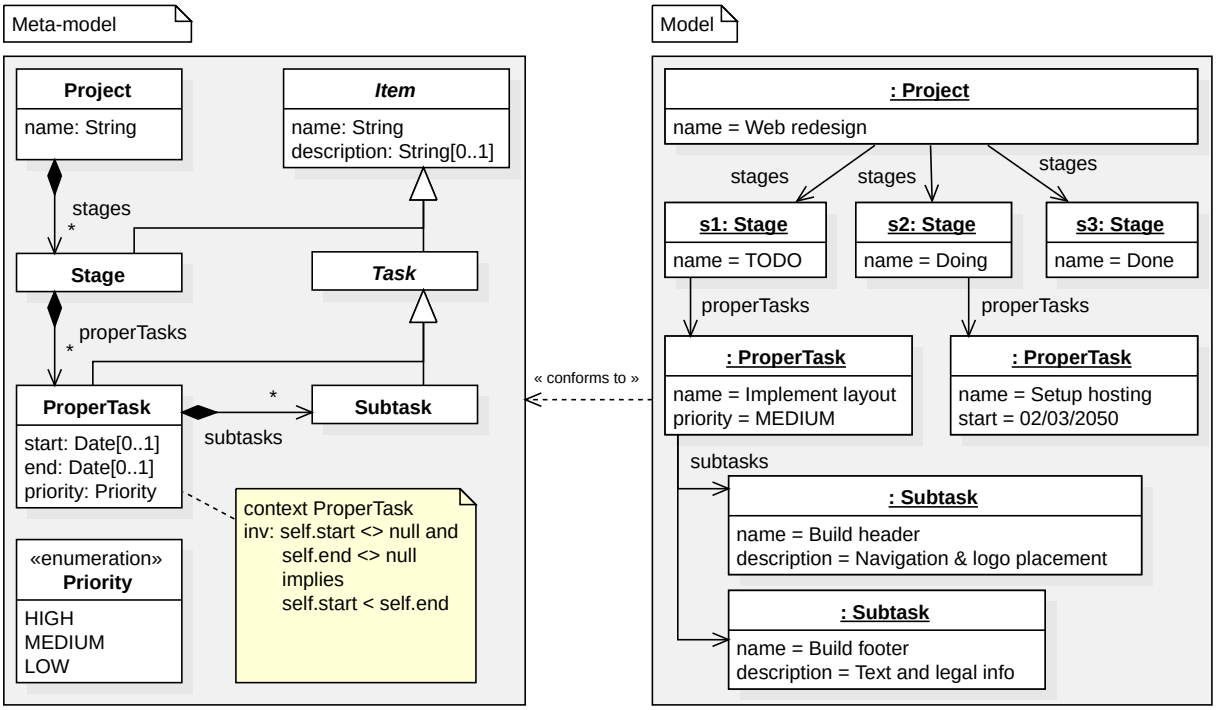


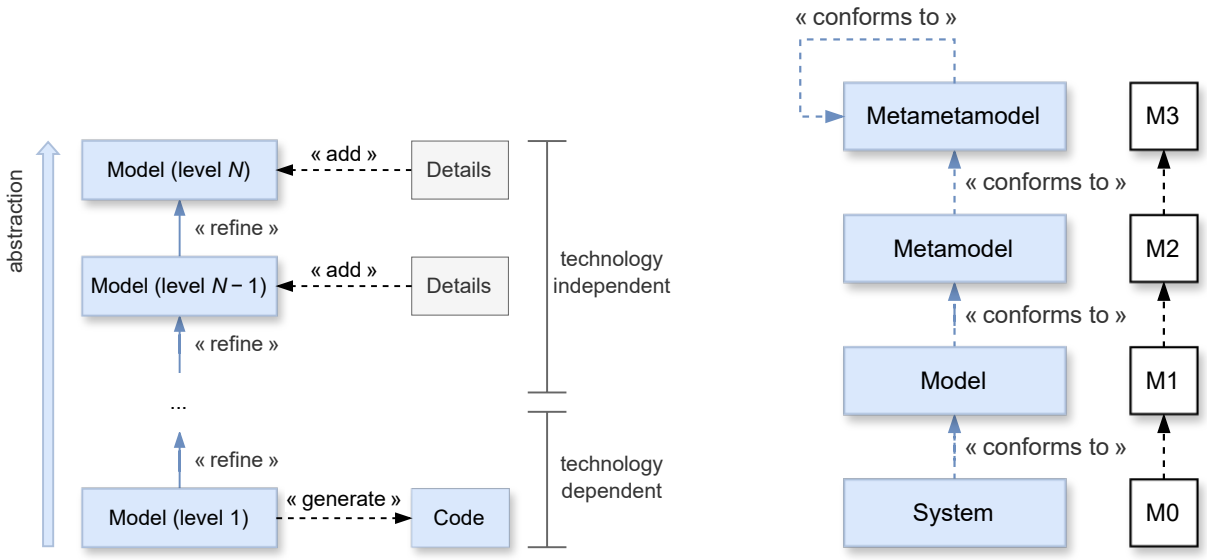
Figure 2.1 Example of a task management meta-model and an instance.

between them. The arrow is decorated with *, denoting a multiplicity that indicates that a project may have zero or more stages. The title of the Task and Item meta-classes is in italics, which means that they are abstract (meta-)classes: they cannot be instantiated directly. Instead, they define subclasses through inheritance (i.e., between Task and its subclasses ProperTask and Subtask). Observe how ProperTasks have priorities, which are defined in an enumeration with three literals or *literal values*. Finally, the ProperTask meta-class is annotated with an Object Constraint Language (OCL) integrity constraint. In this case, it ensures soundness between the start and end dates of a task, if any.

The right-hand side displays a model, which is an instance of the previous meta-model. Note that the instantiated objects such as : Project or s1: Stage all have a type (i.e., Project and Stage, respectively), and may declare an explicit identity for them (i.e., s1). Note that the model conforms to its meta-model: types are respected, as well as the relationships between them and multiplicities; in particular, no constraints are violated.

2.1.3 Architecture

Model-based software systems are typically structured in a layered architecture, as shown in Figure 2.2a. In this configuration, higher layers are more abstract,



(a) A general *N*-layered MDSE architecture. Each layer adds levels of detail at a lower degree of abstraction.

(b) The four-layered MDSE architecture, as popularized by the OMG.

Figure 2.2 MDSE architectures.

while models become more detailed and concrete towards the lower layers of the hierarchy. The highest levels encode concepts of the domain that are not part of the final system, thereby permitting technology independence at these layers. On the other hand, the lowest levels reify the concepts found at higher levels and yield the final system coupled to a specific technology stack.

Although the layered architecture can extend to an arbitrary number of layers, its most common realization is the four-layered architecture, as seen in Figure 2.2b. This architecture was popularized by the Object Management Group (OMG) [53], which standardized its use in the Meta Object Facility (MOF) [54]. The four layers are: meta-meta-model (M3), meta-model (M2), model (M1), and system (M0). Each layer is an instance of the layer above, thereby establishing a conformance relation between them. For instance, the meta-model (M2) is *described* by the meta-modeling language defined at the meta-meta-model (M3), such as MOF or Ecore. In turn, the model (M1) *conforms* to the meta-model (M2), and the system (M0) *instantiates* the model (M1).

Meta-meta-model (M3) This is the highest and most abstract layer, and it defines a language to create meta-models. This layer defines elements including meta-classes, relationships, or attributes: the necessary elements to describe meta-models. As this layer closes the stack, it bootstraps itself in a self-describing way. Moreover, given its abstract

nature and the complexity of its definition, this layer is typically specified once and reused across many projects from many different domains. Notable, well-documented specifications include EMF's Ecore [26] or the aforementioned MOF.

Meta-model (M2) This layer defines the language for creating models. It specifies the types of modeling elements (meta-classes), their attributes, allowed relationships, and constraints. In essence, it establishes the grammar that models at the M1 layer must conform to. Meta-models are typically domain-specific, tailored to capture the concepts and rules of a particular application area.

Model (M1) This layer contains platform-independent models that describe the structure and behavior of a system in concrete terms, but without implementation details. An M1 model conforms to an M2 meta-model and is detailed enough to drive transformations or code generation. Models at this layer are often created by domain experts and serve as the primary artifacts for analysis, design, and communication.

System (M0) This layer represents the concrete instances of the system, including runtime objects, user data, processes, and real-world states that the software manipulates. M0 consists of the actual values and occurrences represented by the types and structures defined at the M1 layer. This is the layer where the software operates in practice, executing the logic and behavior specified in the models above. This layer, as potentially being part of a larger software system, is technology-specific and completes the model-driven stack.

This thesis makes use of the four-layered MDSE architecture in Dandelion and LowcoBot. However, this architecture falls short of fully supporting SMSs and underpinning Dandelion+, where a multi-level solution is adopted, as explained later.

2.1.4 Domain-specific languages

Domain-specific languages (DSLs) are tailor-made, specialized notations for concrete domains that solve concrete collections of problems [24]. They are, therefore, domain and purpose-specific [17].

Unlike general-purpose languages (GPLs) such as Java or Python, which aim to support a large collection of domains, DSLs deliberately restrict their scope to cater to a reduced problem space. This allows DSLs to get closer to the terms used by domain experts, and their models can be manipulated and analyzed using MDSE mechanisms. DSLs, as *languages*, can be roughly decomposed into two significant parts: syntax and semantics. On the one hand, *syntax* is "the definition of the principles and processes

by which sentences are constructed in a particular language" [55]. As a consequence, instances of a language can be understood both from structural and notational perspectives. These are called abstract syntax and concrete syntax, respectively. On the other hand, *semantics* provides meaning to the syntactic constructs defined by the language.

Abstract syntax The abstract syntax of a DSL is typically encoded by a meta-model, as defining concepts and their relationships is a natural match for meta-modeling. The meta-model introduces meta-classes for the primitives of the language, together with their attributes, references, and integrity constraints. For instance, the meta-model presented before in Figure 2.1 defines the abstract syntax of a simple task management DSL.

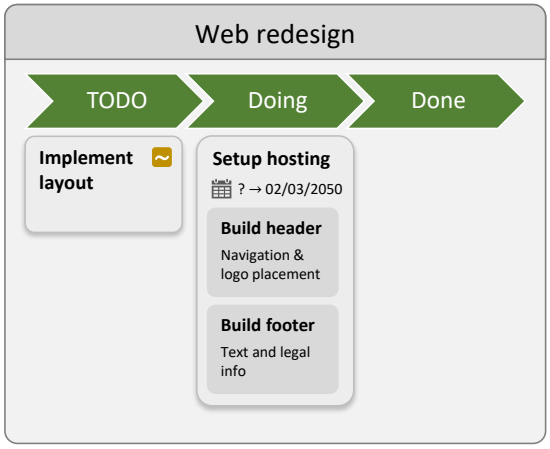
Concrete syntax The concrete syntax of a DSL determines how users interact with DSLs, from a notational perspective. The most common concrete syntax categories are textual and graphical. A DSL may support different concrete syntaxes, potentially with different natures, and even hybrid approaches [56]. Figure 2.3 demonstrates two possible concrete syntaxes for the task management DSL defined in Figure 2.1: a textual syntax (Figure 2.3a) and a graphical syntax (Figure 2.3b). Notice how both syntaxes unambiguously encode the same model, although both notations use different mechanisms to achieve it. On the one hand, the textual syntax relies on keywords (e.g., `project`, `stage`, or `t`(ask)) and punctuation to do so. On the other hand, the graphical syntax takes advantage of shapes with different features (e.g., color or shape) [57] and spatial relationships (e.g., containment, overlapping, and adjacency) [58].

Semantics The semantics of a DSL provides meaning to the syntactic constructs defined by its abstract and concrete syntax. A common distinction is between static semantics and dynamic semantics. Static semantics covers analyses that can be performed without executing the system, such as type checking, detection of inconsistent configurations, or enforcement of additional well-formedness rules beyond those captured in the meta-model. Dynamic semantics, in turn, describes how models behave when executed or simulated. This is typically done through transformations, for instance, M2T to generate code and execute it, and M2M to generate other models that can be simulated, or in-place transformations to rewrite the model.

Implementing DSLs using MDSE Meta-modeling facilities are well suited for building DSLs. Typically, DSLs are first defined using a desktop IDE like Eclipse with the definition of a meta-model using a meta-modeling language such as Ecore or MOF. Next, concrete syntaxes are defined, either textual or graphical, using dedicated frameworks (e.g., Xtext [59] for textual syntaxes

```
project "Web redesign" {
  stage "TODO" {
    t "Implement layout" @medium
  },
  stage "Doing" {
    t "Setup hosting" until 02/03/2050 {
      st "Build header"
         desc "Navigation & logo placement"
      st "Build footer"
         desc "Text and legal info"
  } },
  stage "Done" { }
}
```

(a) Textual syntax.



(b) Graphical syntax.

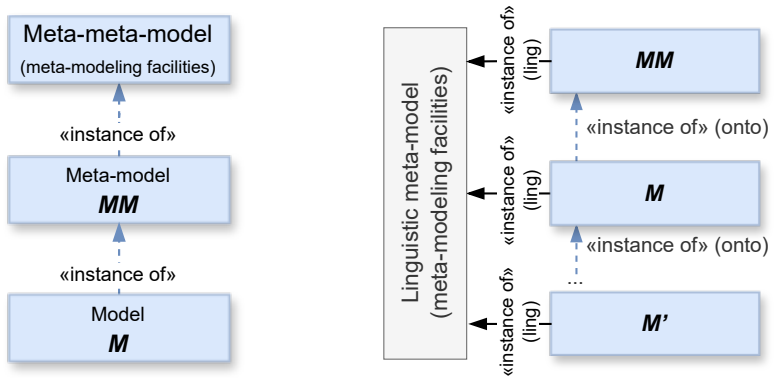
Figure 2.3 Examples of textual and graphical concrete syntaxes showing equivalent representations for the model defined in Figure 2.1.

or Sirius [60] for graphical ones). Then, semantics are either specified formally (e.g., for state machines) or through model transformations and code generators.

Low-code platforms frequently embed DSLs —even when they typically do not use this terminology to refer to them. Typical examples include languages to define data schemas (e.g., in OutSystems [61]), user interfaces, and workflows. In many platforms, these DSLs are managed through graphical editors or forms, so that users can manipulate them easily. In this thesis, both Dandelion and Dandelion+ make use of DSLs as core artifacts. In Dandelion (Chapter 4), DSLs and their graphical concrete syntaxes are defined and edited as models, while Dandelion+ (Chapter 6), model sensemaking strategies (Chapter 5), and LowcoBot (Chapter 7) build on these DSL foundations to define, visualize, and interact with low-code applications at the level of domain concepts.

2.1.5 Multi-level modeling

A traditional layered architecture (cf. Figure 2.4a) is typically sufficient for many MDSE/DSL scenarios: the primitives of a language are encoded in the meta-model (MM), whereas the model (M) instantiates it for the concrete domain-specific reality. However, when the domain itself deals with meta-modeling features such as instantiation, special typings, or inheritance, this architecture may prove insufficient. Typical examples include product types and instances in e-commerce applications, component types and instances in architectural languages, and task types and instances in process modeling languages [62]. In these cases, workarounds contained within



(a) Standard meta-modeling architecture. **(b)** Orthogonal classification architecture.

Figure 2.4 Comparison between meta-modeling architectures. Adapted from [62].

the traditional architecture (e.g., static types, explicit dynamic types, or promotion transformations [63]) fail to capture the requirements of the domain or introduce brittle transformations.

A solution is to use multi-level modeling (or *deep meta-modeling*), a paradigm where type-object, dynamic features, dynamic auxiliary domain concepts, relation configurator, and element classification features are provided natively. To achieve this, multi-level modeling depends on three central ideas. First, elements have a dual type and instance facet: instead of classes and objects, the paradigm uses *clabjects* (from *cla*ss + o*bject*) [64]. Second, models, clabjects, and their properties have a potency, which is a non-negative integer that indicates how many times the element can be instantiated across meta-levels. Every time an element is instantiated, its potency decreases by one. Finally, each (meta-)model is annotated with a level, which indicates the potency of its elements.

A key concept in multi-level modeling is the Orthogonal Classification Architecture (OCA; see Figure 2.4b) [65]. Multi-level modeling often adopts this architecture, whereby each element participates in two independent (i.e., orthogonal) classification relationships: an ontological type and a linguistic type. On the one hand, ontological typing generalizes the traditional type-to-object relationship within domains. It is the classification relationship that links a domain instance with its ontological type in the conceptual hierarchy of the domain, independently of the meta-modeling technology used. For instance, in Figure 2.1, object s1 is ontologically typed by Stage. On the other hand, linguistic typing is the classification relationship that links any modeled element with the primitive used to represent it within the meta-modeling language. For example, in the same setting, s1 and Stage may be linguistically typed by entities such as Object and Metaclass in a multi-level meta-modeling language that distinguishes these categories, or both may be instances of a single generic Clabject entity.

In this thesis, multi-level modeling is reflected in how model sensemaking

strategies (SMSs, cf. Chapter 5) and Dandelion+ (Chapter 6) are architected. SMSs can be bound either to domain meta-models (ontological typing), to analyze instances of a particular DSL, or to the linguistic meta-model of the meta-modeling language (linguistic typing), yielding domain-agnostic visualizations over entire modeling ecosystems. Likewise, Dandelion+ is built on a multi-level foundation: it provides a linguistic meta-model tailored to low-code applications, for which users define their DSLs and applications on top of it.

2.1.6 Flexible modeling

Most MDSE frameworks assume that models are well-formed at all times. That is, every element is typed, all attributes have a value of the correct type, multiplicities are respected, and integrity constraints hold. This has several benefits, including guaranteeing the soundness of transformations and analyses, and ensuring well-typed relationships between different levels of the modeling stack. However, strict conformance can pose a barrier in several scenarios. For instance, during early design stages, this rigidity can hinder creativity; in reverse engineering scenarios or legacy technology, conformance may not be fully met; and in many practical settings, engineers need to work with *partially specified* or *incomplete* models that are temporarily inconsistent with their meta-models [66].

Flexible modeling addresses these issues via mechanisms to specify and regulate the degree of conformance and rigidity between models and their meta-models. Instead of enforcing all well-formedness conditions at all times, flexible modeling approaches allow certain violations to persist temporarily, or postpone specific checks to later phases of the process. Common features of flexible modeling include modeling processes that define the different phases of modeling and the checks that apply in each phase, support for different modeling intents (e.g., bottom-up or top-down modeling), and assistance through quick fixes that help users repair or complete models when they are ready to enforce stricter conformance [67].

In this thesis, Dandelion implements flexible modeling mechanisms by providing explicit and configurable modeling processes, where each phase can enable or relax specific conformance checks. This allows users to work with partially specified graphical models during exploratory or early design phases, and then gradually tighten the degree of strictness as the language and its models mature, with quick fixes supporting common repair and completion scenarios.

2.1.7 Model transformations and code generation

Model transformations encompass the collection of techniques that consume models and produce other artifacts —possibly other models— in an auto-

mated fashion. They are key enablers of MDSE, as they are involved both in internal stages (e.g., for intermediate model representations) and in final stages, which generate the final artifacts. There are two major categories of model transformations: model-to-model (M2M) and model-to-text (M2T) transformations [68].

Model-to-model (M2M) transformations are used in a collection of scenarios, including version migration, model refactoring, and modernization of legacy systems [18], [69]. M2M transformations can be classified as *endogenous*, when the source and target meta-models coincide, or *exogenous*, otherwise [70]. Moreover, M2M can be further classified as *in-place*, when the source model is updated directly, or *out-of-place*, when a new target model is created [71]. Endogenous transformations can be either in-place or out-of-place, while exogenous transformations are typically out-of-place.

Model-to-text (M2T) transformations, in turn, take models as input and produce textual artifacts (e.g., source code, configuration files, or documentation) that can be consumed by other tools. Code generation is a specific type of M2T transformation that produces textual artifacts that conform to the syntax and semantics of a target programming language. Code generation is particularly relevant in model-based scenarios, as it allows preserving technology independence at the modeling level until the code generation step, where multiple technologies can be targeted from the same model using one or more code generation steps. Moreover, assuming the soundness of the source model, correctness of the generated code —which may involve high variability and complexity— is reduced to the correctness of the code generator, which typically relies on template-based approaches [72].

In this thesis, both Dandelion+ and LowcoBot make use of model transformations. The former defines code generation facilities for implementing low-code applications atop Epsilon M2T transformations [72]. Moreover, Dandelion+ implements parsers that consume common artifact languages (e.g., Emfatic, Flexmi, and JSON) via M2M transformations. Finally, LowcoBot relies on an M2T architecture that generates chatbot tools out of a chatbot model specification.

2.1.8 Model management tooling

The ecosystem of available tooling strongly shapes MDSE practice. In many setups, the Eclipse Modeling Framework (EMF) [26] serves as the core upon which other tools are built. For instance, editor frameworks such as Xtext [59] for textual syntaxes and Sirius [60] or Graphiti [73] for graphical notations offer facilities for building DSL editors. Common model management tasks are then automated by dedicated transformation and code generation engines: model-to-model transformations are often expressed in languages such as ATL [74], while model-to-text and code generation

are typically implemented using template-based tools such as Acceleo [75] or Xtend [76]. Together with validation frameworks, these tools enable the systematic editing, transformation, validation, and generation of artifacts from models. However, the heterogeneity of these standalone languages can lead to a "Tower of Babel" effect, as language fragmentation can hinder a systematic coordination of model management tasks. For this reason, this thesis makes extensive use of Eclipse Epsilon [41], a family of model management languages that operate under a unified framework.

At its core, Epsilon relies on the Epsilon Object Language (EOL) for querying and manipulating models [77], together with task-specific languages such as the Epsilon Transformation Language (ETL) for model-to-model transformations [78], the Epsilon Generation Language (EGL) and the EGL Co-Ordination Language (EGX) for template-based artifact and code generation [72], and the Epsilon Validation Language (EVL) for expressing validation rules and quick fixes [79]. These languages operate atop the Epsilon Model Connectivity (EMC) layer [41], which abstracts over persistence technologies, allowing the reuse of model management logic across artifacts expressed in heterogeneous formats. In this thesis, Epsilon is used extensively in Dandelion+ (Chapter 6), as all the model manipulation and code generation tasks in PlatFlow are implemented using Epsilon languages.

2.2 Low-code development

This section introduces low-code development, its stakeholders, usage scenarios, and architecture.

2.2.1 Introduction

Low-code development is a software development paradigm that seeks to speed up development by replacing hand-coding with visual modeling and pre-built components. Instead of writing code, users build applications by dragging and dropping elements, configuring their properties, and defining behavior through forms or visual artifacts [80]. Compared to purely hand-coded approaches, low-code platforms raise the level of abstraction and automate much of the underlying infrastructure.

Although the term *low-code* gained traction in the late 2010s, its roots can be traced back to contributions such as fourth-generation languages (4GLs), rapid application development tools, and visual database and form builders, which sought to reduce handwritten code and accelerate productivity via specific notations and generators [81]. Modern low-code platforms build upon these foundations in cloud-based or platform-as-a-service (PaaS) settings, providing browser-based modeling tools, integrated persistence and

deployment, and connectors to external services [82]. Overall, although the term only entered the scientific literature in 2017, the number of related publications has been growing since then, consolidating low-code as an emerging research field in its own right [83]. From a research perspective, these platforms are closely related to work on end-user development, end-user programming, and end-user software engineering [84].

There is no universally accepted definition of what constitutes a low-code platform. Surveys of existing tools reveal that most of them share a layered architecture and core features. For instance, most platforms combine visual composition tools, workflow mechanisms, integrated data management, and connectors to external services. They also typically support the deployment of the developed applications in cloud-based environments [15]. At the same time, these tools exhibit high variability regarding model-driven features, workflow management, access control, and third-party integrations, as they can cover a wide range of target scenarios [85].

This thesis focuses on a particular subset of low-code platforms that are deliberately grounded in model-driven foundations, leading to the notion of low-code engineering platforms (LCEPs). In particular, the LCEPs developed in this thesis are DSL-based, model-driven application platforms in which users work primarily with domain-specific data and application models through visual or form-based editors. These platforms automatically interpret or generate executable applications from these models [31], [32]. Under this definition, simpler scripting tools and basic form wizards are excluded. Within this scope, these platforms include tools designed for citizen developers, offering graphical environments for small automation tasks, as well as enterprise-grade environments targeting professional development teams [86].

2.2.2 Citizen developers and usage scenarios

The main stakeholders of low-code platforms are citizen developers: professional individuals who possess digital skills and domain knowledge. Their backgrounds are diverse, coming from areas such as operations, finance, logistics, or marketing, and they routinely work with spreadsheets and web-based tools [87]. This aligns with classic end-user development ideas, where tools are expected to be adaptable by users themselves [88]. The main goal for targeting citizen developers is to empower them to create solutions that reduce manual work, streamline their work processes, and prototype software solutions quickly [89]. Although low-code platforms can be beneficial, it is important to acknowledge the widespread lack of formal training in software engineering, which poses a trade-off between the expressivity and flexibility of the platform and the complexity that citizen developers must handle to use them effectively [90].

Several usage scenarios of low-code platforms have been identified. In particular, low-code platforms are common solutions for data entry tasks, creating dashboards, and managing commercial lifecycles, including state-based workflows such as those using the Business Process Model and Notation (BPMN) [91]. Moreover, low-code platforms are used to provide visual applications for existing Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) systems, as well as for legacy systems [92]. All of these scenarios vary in terms of complexity, number of elements, and requirements, motivating the need for scalable, battle-tested artifact management that model-driven approaches can provide [31].

2.3 MDSE and low-code: synergies and challenges

Model-Driven Software Engineering (MDSE) and low-code development are two practices with notable commonalities and differences. On the one hand, both approaches converge in aiming to improve software development by means of raising abstraction and hiding implementation-level details. MDSE achieves this through meta-modeling, DSLs, and model transformations, whereas low-code platforms rely on visual modeling environments and reusable components that avoid hand-coding to do so. On the other hand, there are notable examples of MDSE and low-code solutions that exclusively belong to one of these paradigms. For instance, most MDSE frameworks do not provide cloud-based solutions or flexible deployment mechanisms. Similarly, most low-code platforms do not have a model-based foundation [31].

Synergies MDSE and low-code can establish a symbiotic relationship that leverages the strengths of both paradigms. First, low-code platforms can broaden their applicability to several domains thanks to the genericity that MDSE brings through meta-modeling. In particular, MDSE can bring tools, practices, and standards to develop DSLs that can be used to make low-code platforms domain-agnostic. Second, MDSE tooling has been mainly developed for desktop-based environments, whereas low-code platforms are typically cloud-based and web-first. By bringing these MDSE capabilities into low-code environments, platforms can exploit advances in cloud computing to scale these operations. Third, low-code platforms often lead to solutions that exhibit vendor lock-in. That is, they only work within the platform where they have been developed. MDSE can help mitigate this issue by relying on open standards and battle-tested formats for model representation and persistence, by unifying meta-models, and by employing model transformations for migration. Finally, MDSE has extensive literature on model and meta-model co-evolution, model versioning, and flexible modeling. These practices and knowledge can be transferred to low-code platforms to enhance their capabilities.

Challenges Integrating these two paradigms is not straightforward. First, low-code platforms tend to evolve rapidly, whereas MDSE tools and practices have a longer history and change more slowly. Second, low-code platforms have particularities, such as managing users, roles, and files, that are not commonly addressed as primitives within desktop-based MDSE environments. Third, model persistence must be implemented in the cloud, which is not common in desktop-based MDSE tools. Finally, many MDSE tools are designed for professional software engineers, who are at the other end of the spectrum from citizen developers.

This thesis explores the synergies and challenges of combining MDSE and low-code by proposing the creation of low-code engineering platforms (LCEPs) [32] in its contributions. First, Dandelion makes use of MDSE to underpin the construction of graphical DSLs, as well as for providing flexible modeling mechanisms. Additionally, Dandelion makes use of cloud-based persistence to provide scalable storage and management for large models. Second, SMSs leverage MDSE for creating domain and level-agnostic purposeful visualizations that integrate out-of-the-box with LCEPs. Third, Dandelion+ extends Dandelion's persistence layer and improves it with support for all low-code-relevant artifacts (e.g., user management, roles, files), as well as model-driven workflows for specifying application behavior, making use of stable MDSE engines (i.e., Eclipse Epsilon languages). Finally, LowcoBot exploits the model-driven foundation of LCEPs to generate chatbots that operate on the platforms' concepts.

2.4 Summary and conclusion

This chapter has introduced the main concepts that underpin this thesis. On the MDSE side, it has reviewed models and meta-models, multi-level and flexible modeling, model transformations, and model management tooling, framing them as mechanisms to raise the level of abstraction and automate development. On the low-code side, it has characterized low-code platforms as visual, component-based environments, often web-based and aimed at citizen developers, and has outlined their key features and usage scenarios. Synthesizing these views, this chapter argues that MDSE and low-code share core principles but differ in target stakeholders, tooling, and trade-offs, and has introduced low-code engineering platforms (LCEPs) as a way to bridge both approaches. This perspective grounds the contributions presented in this thesis: Dandelion, model sensemaking strategies (SMSs), Dandelion+, and LowcoBot.

The next chapter builds on this foundation by surveying related work in these areas in more detail.

Chapter 3

Related Work

This chapter reviews the state of the art related to the contributions presented in Part II. First, it presents works about graphical editors in the cloud, model scalability, and flexible modeling (3.1), necessary for Chapter 4. Second, it discusses existing approaches for understanding models through purposeful visualizations, covering visualization of large artifacts as well as genericity and pattern-based reuse (3.2), which are relevant for Chapter 5. Third, it reviews commercial and academic low-code platforms, together with workflows to create model-driven low-code platforms (3.3), as done in Chapter 6. Finally, it examines works on the intersection between chatbots and modeling (3.4), which are related to Chapter 7.

3.1 Definition of graphical language workbenches in the cloud

This section reviews existing works on frameworks for building cloud-based graphical language editors (3.1.1), scalability in modeling (3.1.2), and flexible modeling (3.1.3).

3.1.1 Workbenches for graphical domain-specific languages

Language workbenches are tools that facilitate the creation, design, and evolution of domain-specific languages (DSLs) by means of high-level specifications. More concretely, graphical language workbenches support the development of graphical syntaxes for these DSLs. These workbenches have evolved considerably over the last three decades, moving from desktop-based to web- and cloud-based environments. This subsection reviews this evolution with a focus on language definition, editor deployment (desktop versus web), and integration with surrounding modeling infrastructures.

Early desktop graphical language workbenches In the 1990s, the intensive use of (often graphical) modeling notations in MDSE drove the need for techniques for automated generation of graphical modeling editors. These were often defined using a model-driven approach, where the languages' abstract syntax and their visualizations were defined via models. Early proponents of this approach included tools like MetaEdit [93] (from 1991), Multigraph [94] (from 1995), Honeywell's DOME [95] (from 2000), or AToM^3 [96] (from 2001). The initial selling points of these tools were methodology modeling [93], model-integrated computing [94], rapid evolution of domain-specific tools [95], and multi-formalism modeling [96]. All these approaches relied on desktop computing and established the now common pattern of specifying both abstract and concrete syntaxes through models. In addition, they were typically distributed as heavy desktop applications, treating models as local files and relying on external version control systems (e.g., Git) for collaboration, which made real-time multi-user modeling difficult.

Eclipse/EMF-based desktop language workbenches A second wave of frameworks for graphical editors emerged with the rising popularity of Eclipse and EMF in the mid-2000s and the 2010s. Approaches like Tiger [97], GMF [98], Sirius [60], Eugenia [99], or Graphiti [73] enabled the definition of graphical editors, which could be integrated as plugins with other tools and become more interoperable by the use of EMF (an implementation of the OMG's Metaobject Facility [54]). These workbenches consolidated EMF as the *de facto* meta-modeling platform and primarily targeted professional developers working within the Eclipse desktop ecosystem.

Web-based language workbenches More recently, advances in networking, cloud computing, and the popularization of low-code platforms are motivating the development of graphical editors accessible from web browsers. In this context, several web-based editors have been proposed, including web-based versions of desktop frameworks (WebGME [100], AToMPM [101]), developments such as Pyro [102], and Eclipse-based frameworks such as Sirius Web [103] and EMF.cloud [104]. Many of these platforms adopt a centralized, database-backed architecture that exposes modeling-as-a-service via web APIs, with WebGME in particular offering integrated, Git-like version control and branch/merge support for models, and AToMPM providing a fully web-native, multi-user modeling environment. A more recent proposal is Jjodel [105], a cloud-based reflective language workbench that offers a low-code web environment with modular viewpoints for abstract and concrete syntax, validation, and semantics, together with flexible modeling and co-evolution support aimed in particular at educational and lightweight industrial scenarios.

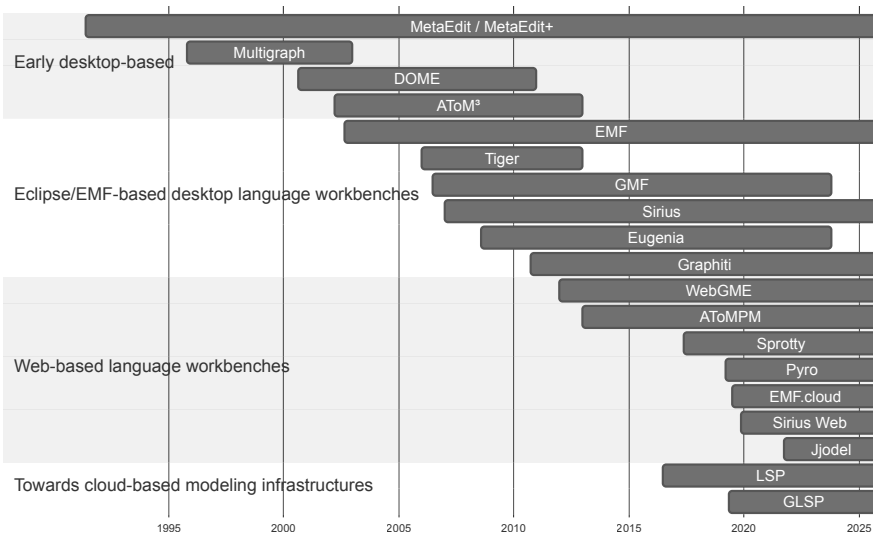


Figure 3.1
Timeline of the main graphical modeling frameworks.

Towards cloud-based modeling infrastructures In the late 2010s, the proliferation of textual languages and IDEs motivated the standardization of communication between them. To date, the Language Server Protocol (LSP; from 2016) [106] is widely adopted, and the most commonly used languages and IDEs feature it. A similar need arose for graphical languages [107], leading to the Graphical Language Server Platform (GLSP) [107] by the Eclipse Foundation. GLSP is still a work in progress, though it is finding proponents in Eclipse frameworks such as Sprotty [108], used by EMF.cloud [104] components. Such tools mark a shift towards web-first modeling infrastructures, exploiting advances in cloud computing.

The evolution of these tools is illustrated in Figure 3.1, which presents a timeline of the main graphical modeling frameworks and their (approximate) periods of activity. The new wave of web- and cloud-first frameworks, emerging mainly in the late 2010s and early 2020s (including, for example, Sirius Web, EMF.cloud, and Jjodel), marks an inflection point and indicates that most of these tools are still in an active phase of development.

Even though several of these proposals support zero-installation use via a web browser (e.g., WebGME, AToMPM, and Pyro), they still lack essential features for DSL editors to be effective in low-code development. First, they provide limited support for handling large models and for exposing configurable scalability mechanisms that can be adapted to the DSL and its models. Second, interoperability with multiple modeling technologies is often restricted. Finally, support for tolerating inconsistencies in the modeling process through a customizable conformance relation remains limited.

3.1.2 Model scalability

Handling large models efficiently has been a recurrent scalability challenge in MDSE. Traditionally, the EMF ecosystem has relied on file-based persistence for models, often in a monolithic way. However, this approach becomes problematic when models grow large.

Several solutions have been proposed to address this issue. In [109], the authors propose fragmenting model files by defining fragmentation strategies at the meta-model level, which results in smaller files that can be loaded and processed faster. A similar approach is followed in [110] for storing models in multiple files. In [111], the authors propose a method for partial loading of EMF models. To facilitate model management, some authors decompose complex models into smaller sub-models conformant to the same meta-model [112]. For faster access to model elements, some authors have also proposed model indexers [113], similar to those in relational databases. Finally, caching techniques for large models and for queries over them have been proposed in [114], [115].

Some approaches for EMF, such as CDO [116] and Teneo [117], provide data persistence mechanisms based on relational databases. Other approaches, like Morsa [118] or NeoEMF [119], use non-relational databases to persist and query very large models.

Traditional MDSE settings based on EMF typically involve expert engineers operating in a desktop IDE, where models are treated like regular software artifacts and are thus persisted as files. These are amenable to asynchronous collaboration, e.g., via version control systems [120]. LCDPs, in contrast, target citizen developers with low technical expertise and strive for multi-user —possibly synchronous— collaboration. In particular, models serve as the foundation for low-code platforms built using MDSE (i.e., low-code engineering platforms), with the difference that every model interaction takes place within a browser. As a result, model persistence is transparent to end users, and platform providers have comparatively more flexibility in how model persistence is implemented.

The goal in this regard is to devise solutions that are *cloud-based*, *industrial*, and *low-code* (i.e., suitable for citizen developers). The proposals above effectively attain some of these goals but not all simultaneously. In many cases, they achieve improvements by circumventing some of the issues present in the modeling infrastructures on which they are built. For example, many EMF tools rely on file-based persistence, which is unsuitable for cloud-based industrial solutions, as such formats lack native model-specific caching mechanisms and multi-user handling. Additionally, XMI, the *de facto* format for model persistence, has a fixed language granularity, thereby hampering scalability. Despite being a legacy format, it is deeply ingrained in MDSE tooling. Solutions such as indexers [113] or partial loading [111]

are effective given the technology lock-in of Eclipse-based solutions, but do not solve the problem at its root. Outside Eclipse, MDEForge [121], [122] is a cloud-based repository of modeling artifacts that employs Lucene to create indices for the repository content and enable an efficient text search over the artifacts. While MDEForge exposes web services for storing and managing modeling artifacts, it does not provide any specific scalability mechanism for models.

However, while all the previous works alleviate some of the scalability problems of modeling tools, they only target the abstract syntax of models and neglect their concrete syntax and visualization aspects. Moreover, they do not provide concrete facilities for model comprehension tasks.

In contrast, full cloud-based industrial low-code solutions can approach scalability issues differently. For example, offloading model computation to a cloud-based environment can eliminate the dependence on the computational resources of the user's device. Moreover, these solutions can use highly scalable services (i.e., able to self-regulate their storage and computing capabilities), such as cloud databases to query and persist industrial-scale models efficiently, eliminating the need for file-based persistence. Additionally, thanks to their cloud nature, these solutions can be easily integrated with other low-code or MDSE tools as long as they are exposed as web services to support tasks such as model-to-model transformations, graph summarization, or model analytics.

In this thesis, Dandelion and Dandelion+ instantiate this approach. They rely on harmonizing meta-models that represent models from diverse technologies, using Elasticsearch [123] (a flexible, highly-scalable search engine based on Apache Lucene [124]) for persistence. Moreover, Dandelion integrates with several web services to support model layouts, recommendation services, and external technology injectors.

3.1.3 Flexible modeling

Many researchers argue that tolerating some degree of inconsistency can benefit various software engineering scenarios. For example, in creative design thinking it can help mitigate premature commitment to design decisions and promote the involvement of all stakeholders [125]. Most modeling tools, however, enforce strict consistency at all times. As discussed in the following, only a few proposals support what is often termed flexible modeling, where the conformance relation can be relaxed. This flexibility is particularly relevant in low-code settings, where citizen developers may not be used to rigid modeling processes.

FlexiMeta [126] relaxes conformance to permit the creation of models and meta-models in any order. It allows prototyping objects with no class and deriving a meta-model from the objects —a form of bottom-up meta-

modeling— to enable simple checks (e.g., multiplicity violations and missing or spare attributes). It implements an implicit modeling process with three phases that can be selected manually: exploration, with no meta-model; consolidation, with an inconsistency-tolerant meta-model; and finalization, with full conformance to the meta-model.

The JavaScript Modelling Framework (JSMF) [127] allows disabling the cardinality and type checks of fields on a per-class basis. JSMF is a JavaScript-embedded domain-specific language. Therefore, models and their conformance relations must be configured using JavaScript.

The modeling tool FME [128] supports the same modeling scenarios as JSMF, but the conformance relation is not configurable. The authors propose a manual reconciliation process that modifies the meta-model and the model to make them fully conformant.

Kite [66] is an Eclipse plugin for flexible meta-modeling. It supports objects with zero or multiple types, multi-level models, a fine-grained customization of the conformance rules, and the definition of modeling processes whose phases can have different conformance levels.

More recently, Jjodel [105] has been proposed as a cloud-based reflective language workbench with out-of-the-box support for flexible modeling. It lets users create untyped objects with *ad hoc* properties, which may later be typed as instances of an existing or new meta-class. When this happens, matching properties are bound to the meta-class features and the remaining ones are kept as unbound data. In addition, Jjodel implements runtime co-evolution heuristics that adapt stored values to meta-model evolution on access (e.g., casts and multiplicity enforcement) while preserving the original user input whenever possible, so typing decisions are non-destructive.

Informal diagrams and relaxed conformance CouchEdit [129] permits the creation of unrestricted diagrams, which are internally translated into a meta-model instance to establish their conformance as far as possible. Similarly, the approach in [130] proposes migrating selected parts of informal architecture diagrams into formal textual models for analysis tasks. Thus, both proposals implement a relaxed conformance mechanism that is not designed to be customized or partially disabled by the end user.

Bottom-up and example-based meta-modeling Other flexible modeling tools permit deriving a suitable meta-model from a set of untyped models (so-called bottom-up or example-based meta-modeling). Examples of such tools include FlexiSketch [131], FME [128], metaBup [132], and Muddles [133]. This idea has also been applied to UML in [134], where the UML notation is extended with symbols representing incomplete, inconsistent, or incorrect objects, and a process derives a —likely imperfect— UML class diagram

Tool	Configurable conformance	Bottom-up	Concrete syntax
FlexiMeta [126]	○	•	Textual
JSMF [127]	•	○	Textual
FME [128]	○	•	Textual
Kite [66]	•	○	Textual
Jjodel [105]	○	◒	Graphical
CouchEdit [129]	○	○	Textual
Blended models [130]	○	○	Textual
FlexiSketch [131]	○	•	Textual
metaBup [132]	○	•	Textual
Muddles [133]	○	•	Textual
UML bottom-up [134]	○	•	Textual
Multi-phase modeling [135]	○	○	Textual
Dandelion *(this thesis)* [37]	•	◒	Graphical

LEGEND: • supported, ◒ partial support, ○ not supported.

Table 3.1
Summary of flexible modeling tools and their support for configurable conformance and bottom-up meta-modeling, along with the concrete syntax used for modeling.

from an object diagram so annotated. However, none of these tools supports configurable conformance relations.

Process-based flexible modeling Finally, some tools enable defining domain-specific modeling processes encompassing phases with different restrictions. For example, Kite, the tool mentioned above, allows configuring the type checks to carry out in each phase, and multi-phase modeling [135] uses concepts of multi-level modeling to establish the order of the phases in the modeling process and to control the object types that can be used in each phase.

Altogether, the literature reports several tools that bring flexibility to modeling. However, except for JSMF and Kite, they provide fixed, non-configurable conformance relations. JSMF and Kite are textual frameworks aimed at language engineers and require programming, whereas Jjodel realizes flexibility through built-in heuristics for untyped objects and co-evolution rather than an explicitly configurable conformance palette. To the best of our knowledge, Dandelion is the first low-code graphical modeling tool supporting a configurable, flexible conformance relation suitable for citizen developers. In addition to configurable conformance, Dandelion also offers partial bottom-up meta-modeling support through quick fixes. Table 3.1 condenses the main features of the flexible modeling tools discussed above.

3.2 Purposeful visualizations for understanding models

Graphical modeling notations often lack visual scalability. In particular, the amount of information that can be presented is constrained due to human comprehension limits (large models are difficult to grasp [57]) and technology limitations (large models do not fit on the screen and take a long time to display [136]). Hence, techniques have been proposed to visualize, explore, and comprehend large models. To this end, this section reviews work on methods for visualizing and comprehending large artifacts (3.2.1) and works on pattern-based reuse (3.2.2).

3.2.1 Visualizing and comprehending large graphs and models

Most techniques to visualize, explore, and comprehend large models come from the field of graph visualization. Large structured datasets are often represented as graphs, and numerous efforts target effective large-graph visualizations [137], [138]. Many of these efforts have been triggered by the field of Visual Analytics, which employs analytical reasoning facilitated by interactive visual interfaces [139], [140].

For example, [141] surveys techniques for visualizing graph structures, including node coloring, contour drawing, adjacency matrices, and their embedding into graphs. In [142], techniques for visualizing dynamic graphs are studied so that interaction techniques do not produce abrupt, disruptive changes in the user experience. FACETS proposes the exploration of large graphs driven by the most interesting neighborhood of the current node [143].

Related to visual analytics, Pienta et al. [144] propose graph sensemaking as "*the iterative process of understanding and making sense out of graph-formatted data, where a user gradually builds up a representation of the information space to achieve the user's goal.*" They also provide a survey of the graph exploration and visualization literature and create a taxonomy of graph sensemaking techniques [145].

This thesis takes inspiration from this line of work, adapting the idea to the *modelware* technical space and making model sensemaking strategies (SMSs) reusable. Plain graphs have no explicit semantics —nodes and edges have no types or properties— which enables aggressive simplifications. However, when applying SMSs to models, it is worth considering the structure and semantics provided by their meta-models, the possibility of applying SMSs to meta-models themselves, and the analysis of modeling ecosystems.

The need to understand and monitor large code repositories has triggered the appearance of tools [146] offering dashboards to visualize aspects of the construction process (e.g., commits) and the code itself (e.g., LoC). Similarly, SMSs can be grouped into dashboards and are adaptable to different DSLs.

Approaches	Artifacts	Visualizations	Meta-modeling	Reusability
Large-graph visualization [137], [138]	Large graphs	Layouts (node-link, matrices), clustering, multi-level views	○	●
Visual analytics and graph sensemaking [139], [140], [144], [145]	Analytic data, large graphs	Interactive visual analytics; sensemaking tasks and workflows	○	●
Adaptive local graph exploration [143], [156]	Attributed graphs, query results	Via 'interestingness' factor and query result graphs	○	●
Software analytics dashboards [146]	Software repositories	Dashboards, metrics, time-series, drill-down	○	◒
Semantic zoom, focus, and context for modeling [147], [148], [149], [150]	UML diagrams, OWL ontologies, large graphs	Semantic zoom, focus + context, details-on-demand	●	◒
Layers and label management [151], [152], [153]	UML diagrams	Graphical layers, automatic layout, compaction, label control	●	◒
Language-independent abstractions and scalable exploration [136], [154]	Large EMF or DSL models	Structural abstractions, scalable model navigation	●	●
Model views [155]	Models	Views (queries and transformations)	●	◒
Flexible concrete syntax / visualization DSLs [105], [157]	DSL models, data visualizations	Reactive, template-based concrete syntaxes (Jjodel); DSL-specified visualizations (VizDSL)	●	●
Model sensemaking strategies (*this thesis*) [34]	Models, meta-models, modeling ecosystems	Graphs, dashboards	●	●

LEGEND: • supported, ◒ partial support, ○ not supported.

Table 3.2 Comparison of model visualization and comprehension approaches.

There is little literature on applications for the visualization and understanding of large models. For instance, visualization techniques based on semantic zoom [147] have been applied to UML diagrams [148], [149] and OWL ontologies [150] to improve their understanding. Layering has been used for UML diagrams [151]. Similarly, in [152], label management [153] and vertical message compaction are applied to reduce the height of sequence diagrams and make them fit on a screen. While useful, all these works target specific languages. In contrast, language-independent catalogs of model abstractions have been proposed in [154] to simplify models for better comprehension, and in [136] to facilitate the exploration and processing of large models. Still, these abstractions do not consider the models' concrete syntax. Model views [155] can be defined to extract a relevant portion of the model via a query language. However, the result is still a model, while SMSs can benefit from different visualization metaphors.

More recently, a taxonomy of advanced visualization techniques for

conceptual modeling was proposed in [158] based on an analysis of 46 tools such as IntelliJ IDEA, Google Maps, and PowerPoint. In most of them, visualizations were built *ad hoc* and were not reusable. Finally, a few modeling tools have been designed to permit a flexible definition of concrete syntaxes. This is the case of Jjodel [105], a web-based modeling environment supporting the definition of views reactive to variations on the model objects, in the style of Visual Basic. VizDSL is a DSL to define interactive data visualizations, which can be applied to render models [157] (i.e., define their concrete syntax), but does not target or directly support SMSs. This thesis, in contrast, proposes an approach that focuses on strategies that are defined once and can be reused across different DSLs. Table 3.2 summarizes the features of these approaches.

3.2.2 Genericity and pattern-based reuse

Reusability and genericity are powerful mechanisms in Software Engineering to reduce complexity. In this thesis, SMSs are designed to be domain-agnostic and generic, so that they can be reused across different DSLs and low-code platforms without redefining them for each meta-model. To this end, SMSs expose a meta-model pattern, which can then be bound to target meta-models via a binding.

This is a form of generic approach that other researchers have also exploited to define generic refactorings [159], generic model abstractions [154], generic model fragmentation strategies [35], or generic model transformations [160], [161]. Several tools, such as Kermeta [162] or MetaDepth [163], support the definition of generic operations. All these approaches have in common that the generic operation exposes a small meta-model that has to be bound to concrete meta-models to make the operation reusable.

First, SMSs can be bound to the linguistic meta-model to define agnostic strategies, applicable to any (meta-)model. Second, the proposed binding language offers an expression mechanism that enables gathering information from lower meta-levels. Third, this thesis provides a recommender that automatically precomputes feasible bindings, easing the reuse of SMSs across different DSLs and modeling ecosystems.

More generally, pattern-based approaches have been used in model transformations [164], DSLs and associated services [165], and domain-specific frameworks such as secure systems development [166] or dependability engineering [167]. To the best of our knowledge, this thesis is the first to apply patterns to model sensemaking. This pattern-based design enables SMSs to be defined once and then reused across the different DSLs supported by Dandelion and by the low-code platforms modeled with Dandelion+, supporting the thesis goal of scalable, language-independent model comprehension.

3.3 Modeling low-code platforms: structure & behavior

This section examines works related to low-code platforms (3.3.1) and workflow languages (3.3.2). Then, it compares the features and limitations of these works with those of Dandelion+ in a gap analysis (3.3.3).

3.3.1 Low-code platforms

Low-code platforms are automation solutions built on top of domain-specific languages (DSLs) [85]. As discussed in Section 3.1.1, language workbenches such as MetaEdit [23], AToM^3 [96], and EMF-based tools like Sirius [60], Xtext [59], or GEMOC Studio [168] already support the definition of DSLs and their editors, mostly targeting expert developers in desktop environments. Industrial low-code development platforms (LCDPs) can be seen as their cloud-based successors: they integrate DSLs into web-based environments aimed at citizen developers, typically offered as platform-as-a-service (PaaS) solutions [15], [31], where traditional coding is replaced by drag-and-drop composition of premade components and configuration via forms or expression languages.

Notable LCDPs include OutSystems [61], Mendix [169], Zoho Creator [170], Microsoft Power Apps [171], Google AppSheet [172],[1] Kissflow [173], Salesforce Platform [174],[2] Appian [175], and Caspio [176]. The next paragraphs discuss the salient features of these platforms.

OutSystems [61] is one of the most prominent low-code platforms. Application development takes place in Service Studio, its dedicated desktop IDE. Each application is made of modules which, in turn, are divided into interface, logic, and data. The interface editor features many visual widgets, including graphs, forms, and lists, which can be arranged in a drag-and-drop graphical editor. The structure of entities is managed through databases. Regarding logic, OutSystems differentiates between client and server actions. The former are used to manipulate the UI and can be triggered by UI events, while the latter may perform small backend computations.

OutSystems can expose server actions as services using SOAP, REST, or SAP. After every development iteration, the final application must be generated, compiled, and deployed to the cloud using a predefined and closed-source generation process. The result is a cross-platform application, covering web and mobile. The strength of OutSystems lies in its rich set of visual components, its numerous integrations with third parties, and its ease of use.

Mendix [169] is another key proponent of low-code development. Its IDE is called *Mendix Studio*, and it is desktop-based. In Mendix, applications

[1] Google App Maker closed in 2021.

[2] Previously, it was called Salesforce App Cloud.

(or 'modules') contain domain models, security configuration, images, pages, and (micro-)flows. Mendix follows a closer approach to modeling, as the domain model manages entities and relationships unlike OutSystems which relies on database schemas. For behavior, Mendix uses microflows, which are built using a drag-and-drop editor and manage objects, lists, variables, and client activities. Mendix also supports integration with external services like SAP and allows data manipulation via REST services. As with OutSystems, applications are built before they are deployed to the cloud.

While OutSystems and Mendix focus on creating interactive web applications (typically enterprise applications), Zoho Creator [170] covers the niche of data-intensive applications. It offers over 500 integrations with services like Salesforce, GitHub, and Stripe, which can be invoked through workflows triggered by CRON jobs or events. The platform uses the proprietary scripting language Deluge[3] to standardize the integration with third-party services in the workflows. The salient feature of the platform is the construction of dashboards. The development is agile, as dashboard construction is guided by a recommender system that exploits the definition of the application's domain. The visual representation is also highly customizable thanks to the Zoho Markup Language (ZML),[4] a proprietary, XML-based language for describing the visual structure of widgets within dashboards.

The proposal from Microsoft is Power Apps [171], a low-code platform deeply integrated within the company's ecosystem, allowing the development of applications connected with services like Power BI, Power Automate, or Microsoft Dataverse (i.e., Microsoft's centralized data storage). The platform relies on Power Fx, a formula-based language similar to Excel's, for application behavior. The language is declarative, and it allows a reactive definition of applications' behavior. A differentiating trait of Power Apps is that it features both robotic process automation (RPA) and digital process automation (DPA): RPA automates repetitive tasks by replicating user actions, while DPA supports a traditional approach of workflows for backend development. Following the same approach as Zoho Creator with ZML, the structure of the visual widgets can be customized using Adaptive Cards,[5] a JSON-based language with templating facilities.

While many low-code platforms aim to produce highly customizable solutions, the scope of Google AppSheet [172] is different: to bring users' spreadsheets (e.g., Google Sheets) to life as dashboards. Given a spreadsheet, the platform derives a data schema from the data it contains. This way, every worksheet maps to the definition of an entity; columns within the worksheet define fields of the entity; and every row becomes an instance. The look

[3] https://www.zoho.com/deluge

[4] https://help.zoho.com/portal/en/kb/creator/developer-guide/zml-guide/articles/zml-an-overview

[5] https://adaptivecards.io

and feel of the entities can be customized through a collection of predefined widgets, including calendars, tables, galleries, and forms. Moreover, besides Google's proprietary artifacts, AppSheet integrates with third-party vendors like AWS, Azure SQL, or Salesforce. On a related note, using spreadsheets as model artifacts has also been explored by model-centric solutions like Epsilon [41], which offers a driver for spreadsheets [177] with a similar mapping approach.

Kissflow [173] is a web-based low-code platform specializing in applications that track entity lifecycles with business processes. Kissflow features a workflow language that integrates with enterprise tools like SAP and Salesforce. Moreover, it offers an AI copilot to assist during the development of workflows. The platform is highly customizable, allowing users to define roles, forms, pages, navigation, analytics, and integrations. The platform also features a proprietary declarative expression language to define application behavior.

Salesforce is a customer relationship management (CRM) SaaS to manage the full lifecycle of customer interactions, e.g., from receiving an order to its processing and delivery. Its selling point is automating processes to avoid manual errors. Its low-code platform is Salesforce Platform [174], which builds atop Salesforce's CRM capabilities. The platform allows linking pages to data types and configuring permissions and applications. Pages defined within the tool may include dashboards that display data in tables and charts.

Appian [175] is a web-based low-code platform. It allows managing record types and process models that integrate data from databases, Salesforce, or web services. The platform features widgets to manage entities, as well as workflows for business automation. The platform is flexible with security, allowing fine-grained role and data access management.

Many low-code platforms rely on graph-like workflows to manage behavior. Caspio [176] proposes a different approach, whereby workflows are defined using the MIT App Inventor.[6] In particular, workflow nodes are SQL-based, providing a Scratch-like interface for building applications[7]. The platform supports building reports, forms, and charts directly from SQL tables.

Unlike other low-code solutions, UGROUND aims to produce Semantic Digital Twins [178]. These are developed following an approach based on a self-referential, ontology-driven architecture, enabling continuous adaptation without the need for bespoke software redevelopment [40], i.e., so-called no-code. The company covers several domains, from banking to insurance and human resources. The development of these platforms involves ROSE (Recursive Ontology Semantic Engineering) [40], both a methodology and a

[6] https://appinventor.mit.edu

[7] https://scratch.mit.edu

model-driven engine. In ROSE, applications are structured through semantic modeling, while JCORE, a dedicated general-purpose language, defines behavioral logic. The company has developed a comprehensive catalog of reusable functionalities and visual components, continuously expanding its internal JCORE library. Thanks to the reliance on a model-driven architecture, UGROUND can reuse many components across domains and applications, accelerating the development process.

Finally, there are few academic results on the intersection between MDSE and full-fledged low-code platforms. One example is BESSER [179], an MDSE-based low-code platform that relies on UML for structural modeling and on template-based code generators for execution. A complementary contribution is RhoArchitecture, a JSON-based DSL approach for web applications, which combines an evaluation engine, a DSL programming model, and an integrated web development environment [180], [181]. BESSER is still in development and uses a standard general-purpose domain modeling language (like UML class diagrams), which is not the optimal choice to describe low-code applications. Instead, such a language needs to consider primitives specific to low-code, such as platforms, users, roles, and resources. Similarly, code generation alone significantly limits the production of low-code applications. These applications would generally benefit from having other services available, including model transformations and powerful model visualizations [34], which can be orchestrated and executed within the defined low-code application itself.

3.3.2 Workflows

Workflow languages allow describing processes in terms of resources, tasks, and decision blocks [182]. Workflows are often represented with diagrams composed of nodes and connectors to facilitate their comprehension. They can be used for a variety of purposes and domains, from machine learning and data processing to business processes and protocols. This section reviews four broad workflow categories according to their purpose: process modeling, MDSE development, MDSE in the cloud, and low-code development.

Workflows for process modeling Several workflow languages have been proposed for modeling processes. The most notable example is the Business Process Model and Notation (BPMN) [91], a standard to specify business artifact behavior. Another example is the Software & Systems Process Engineering Metamodel (SPEM) [183], which relates processes, roles, and work products in an ordered manner. To further facilitate the definition of business processes, works like [184] define workflow patterns, which, in the style of '*The Gang of Four*' [185], correspond to recurrent structures that emerge in workflows, and may consider diverse aspects of the process such

as control flow, data flow, or resource management.

Workflows for MDSE development MDSE has typically taken place in dedicated editors (IDEs) for programming with facilities for model management. As such, these environments have stimulated the development of workflow languages. Some languages focus on very specific use cases, such as transformation chains [160], [186]. Others allow for general model management tasks, such as ModelFlow [187], [188], the workflow language within AToMPM [189]; or the Modeling Workflow Engine (MWE) [190], for Eclipse modeling components. In some cases, the traceability of the transformations in workflows is preserved for a better understanding and for enabling validation [191].

Workflows for MDSE in the cloud Some workflow languages have been designed to perform MDSE activities in the cloud, while not being integrated within low-code environments. For example, MDEForgeWL [192] is a workflow language to orchestrate model management services, and MDEForge-Search [193] includes a query language to retrieve MDSE artifacts from a distributed cloud-based model repository.

Workflows for low-code development Most of the low-code platforms overviewed in Section 3.3.1 use workflows in different ways. For instance, workflows in Mendix feature loops, decisions, and recursive actions, in the same way as OutSystems. OutSystems, in particular, distinguishes between client and server actions. In Zoho, the focus is on event-driven workflows with basic logic (e.g., decisions or delays). Microsoft Power Apps integrates with Power Automate, also from Microsoft, providing robotic process automation (RPA) in addition to traditional digital process automation (DPA). Many LCDPs make use of expression languages, either declarative (e.g., Power Fx in Power Apps or Google AppSheet) or imperative (e.g., Deluge in Zoho Creator).

3.3.3 Gap analysis

Existing work sits at the intersection of low-code development, MDSE, and workflows, but does not fully cover their combination in a cloud-based, model-driven low-code platform. A key question for this thesis is whether techniques from these paradigms can be combined to overcome the limitations of current low-code platforms.[8]

[8]Note that the goals of this thesis do *not* include building full-blown professional LCDPs. Instead, the scope is limited to the investigation of bringing MDSE techniques to the low-code world, and assessing their benefits.

MDSE stands out for its flexibility in defining model-based solutions for specific domains. However, MDSE workflows are typically tied to concrete IDEs, which hinders their use in other environments such as the cloud. MDSE workflows deployed in the cloud are typically ports of other domain-specific workflows, and are thus still not appropriate for low-code development, since they lack primitives and integration with low-code platforms. This is also a limitation of generic workflow languages for process modeling. As a result, to the best of our knowledge, no workflow language enables low-code behavior development in the cloud while following a model-driven approach. Dandelion+ —the approach introduced in this thesis— addresses this need with PLATFLOW.

Regarding low-code platforms, Table 3.3 compares current LCDPs with Dandelion+, in terms of the offered capabilities for the following aspects of applications: model-driven features, workflows to define application behavior, access control rules, and external integrations.

Most LCDPs in Table 3.3 lack MDSE features. In particular, almost no LCDP supports the definition of custom application generators. The only exceptions are Power Apps and Mendix, which focus on PDF/HTML document generation, rather than full code generators; BESSER, which lacks support for key low-code aspects like users or roles; and Dandelion+ (introduced later in this thesis), which offers tech-agnostic code generation with EGL/EGX. Thus, LCDPs generally rely on inflexible, predefined code generators, leaving citizen developers with few options for customizing their applications for specific domains. LCDPs rely on built-in application generators that produce similar applications, with little room for further customization.

Regarding workflows, LCDPs typically allow the encoding of behavior using dedicated nodes, such as for entity management (CRUD operations) and connecting to third-party services. Workflow processes may also call other workflows. Access control is often specified through expression languages or forms, allowing permissions and roles to be defined for both citizen developers (who can access the development environment to modify the application) and for end users of the runtime applications. Most LCDPs also support integration with external services, though the scope of integrations varies between platforms.

Overall, Table 3.3 shows that all LCDPs (with the exception of BESSER) provide mechanisms to specify and execute workflows, to define access control policies, and to integrate external services within the LCDP operation. Even if the expressiveness of the workflow primitives, the granularity and customizability of the access control, and the integrability possibilities largely vary among LCDPs, it is clear that these three aspects are key to any successful LCDP today. In this respect, as discussed at the beginning of Section 3.3.1, MDSE solutions typically run on desktop computers and

Platform	*Model-driven features*			Workflows	Access Control	Integrations
	Custom code gen.	**Meta-modeling**	**Model transf.**			
OutSystems [61]	○	◒ relational DB schema with visual editor	◒ to adapt data to populate fields of widgets	Divided into server- and client-side; supports actions like message, compute data, refresh data, conditionals, loops, etc.	Application, data, and screen levels; fine-grained control	SOAP, REST, SAP
Mendix [169]	◒ PDF, HTML	◒ object-oriented schema with visual editor	○	Microflows for object activities, REST services, Java actions, ML activities, decisions, and merges.	CRUD permissions for types with XPath expression constraints	REST, SAP, and other external services
Zoho Creator [170]	○	○	○	Workflow nodes include apps (integrations), logic (IF, decision, set variables), and integrations.	Filter pages by role, visibility, and read-only status	500+ integrations (Salesforce, GitHub, Zoho CRM, Slack)
Power Apps [171]	◒ PDF, HTML	◒ relational DB schema	◒ to select (custom) columns of a data entity	RPA and DPA for automating user interactions and backend workflows; executes workflows on events and schedules.	Ownership models with permissions (full, reference, collaborate, private, etc.)	Power BI, Power Automate, AI Builder, Microsoft Dataverse
AppSheet [172]	○	◒ relational DB schema	○	Bots define workflows; simple tasks like email, notifications, and webhooks.	Role-based filters on fields and attributes with expression language	Google Sheets, Office 365, Salesforce, Cloud SQL, Dropbox
Kissflow [173]	○	○	○	Business process modeling using drag-and-drop workflows with forms and integrations.	CRUD control per type, form roles for manage/read-only/edit	SAP, Salesforce
Salesforce [174]	○	◒ relational DB schema with visual editor	◒ derived fields	Workflow includes decisions, loops, create/update elements, and subflows.	Fine-grained control per page, type, and role	Salesforce ecosystem (Sales Cloud, Marketing Cloud, etc.)
Appian [175]	◒ HTML, PDF, Word	◒ relational DB schema with visual editor	◒ derived fields	Process models with events, tasks, automation, data services, APIs, and document generation.	Complex role and object-based permissions with granular access control	Salesforce, databases, REST APIs
Caspio [176]	○	◒ relational DB schema	○	Scratch-like puzzle-piece workflows using SQL-based nodes (INSERT, UPDATE) with limited automation.	Limited access control through apps	Direct SQL access
ROSE [40]	◒ imperative JCORE code	● multi-level modeling: ROSE with visual editor	◒ imperative JCORE code	Business process models (BPMs), to model the lifecycle of entities; finite state machines.	Managed at the entity level, has roles	SOAP, REST, SAP, DBs (Oracle, Elastic, MongoDB, etc.)
BESSER [179]	● tech-agnostic, with Jinja templates	● full modeling: B-UML with web-based graphical editor	○	—	—	—
Dandelion+	● tech-agnostic, with EGL/EGX	● full modeling: Dandelion format with visual editor	● tech-agnostic, with EOL and ETL	Workflows can be chained and executed within the platform, offering PlatFlow for orchestrating both MDSE and low-code tasks.	Complex, platform-specific access control tied to MDSE models	Third-party APIs, format importers, Eclipse Epsilon drivers

LEGEND: • full support, ◒ partial support, ○ no support.

Table 3.3 Features for application configuration of the leading commercial and academic LCDPs.

do not provide first-class support for these aspects, which hinders their applicability for low-code development.

Hence, this thesis identifies two gaps in the state of the art. First, industrial LCDPs lack MDSE features, especially code generation facilities. On the other hand, academic LCDPs (i.e., BESSER) mostly port MDSE concepts to the cloud. Currently, they lack support for low-code concepts (e.g., platforms, users, or files), integration via APIs, access control, and workflow facilities.

3.4 Conversational agents for low-code platforms

There have been several efforts at the intersection of chatbots, LLMs, modeling, and low-code platforms.

Some works focus on chatbots and modeling. For instance, Socio [194] is a chatbot that produces UML class diagrams from user descriptions of the domain. The chatbot is based on syntactic analysis of user utterances, using the Stanford parser. Later, the advent of LLMs triggered the use of LLMs for modeling tasks. For example, Cámara et al. [11] assess ChatGPT for generating UML models from natural language descriptions. Chaaben et al. [195] use LLMs as assistants for model completion, and Chen et al. [196] evaluate different LLMs for automated domain modeling. Droid [197] takes a complementary path by recommending additional model elements (e.g., attributes or operations) within EMF-based editors rather than generating entire models. Tools like Xatkit [198], Conga [199], and DemaBot [200] follow a model-driven approach to designing chatbots.

Other works leverage LLMs for low-code tools and environments. For example, [201] presents an LLM-based tool for designing AI pipelines within a dedicated low-code environment, and the Eclipse GLSP project is currently exploring the potential of AI for improving graphical editors.[9]

Some approaches combine chatbots with low-code platforms. In [202], chatbots are generated for webpages from their HTML source code, taking into consideration their semantics, links, and contained information. In contrast, our approach relies on (nested) components and considers the linguistic concepts managed in each component. In [203], Weber uses a node-based editor to orchestrate LLM tools for an IoT-targeted chatbot.

Finally, commercial low-code platforms are integrating chatbots as services for their development suites. For instance, OutSystems [61] and Salesforce [174] can generate chatbots from customers' application data. Salesforce also supports dedicated intents that trigger internally modeled workflows. Appian [175] covers these use cases and employs smaller,

[9] See *Enhancing Modeling Tools with AI: A Leap Towards Smarter Diagrams with Eclipse GLSP*

task-focused chatbots for concrete activities, including model retrieval and diagram construction. Although these chatbots are specifically crafted for each platform, to our knowledge they lack a conceptual understanding of the modeled entities and their relations, and their responses do not include links to help users navigate their platforms. In contrast, LowcoBot takes a model-driven approach in which the graphical, programmatic, and ontological aspects of a low-code platform are explicitly modeled and used to generate a chatbot configuration that can both understand platform concepts and guide users through the platform.

However, achieving this structural connection does not imply behavioral integration. That is, chatbots are typically treated as external components rather than integrated actors sharing the platform's context and permissions (e.g., Salesforce [174]). Therefore, while LowcoBot grounds chatbots in the platform's structural components, a deep integration of chatbots within behavioral workflows remains an open challenge.

3.5 Summary and conclusion

This chapter has reviewed the state of the art in the areas most closely related to the contributions of this thesis. First, it has traced the evolution of graphical language workbenches from early desktop tools to recent web- and cloud-based infrastructures. Second, it has examined approaches for visualizing and comprehending large graphs and models, as well as generic, pattern-based mechanisms for reuse. Third, it has analyzed commercial and academic low-code platforms together with workflow languages, specifically examining their model-driven features, access control mechanisms, and integration capabilities. Finally, it has discussed work at the intersection of modeling, low-code platforms, and conversational agents.

From this analysis, several gaps emerge. First, existing cloud-based modeling frameworks rarely combine industrial-grade, scalable persistence with configurable conformance relations that are accessible to citizen developers. Second, visualization approaches for large models are often language-specific and lack reusable, domain-aware sensemaking strategies. Third, most low-code platforms provide workflow and integration facilities, but these are typically inflexible, proprietary, and not systematically integrated with MDSE techniques. Finally, there is a lack of chatbots for low-code platforms that are systematically grounded in their domain models, which prevents fully exploiting their ontological structure.

These gaps motivate the next chapters, which present the contributions of this thesis —Dandelion, model sensemaking strategies, Dandelion+, and LowcoBot— beginning with the definition of graphical language workbenches in the cloud.

Part II

Contributions

Chapter 4

Definition of Graphical Language Workbenches in the Cloud

This chapter presents Dandelion, a cloud-based graphical language workbench for defining and using graphical domain-specific languages (DSLs). The chapter presents the approach (4.2), architecture (4.3), and tool support (4.4) of Dandelion. The tool is available online[1]*, and a demonstration video is available on YouTube.*[2] *The contributions of this chapter have been published in a technical paper and a tool demo in [37] and [38], respectively.*

4.1 Introduction

As discussed in Section 2.3, both MDSE and low-code development share the goal of raising the level of abstraction to streamline software development. However, when graphical DSLs are used inside industrial low-code platforms, several complications arise. First, most language workbenches are desktop-based and rely on file-based persistence, which hinders their integration into cloud-native and web-based tooling. Second, low-code platforms often combine multiple heterogeneous data formats, so editors must provide dedicated support for all of them. Third, industrial models can grow to hundreds of thousands of elements, requiring scalable storage and navigation mechanisms. This chapter introduces Dandelion, a cloud-based graphical language workbench for graphical DSLs that is tailored to circumvent these issues. Dandelion allows defining and using graphical DSLs directly from the web browser, and leverages MDSE principles so that languages are defined through models rather than code. The key features of Dandelion include:

[1] https://miso.es/tools/Dandelion.html
[2] https://youtu.be/kQjdOgopGvI

1. A harmonizing meta-model capable of representing (meta-)models from different technologies uniformly;
2. a concrete syntax meta-model to augment the designed DSLs with visual notations, usable from the web browser;
3. scalability configurations to explore, visualize, and comprehend large-scale models efficiently; and
4. support for flexible modeling to relax the conformance between models and meta-models in a controlled manner.

Part of Dandelion's design was driven by the industrial needs of the software firm UGROUND [36], which uses low-code development for its projects in areas including banking, human resources, insurance, and transportation [204]. In this sense, the challenge was developing a workbench that could produce a web modeling frontend integrated with UGROUND's low-code infrastructure, while still supporting generic low-code scenarios beyond this specific case. Models developed by UGROUND reach hundreds of thousands of elements and are defined using ROSE [40], a proprietary technology with no explicit meta-model to foster agility. The framework interprets models to yield the final system, enabling very fast development times based on digital twins for organizations [178].

4.2 Approach

This section presents Dandelion's approach. In particular, it provides an overview of the working scheme of Dandelion (4.2.1), whose main artifacts are (meta-)models conformant to the harmonizing meta-model presented in 4.2.2. In particular, meta-models can be complemented with a graphical syntax (4.2.3) and a scalability configuration (4.2.4). Finally, the section reports on the flexible modeling support of Dandelion (4.2.5).

4.2.1 Overview of the approach

Figure 4.1 presents the working scheme of the proposed framework.

The tool supports two roles: language engineers and citizen developers. DSLs are created in a two-step process (labels 1 and 2 in the figure). First, language engineers define an abstract syntax via a meta-model. They can then complement the meta-model with a graphical concrete syntax via a concrete syntax model and scalability mechanisms to improve navigation over large models. Moreover, language engineers can design flexible modeling phases to relax the conformance relation between models and meta-models in a controlled manner. Second, citizen developers can create, visualize, and manipulate models in these DSLs.

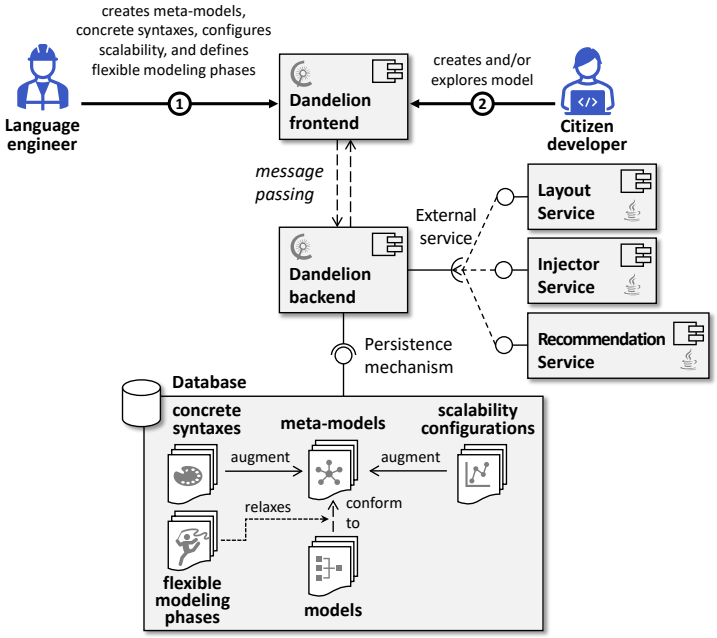


Figure 4.1 Working scheme of Dandelion.

The tool is split into frontend and backend components, which communicate via message passing. The frontend is a web-based graphical editor for defining and using graphical DSLs, whereas the backend is responsible for model management (the "heavy lifting"). The backend features a persistence mechanism that stores and retrieves meta-models —which describe the abstract syntax of DSLs— and models that conform to these meta-models. DSLs can be further augmented with concrete syntaxes, scalability configurations, and flexible modeling phases. Additionally, the backend has access to external services, including layout providers, external technology injectors, and recommendation services. The implementation or the integration with these services is written in the Java language to leverage existing MDSE tools and frameworks —commonly developed in this programming language. The backend interacts with these services via REST APIs.

4.2.2 Harmonizing meta-model

As low-code development platforms (LCDPs) may target different domains, technologies, and formats (e.g., EMF [26], OWL [205], RDF [206], or *ad hoc* industrial formats such as UGROUND's ROSE [40]), a common underlying data model is required to represent heterogeneous data uniformly. For this purpose, Dandelion features a harmonizing meta-model to support models from different platforms (Figure 4.2). This meta-model strikes a balance between simplicity and expressiveness, encompassing a large number of modeling styles while remaining simple, taking inspiration from [207].

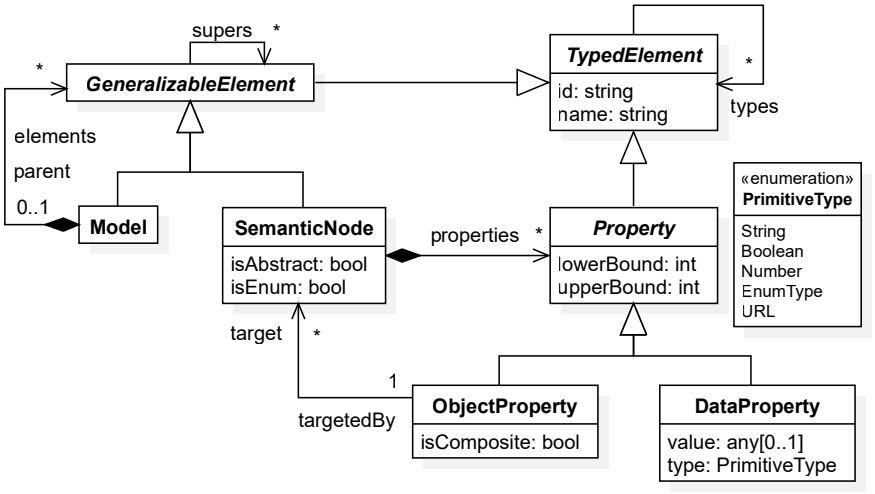


Figure 4.2
Dandelion's harmonizing meta-model.

The meta-model represents objects as SemanticNodes and links as ObjectProperty objects. SemanticNodes have properties with a cardinality given by the lowerBound..upperBound interval. Properties can be either primitive data types (DataProperty) or reference types (ObjectProperty). The latter can be composite. SemanticNodes can be declared abstract (isAbstract) and be used to represent enumerations (isEnum) —where each Property is a literal of the enumeration.

The meta-model can represent both models and meta-models uniformly. SemanticNode and ObjectProperty represent objects/links in the former, and meta-classes/associations in the latter. Therefore, this approach is level-agnostic [208] and —beyond traditional two-level modeling approaches like EMF— can represent an arbitrary number of meta-levels. This is because both Models and SemanticNodes (at any meta-level) may have a type (cf. Section 2.1.5). Also beyond EMF, this approach explicitly reifies the notion of Model, enabling multiple or no model types (relation TypedElement.types) and nested models (relation Model.elements). Both Models and SemanticNodes can be generalized via the reference supers. The current version of the harmonizing meta-model does not support the definition of constraints, like invariants expressed in the Object Constraint Language (OCL) [209].

Figure 4.3 exemplifies how the harmonizing meta-model can uniformly represent both domain-specific meta-models and instances thereof. The example is drawn from a small domain-specific meta-model for libraries (on the left). Next to it, an instantiation of this meta-model with a library and two books is shown (on the right). The top diagrams show how this domain could be expressed in a foreign technology to Dandelion, such as UML or EMF. Below, this domain-specific knowledge is expressed in Dandelion's harmonizing meta-model.

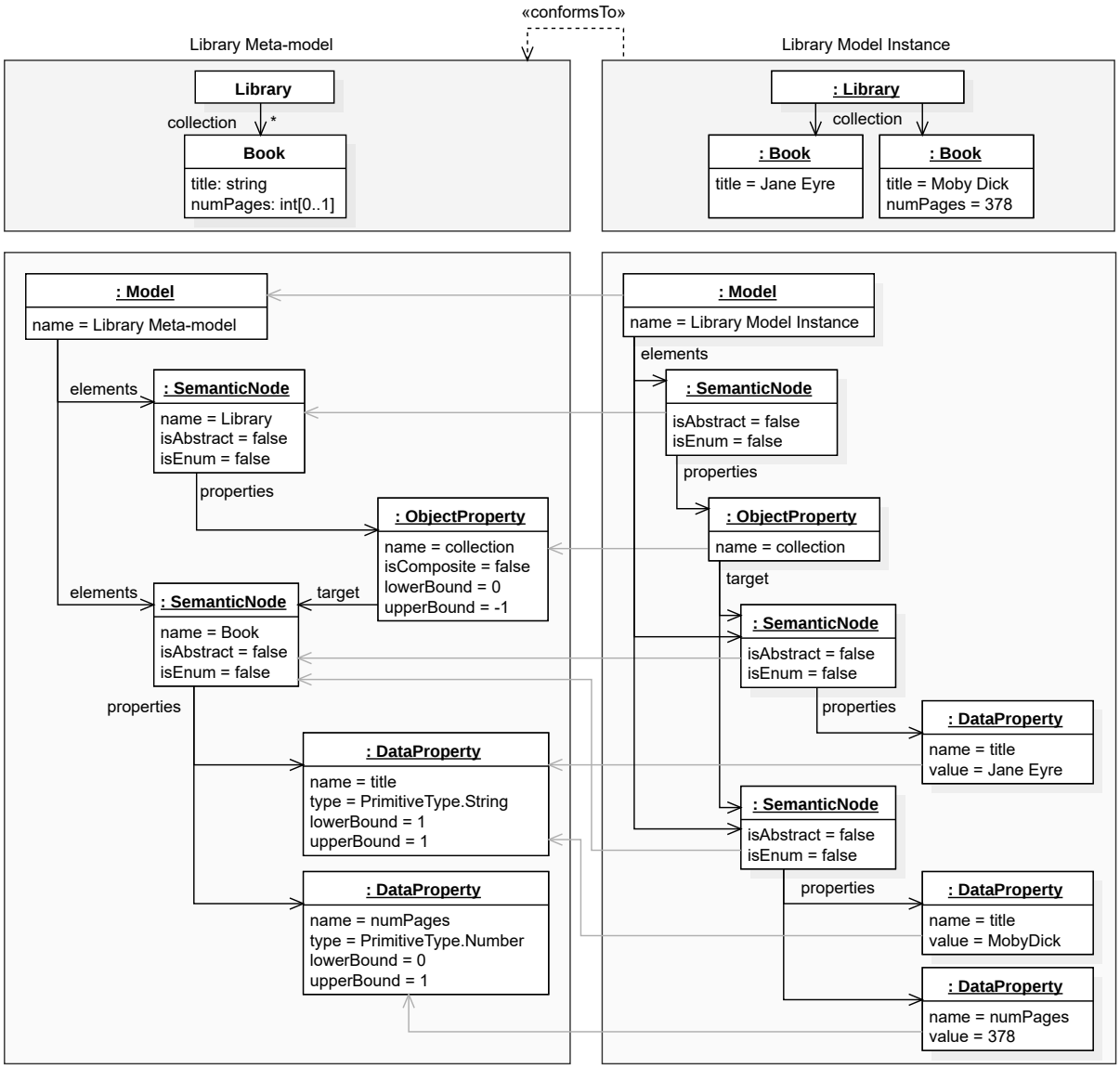


Figure 4.3 Example of a domain-specific meta-model and an instance thereof in Dandelion. Light arrows are TypedElement.types; IDs and many names have been omitted for brevity.

On the one hand, in the meta-model, meta-classes Library and Book are represented using SemanticNodes; primitive data properties Book.title and Book.numPages become DataProperty; and the link Library.collection is represented with an ObjectProperty. On the other hand, for the model instance, every object becomes a SemanticNode. Their properties' values are represented as DataProperty and ObjectProperty, depending on their nature. Their values are set via DataProperty.value and ObjectProperty.target,

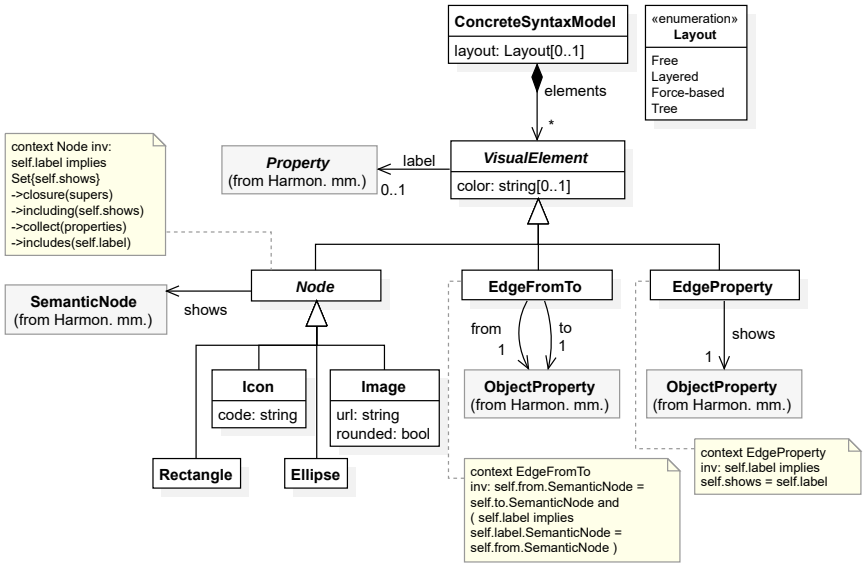


Figure 4.4 Dandelion's concrete syntax meta-model.

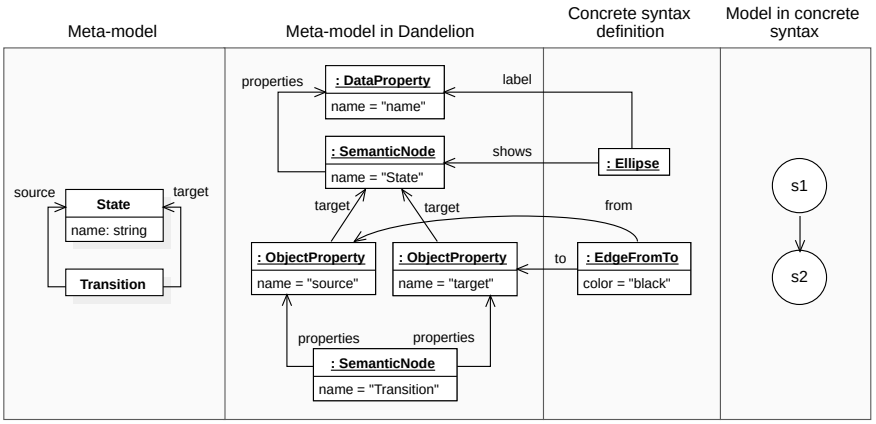
respectively. Additionally, artifacts are wrapped in Models. In the example, both the domain meta-model and its instance are represented with Models that contain SemanticNodes via Model.elements. Finally, to ensure the conformance between the model and its meta-model, every element of the model points to the meta-class it conforms to via TypedElement.types —shown as light arrows in the figure.

4.2.3 Concrete syntax meta-model

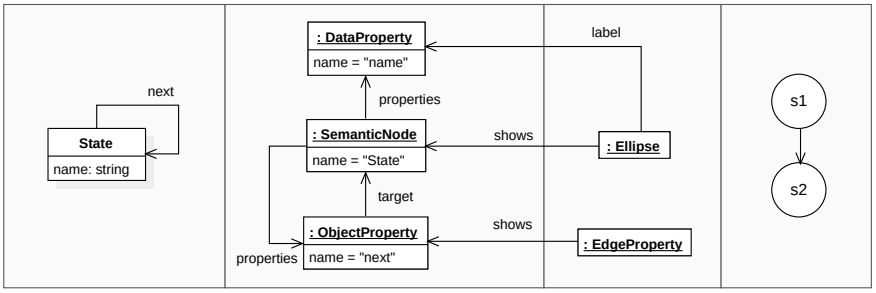
When users create a meta-model using the harmonizing meta-model, Dandelion implicitly assigns a default graphical syntax to it, where objects are depicted as rectangles, and links as arrows. This visualization can be overridden by augmenting the designed meta-model with a concrete graphical syntax through Dandelion's concrete syntax meta-model.

Figure 4.4 presents said meta-model. Notice how a ConcreteSyntaxModel contains VisualElements that assign visual representations to the meta-model elements they target. It may also define a Layout specifying how to arrange the model elements in the diagram. Dandelion currently supports four layout types: Free, for no layout; Layered, for hierarchy-based diagrams (e.g., class diagrams); Force-based, which simulates a physical repulsion-attraction between nodes; or Tree, which is optimal for acyclic graphs. This list is not exhaustive, as more layouts can be added in the future.

Visual elements can be either nodes or edges, and accept an optional color and a label. On the one hand, Nodes represent SemanticNodes via Node.shows. Besides traditional shapes such as rectangles and ellipses, Nodes can be represented using Icons (among those available in a catalog)



(a) Using EdgeFromTo.



(b) Using EdgeProperty.

Figure 4.5
Defining the concrete syntax of a finite-state machine meta-model with different edge types.

or Images. On the other hand, there are two Edge kinds: EdgeFromTo and EdgeProperty. Figure 4.5 illustrates the difference between both edge types. First, EdgeFromTo is used to display SemanticNodes with two ObjectProperty entities. Figure 4.5a shows an example, where Transitions in a finite-state machine meta-model have two properties, source:State and target:State, and are represented by an EdgeFromTo with from and to pointing to Transition.source and Transition.target, respectively. Second, EdgeProperty is used to visualize an ObjectProperty. Figure 4.5b shows a finite-state machine meta-model where the relationship between States is modeled with a next reference, so the concrete syntax uses an EdgeProperty. When the property pointed to by EdgeProperty.shows is set, an arrow is drawn from the SemanticNode that defines the property to the SemanticNode pointed to by the property.

The meta-model includes some OCL constraints to ensure soundness.[3] In practice, the tool ensures their enforcement at all times, as users can only create and edit concrete syntaxes through dedicated forms.

[3]These constraints ensure that VisualElement.label, if set, belongs to the target SemanticNode for nodes, and to the semantic node of the target ObjectProperty for edges. Additionally, EdgeFromTo.from and EdgeFromTo.to must belong to the same SemanticNode.

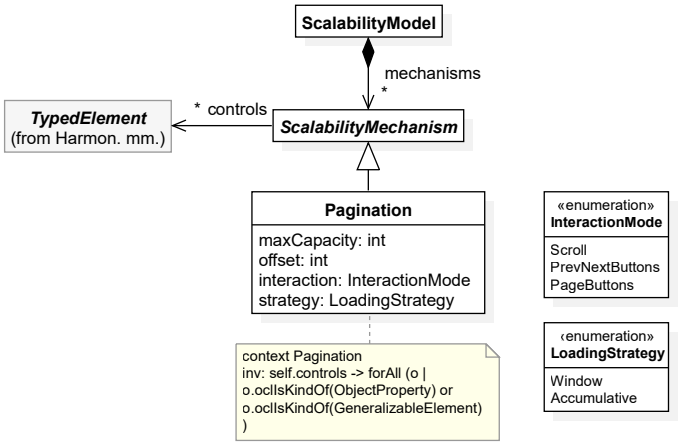


Figure 4.6 Dandelion's scalability meta-model.

Finally, the capabilities of this meta-model are currently constrained by the rendering engine of Dandelion, vis.js [210]. This library was selected to prioritize the rendering performance of large models and automatic layouts in the browser. Therefore, advanced diagramming features like nesting, manual label positioning, or snapping (cf. [132]) are not supported, relying instead on pagination and automatic layouts. This represents a design trade-off between visual flexibility and the simplicity and scalability required for the cloud environment.

4.2.4 Scalability configuration meta-model

Scalability mechanisms are needed to effectively handle large models and to prevent overwhelming users with too much information. In Dandelion, they are defined through the meta-model shown in Figure 4.6.

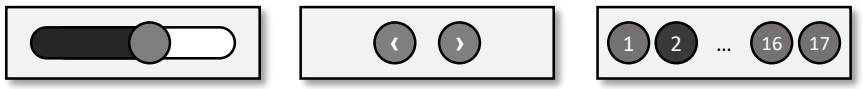
A ScalabilityModel contains a series of ScalabilityMechanisms that control the loading and visualization of some elements of a model.[4]

Currently, Dandelion supports the scalability mechanism of pagination, which splits large models into smaller pages that can be loaded on demand. Each page contains a limited number of nodes and the edges that emerge from them, both represented according to the model's concrete syntax. However, when an edge ends in a node that does not belong to the current page, the target node is displayed as a proxy node, which serves as a placeholder for the actual node. These are resolved upon clicking, and users can navigate to the page where the real nodes are located. Proxy nodes are distinguished by their concrete syntax, as they are represented as small, gray, round nodes.

Pagination is configurable by several parameters. The main one is maxCapacity, which determines the maximum number of nodes that fit in a

[4]Although ScalabilityMechanism.controls can target any TypedElement, specific ScalabilityMechanisms, like Pagination, can restrict their scope as indicated by the OCL constraint in the meta-model. The tool enforces these constraints automatically.

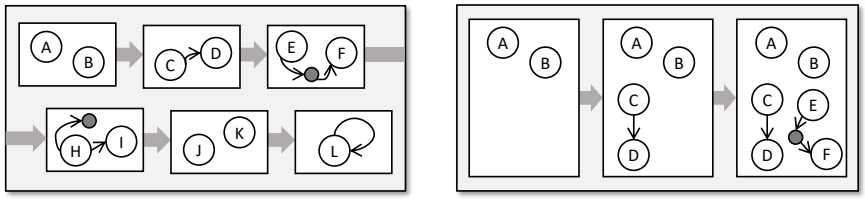
page. Additionally, the InteractionMode determines the UI, which can be a scroll bar, previous and next buttons, or traditional pagination with number buttons (see Figure 4.7).



(a) Scroll mode. **(b)** PrevNextButtons mode. **(c)** PageButtons mode.

Figure 4.7 Scalability interaction modes.

Pagination is also governed by a LoadingStrategy, which determines how elements are loaded and unloaded while switching between pages, as shown in Figure 4.8.



(a) Window strategy. **(b)** Accumulative strategy.

Figure 4.8 Scalability loading strategies.

On the one hand, the Window strategy (Figure 4.8a) unloads k elements from the previous page and loads k elements from the next one upon switching between pages, where the k parameter is determined by the offset attribute. This strategy scales to any model size since the elements shown on screen are limited to the page's maximum capacity, maxCapacity. On the other hand, the Accumulative strategy (Figure 4.8b) does not discard elements between pages, but accumulates them. That is, page n in an accumulative strategy not only contains the elements assigned to the n-th page, but also contains the elements from all previous pages, from the first one to the $(n-1)$-th one. This strategy helps observe the evolution of a model across pages, especially when combined with the Scroll interaction mode. However, as it renders entire sub-models, it is limited to relatively small models that fit in memory.

The assignment of elements to pages is currently determined by the insertion order of the elements in the database. It is planned as future work to consider configurable sorting criteria within pages. It is intended to exploit the semantic distance between elements (i.e., the number of edge hops between them) as a heuristic for the distribution of elements across pages. This way, the closer two elements are in the model, the closer they become in the pagination.

4.2.5 Flexible modeling support

As discussed in Section 2.1.6 and Section 3.1.3, flexible modeling encompasses techniques that allow relaxing the conformance relation between

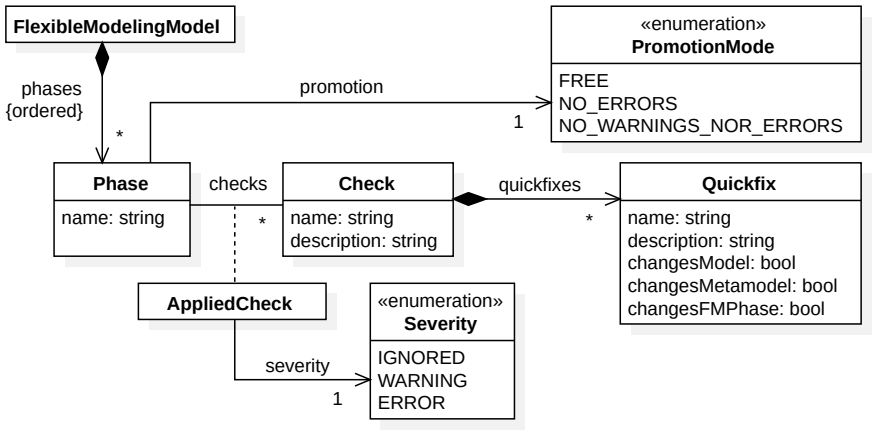


Figure 4.9 Dandelion's flexible modeling meta-model.

#	Check	Violation condition
C1	Cardinality of attributes	Attributes not respecting their multiplicity.
C2	Cardinality of references	References not respecting their multiplicity.
C3	Type of attribute values	Attribute values that do not respect the attribute's type.
C4	Type of reference values	References pointing to objects of the wrong type.
C5	Missing property	Objects missing a property defined in the meta-model.
C6	Superfluous property	Objects defining a property absent in the meta-model.
C7	Duplicate property	Objects defining a property with the same name twice.
C8	Non-existent target	References pointing to non-existent objects.
C9	Instantiated abstract class	Objects that are instances of abstract classes.
C10	Untyped objects	Objects that do not conform to a meta-class.
C11	Untyped properties	Properties not conforming to the meta-model.

(a) Checks for flexible modeling in Dandelion.

#	Check	Quick fixes
C6	Superfluous property	• Add the property to the meta-model. • Retype the superfluous property. • Relax the 'Superfluous property' check.
C9	Instantiated abstract class	• Make the meta-class concrete. • Implement a concrete sub-class.
C10	Untyped objects	• Create a new type matching the object. • Retype the object with an existing meta-class. • Relax the 'Untyped objects' check.

(b) Quick fixes for some checks in Dandelion.

Table 4.1 Checks and corresponding quick fixes for flexible modeling in Dandelion.

models and their meta-models in a controlled manner. Dandelion supports flexible modeling by reifying conformance constraint checks and allowing users to toggle their enforcement. Flexible modeling in Dandelion is governed by the meta-model shown in Figure 4.9, inspired by [66].

In particular, the tool exposes a series of checks (Check), which are conformance constraints. Table 4.1a presents the checks covered by the

tool. Overall, these checks cover conformance issues regarding cardinalities, types, property definitions, and object instantiation.

Next, to support a structured modeling process, checks are grouped into modeling phases (Phase), which specify the severity level of each check and the promotion policy between phases. In particular, the severity level is one of *ignored*, *warning*, and *error*. Regarding the promotion policy (PromotionMode), it can be FREE, if users can advance to the next phase without restrictions; NO_ERRORS, which prevents advancing if errors are present; and NO_WARNINGS_NOR_ERRORS, which is the most restrictive. Some of the checks are complemented with quick fixes (QuickFix), which can automatically repair the model, the meta-model, or the current modeling phase. Quick fixes may, for example, add missing properties, remove or merge superfluous ones, or update the types of objects and properties to restore conformance. Currently, Dandelion implements several quick fixes, as demonstrated in Table 4.1b.

4.3 Architecture

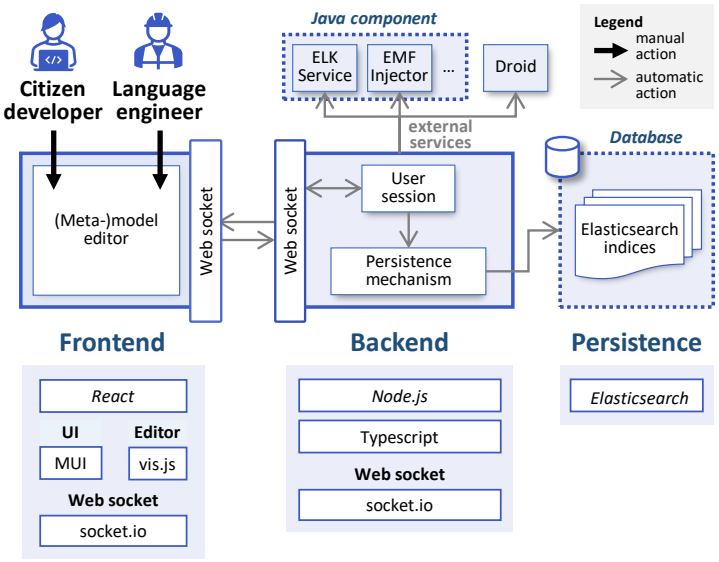
Figure 4.10 presents the architecture of the framework, which is split into frontend, backend, and persistence.

Both citizen developers and language engineers can access the application's frontend from their browsers, using different editors for models and meta-models, respectively. The tool is deployed as a single-page web application and has been developed in React [211]. It uses the vis.js [210] library to visualize models as graphs.

The backend is developed in Node.js [212] and is programmed in TypeScript [213]. It maintains a session for each user interacting with the tool, and acts as an intermediary between the frontend and the persistence layer. Specifically, it relies on a persistence mechanism that holds connections to Elasticsearch databases and dispatches queries on demand. The backend also supports external services (recall from Figure 4.1) including layout, injector, and recommendation services. At the moment, the tool supports out-of-the-box integration with an ELK Service, an EMF injector, and model recommenders built with Droid [197].

On the one hand, the ELK Service exposes the functionality of the Elastic Layout Kernel (ELK) [214], a framework that resolves layout arrangements according to a collection of highly customizable layout algorithms, via a REST API. This service, jointly with the EMF injector, is encapsulated in the *Java component*, which makes it possible to extend Dandelion with other services written in this programming language. On the other hand, Droid deploys model recommenders as REST services. Dandelion integrates with one such recommender to help in the construction of meta-models: given any meta-class, the recommender suggests suitable attributes.

Figure 4.10
Dandelion's architecture.



Frontend and backend communicate back and forth using web sockets: a bidirectional, full-duplex technology. This communication is enabled through the *Socket.IO* library [215] and it permits the frontend to receive real-time updates from the backend and vice versa. Currently, the tool uses an *ad hoc*, (meta-)model-independent protocol that relies on JSON messages to encode the dispatched commands (e.g., "getNFirstChildren" and "loadChildren" for loading a model with or without pagination, respectively). The plan in the future is to standardize it to comply with the Graphical Language Server Platform (GLSP) [107], making the appropriate adaptations to support the tool's particularities.

Persistence is implemented in Elasticsearch [123], a document-based, distributed, scalable search engine. Elasticsearch persists documents in indices, which are, roughly, the counterpart of tables in relational databases. The structure of these indices is shaped by a mapping, which is a JSON description of the indexed fields and how they are stored. To serialize and persist Dandelion (meta-)models (e.g., the ones in Figure 4.3), Elasticsearch indices must conform to a mapping compliant with the harmonizing meta-model presented in the previous section.

However, this step is not immediate, as there are multiple ways to translate a meta-model into a document-based database schema. How it is done can profoundly impact the database-level scalability of the entire system. Arguably, its core component is how associations are represented. The harmonizing meta-model has five one-to-many relations, which are represented as follows:

- Relations TypedElement.types and GeneralizableElement.supers are represented as lists of references to the IDs of the targeted documents.

The rationale is that these associations should not target a large number of elements and it is, thus, safe to store them as references.

- In the case of SemanticNode.properties, they are persisted as nested objects within the SemanticNode that contains them. This is because, according to the experiments in Section 8.1, SemanticNodes do not have more than a few dozen properties and, therefore, do not bloat SemanticNodes' sizes. This way, properties become directly available upon querying SemanticNodes, thus preventing additional requests. Moreover, given the composition nature of this relation, this representation ensures a coupling between the lifecycles of SemanticNodes and their properties. For instance, when a SemanticNode is deleted, all its properties should be deleted as well. In some data-intensive scenarios, such as IoT, SemanticNodes can contain an enormous number of properties. In such cases, this relation should be re-architected.
- Finally, Model.elements and ObjectProperty.target should be treated carefully, as they can point to a potentially unbounded number of elements. As in relational databases, these have been encoded as one-to-many relations in the *many* end: elements keep track of the IDs of the elements that point to them. So, relations GeneralizableElement.parent and SemanticNode.targetedBy are persisted instead of their opposites: Model.elements and ObjectProperty.target.

As a result, this architecture enables the persistence of hundreds of thousands of Models and SemanticNodes per index while maintaining acceptable performance, as will be shown in the evaluation in 8.1.

4.4 Tool support

This section describes Dandelion's meta-modeling and modeling functionalities in Sections 4.4.1 and 4.4.2, respectively. These capabilities will be illustrated with a DSL for event logs.

4.4.1 Meta-modeling with Dandelion

Dandelion is offered as a web application that can be directly accessed through the browser. As shown in Figure 4.11, the (meta-)model editor is divided into a series of collapsible panels. These are explained next, making reference to the circled labels on the figure:

1. Tree view The left panel displays the tree view, which facilitates navigation on the targeted Elasticsearch indices and their content. Each index corresponds to an entry that displays the models it contains in a nested way. It also contains the flexible modeling and general configuration panels.

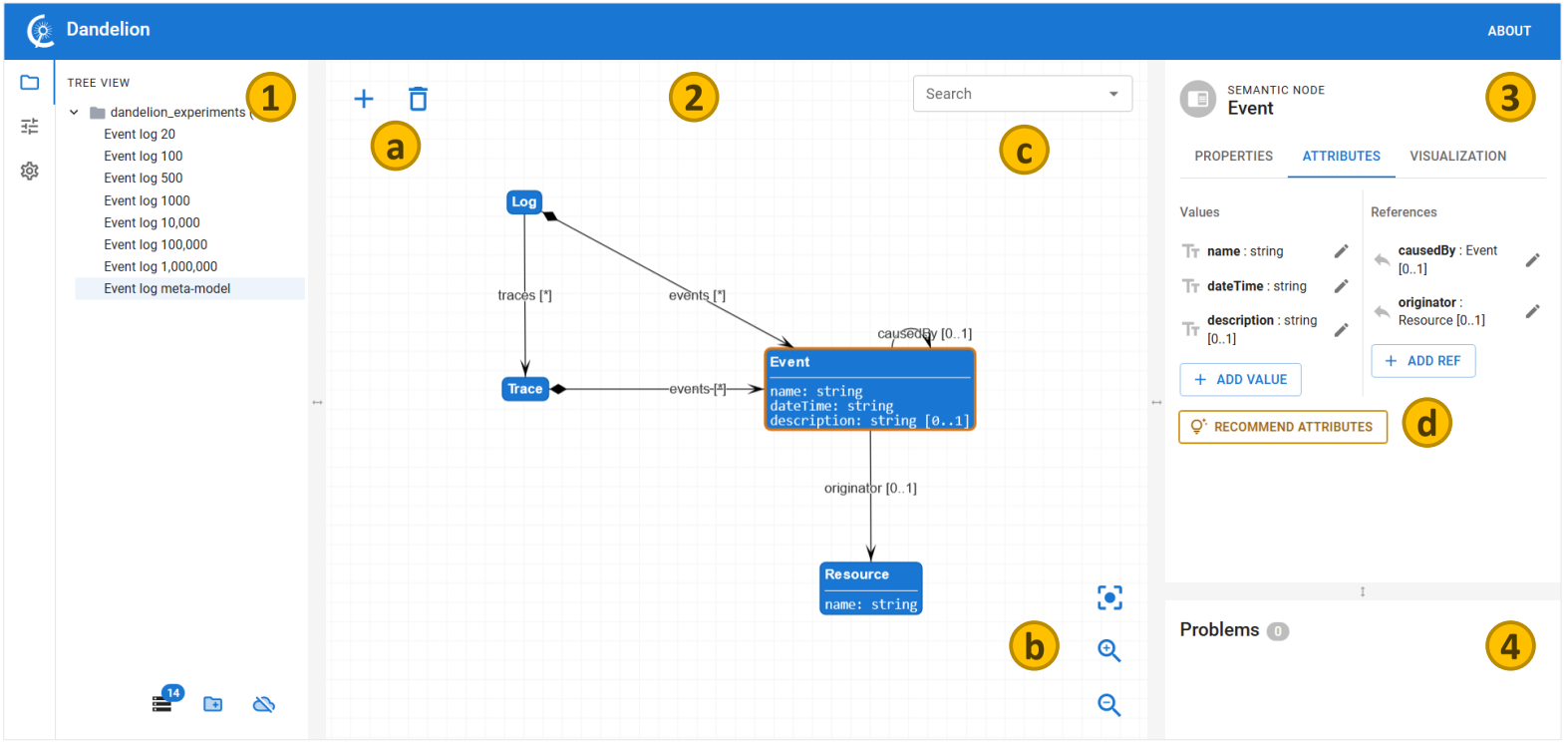


Figure 4.11 The (meta-)model editor, displaying the meta-model definition of a DSL for event logs.

2. (Meta-)model explorer The (meta-)model explorer allows visualizing and editing (meta-)models. For meta-models, it displays meta-classes with a dark blue background, along with their associations and inheritance relations. Abstract meta-classes are distinguished by a lighter shade of blue and their names are shown in italics. On the other hand, model instances display their elements according to the concrete syntax and the scalability configuration defined in their meta-model, if any. If no concrete syntax is provided, a default one for instance models is used, where nodes and edges are colored in gray. The plus and trash can icons in (a) permit adding new elements (i.e., SemanticNodes or Models) and deleting them, respectively. Buttons in (b) recenter the view and adjust the zoom level. Finally, the explorer features a search bar in (c) to rapidly pinpoint elements in the canvas.

The figure shows an adapted meta-model of XES [216], an IEEE standard for event logs that will also be used in Section 8.1.1 to demonstrate the scalability of Dandelion. Logs contain Events, which can optionally be wrapped within Traces. In turn, Events may originate at a certain Resource (in Event.originator), and can be optionally caused by another Event.

The explorer also supports a visualization that filters out all associations and displays meta-classes with a size proportional to their number of instances. As will be shown in Section 8.1.3, this is useful to understand how a meta-model is being used in practice.

3. Selected element editor The selected element editor permits manipulating the selected element's properties. Its tabs depend on the type of the selected element.

For Models (see Figure 4.12), the panel allows specifying the models they conform to, their concrete syntax, and their scalability configuration. Additionally, it exposes an import/export mechanism to download the selected model in a JSON Dandelion-compliant format, and to upload models in the Dandelion format or in EMF. For the latter, it makes use of the EMF injector, which is able to import the basic constructs of the meta-language. If the displayed model is a meta-model, it also permits toggling the list of attributes. Finally, the panel grants access to the layout service to rearrange the model's elements to a selection of predefined layouts.

For SemanticNodes inside a meta-model (i.e., meta-classes, such as Event in Figure 4.11), the panel displays their name, type, superclasses, is-abstract, is-enum, the attributes defined therein, and, optionally, a definition of their concrete syntax in the "Visualization" tab. Moreover, and to facilitate the process of creating meta-models, Dandelion integrates a recommender system generated by DROID [197] —a framework automating the development of recommender systems for modeling languages. The recommender is available through a "recommend attributes" button next to the attributes editor of a meta-class ((d) in Figure 4.11). When clicked, the application requests

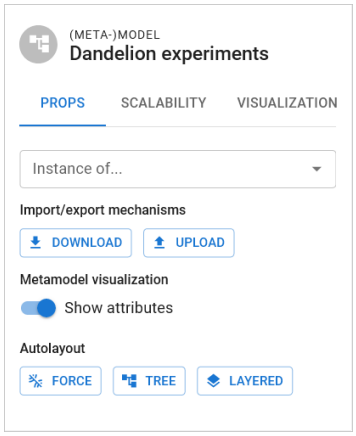


Figure 4.12
Selected element editor for a model.

from DroidREST —the REST API provided by Droid— a recommendation for the selected meta-class. The recommendation is displayed as a list of selectable attributes, sorted by likelihood, and users can select those attributes that may better fit their design, which are then added to the meta-class.

Finally, for SemanticNodes inside models (i.e., instances of meta-classes), the panel lists and permits manipulating the values of the properties.

4. Problems view and flexible modeling The problem view reports conformance problems in the current (meta-)model. This conformance is governed by the flexible modeling settings, which can be configured through the flexibility settings panel, as shown in Figure 4.13, which depicts an exemplary network diagram model. Each phase can be modified in order to specify which checks are enabled and with which severity (Figure 4.15). Violations appear in the *Problems* panel, and those with associated quick fixes present the available repairs. In the example, the object Computer4 is untyped, violating C10, and Dandelion offers the quick fixes associated with this check (cf. Table 4.1b).

The definition of DSLs in Dandelion is highly reactive: concrete syntaxes, scalability configurations, and flexible modeling settings update automatically upon modification. These are interpreted by the tool and, therefore, do not require code generation or restarting the application once defined. However, support for meta-model/model co-evolution techniques to repair already created models after changing the meta-models they conform to is not currently provided [217]. Discrepancies between models and meta-models can, though, be detected thanks to the flexible modeling support.

4.4.2 Modeling with Dandelion

Once a DSL has been created in Dandelion as described in Section 4.4.1, citizen developers can start using it. Figure 4.14 shows an instance of the

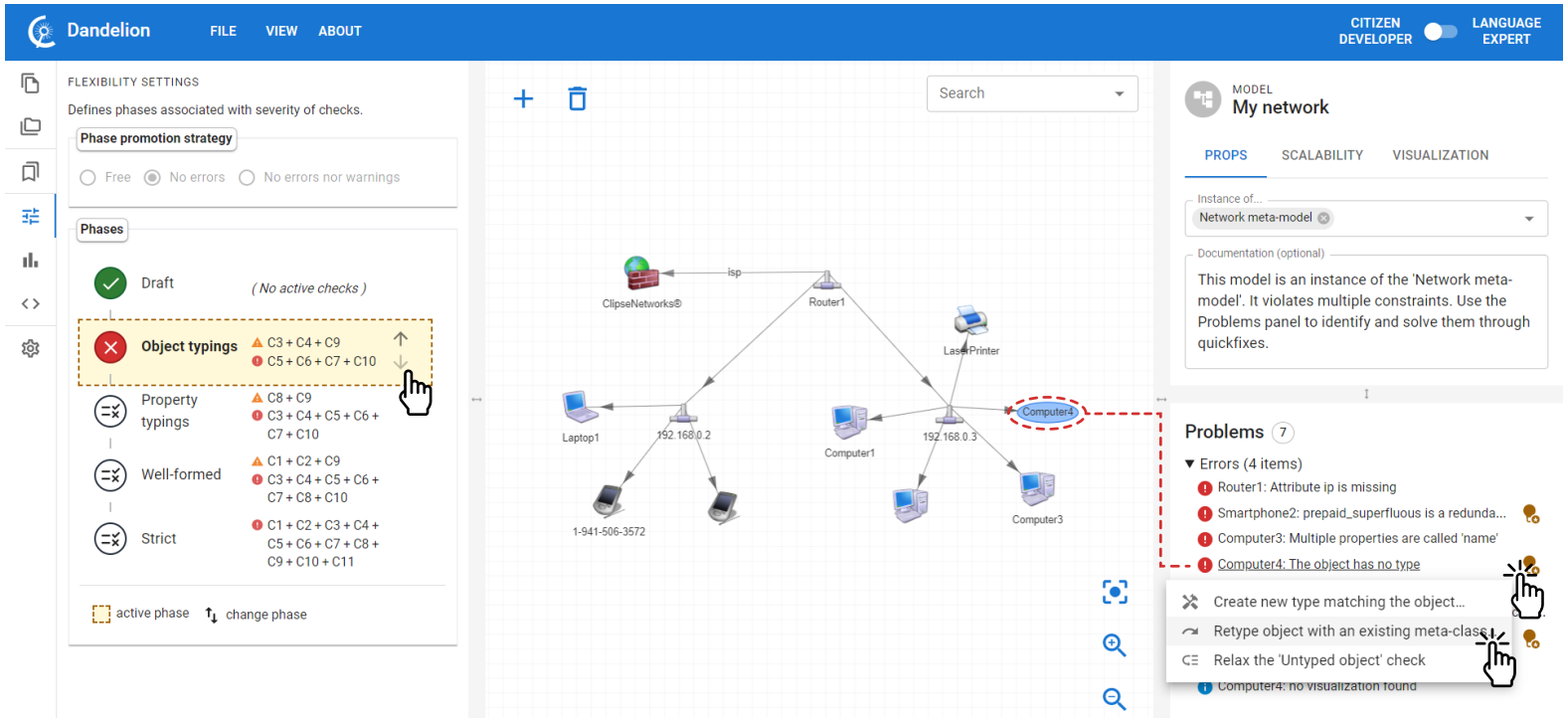


Figure 4.13 Flexible modeling in Dandelion. LEGEND: Left: flexibility settings panel; right: reported problems and suggested quick fixes.

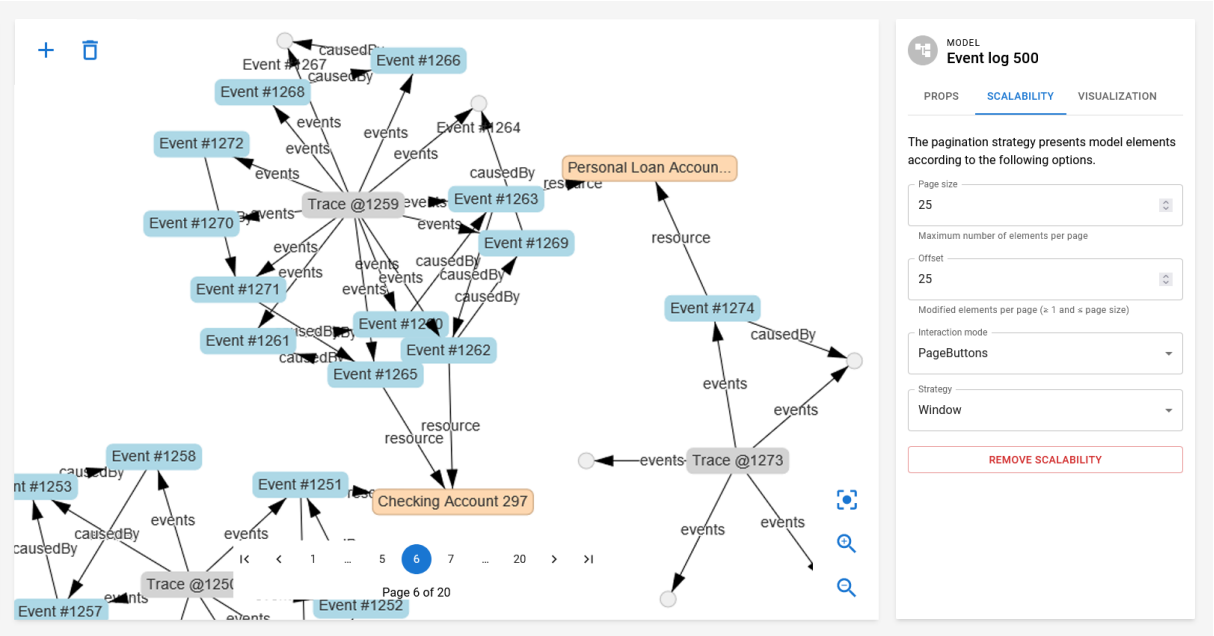
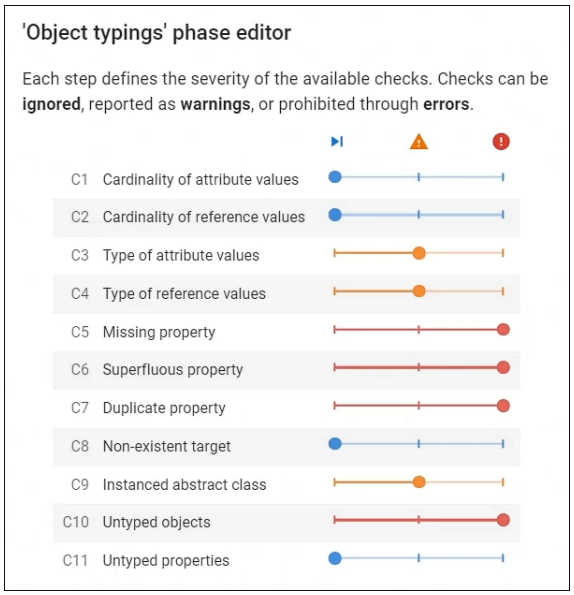


Figure 4.14 An event log model displayed using pagination, showing the PageButtons interaction mode and the Window loading strategy with pages of size 25. The scalability panel of the current model is shown on the right.

Figure 4.15 Flexible modeling phase editor: each check can be ignored, reported as a warning, or prohibited as an error.



meta-model from Figure 4.11. It is an event log of size 500 visualized with pagination. In this case, the page size is set to 25, yielding 20 pages. Traces are represented as rounded rectangles with a gray background, events are represented in blue, resources in orange, and proxies are visualized as

smaller circles. The panel to the right shows the pagination configuration. Its values are predefined at the meta-model level by the language designer, but the citizen developer can adapt them to the specifics of the model being edited.

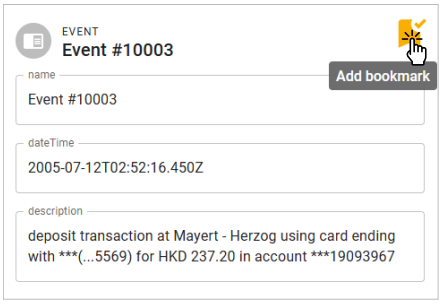
Navigating large models with pagination introduces a new challenge: users may lose track of elements located on unloaded pages. To address this, Dandelion introduces bookmarks to mark SemanticNodes of interest (see Figure 4.16a). Bookmarks are grouped by type for easier identification. Clicking on a bookmarked element navigates to the page where it is located (if pagination is configured), and selects it. In the (meta-)model editor, bookmarks are distinguished with a thick, green border (see Figure 4.16b).

4.5 Summary and conclusion

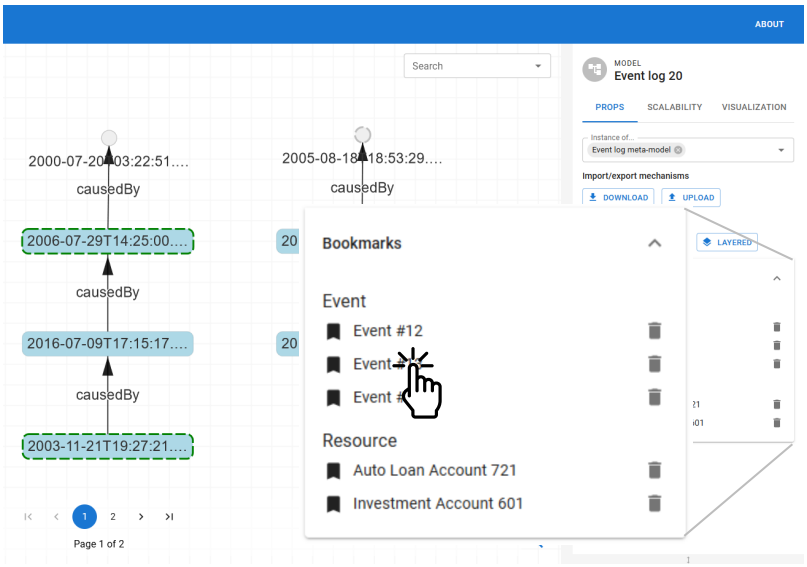
This chapter has introduced Dandelion, a scalable, cloud-based graphical language workbench for defining and using graphical DSLs in low-code engineering platforms. The tool relies on Elasticsearch, a flexible and scalable database, to persist and query models efficiently, and supports heterogeneous modeling technologies through a harmonizing, level-agnostic meta-model. Models defined in Dandelion can be augmented with concrete syntaxes and scalability configurations. In particular, this chapter has detailed the pagination mechanism and, as the evaluation in Section 8.1 will show, its effectiveness in partitioning and visualizing large models within a few seconds for suitable choices of page capacity. Moreover, Dandelion provides flexible modeling mechanisms that relax the conformance between models and meta-models through configurable checks, phases, and quick fixes. Finally, Dandelion can be used to create web frontends for industrial LCDPs by injecting UGROUND's modeling technology and to visualize large industrial models using pagination, as will be demonstrated in the evaluation in Section 8.1.

The next chapter presents model sensemaking strategies (SMSs), a model-driven approach to designing and implementing reusable, domain-agnostic, and purposeful visualizations for modeling ecosystems. SMSs are implemented on top of Dandelion.

Figure 4.16
Bookmarks in Dandelion.



(a) Bookmarking a SemanticNode.



(b) Bookmarked elements are distinguished on the explorer, and grouped by type in the bookmarks section.

Chapter 5

Purposeful Visualizations for Understanding Models

This chapter introduces model sensemaking strategies (SMSs), a novel approach for defining purposeful visualizations for understanding large and complex models. The chapter presents the approach (5.2), how to apply it (5.3), a catalog of strategies (5.4), architecture (5.5), and tool support (5.6). The contributions of this chapter have been published in a conference paper [34].

5.1 Introduction

Large and complex models have become pervasive in many areas, including embedded systems, process modeling, and software design. Understanding these models has therefore become a ubiquitous task, which calls for the development of specialized techniques. Given the limitations of human cognition [57] and the increasing complexity of models [29], many approaches rely on graphical visualizations —typically under umbrella terms such as *graphs*, *diagrams*, *networks*, and *maps* [218]. When these visualizations use domain-specific mechanisms to support the user's understanding, they can offer a cost-effective way to perform sensemaking tasks [145].

The increasing popularity of model-based and low-code platforms has highlighted the need to visualize and understand large artifacts created with these platforms [32]. In these fields (and especially within industrial settings), producing effective visualizations is particularly challenging. Specifically, models can reach millions of elements, exhibit intricate relationships with each other, and contain many attributes. Therefore, naïve visualizations are prone to overwhelming users with graphical information, creating cognitive overload. This is especially problematic in low-code scenarios, used by citizen developers with different backgrounds and limited modeling skills.

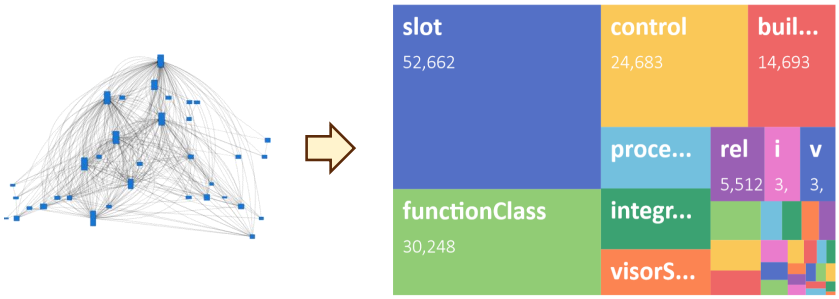


Figure 5.1 UGROUND's ROSE industrial meta-model (left) and the same meta-model through the lens of a *Categorical* sensemaking strategy (right).

Graph-like visualizations —typically offered by modeling tools— may not suffice for model sensemaking tasks, since they may fail to reveal underlying information in the explored models. Instead, other visualizations such as charts, plots, maps, or matrices can better support sensemaking tasks. These may rely on summarization techniques, using derived information by aggregating, filtering, or partitioning the elements within the current model, or sets of them, as covered in Section 3.2. These alternative visualizations can be manually created by developers of modeling environments, but their construction is costly. Moreover, flexibility is required to cater to the specifics of particular domain-specific languages (DSLs).

To exemplify the potential of purposeful visualizations for understanding models, Figure 5.1 illustrates UGROUND's industrial meta-model [40], which includes 35 meta-classes, and more than 300 edges and 500 attributes. Viewed through the lens of a *Categorical* sensemaking strategy (detailed in Section 5.4.1), each rectangle represents a meta-class whose area is proportional to its number of instances. This visualization supports frequent model comprehension tasks for language engineers: "*Which DSL concepts are most relevant for users?*" and "*How is the DSL used in practice?*".

To overcome these issues, this work proposes the new notion of model sensemaking strategy (SMS), drawing from the theory of sensemaking [144], [145]. Model sensemaking strategies are aimed at accomplishing specific model understanding tasks, offering visualization metaphors for them, such as charts, plots, maps, graphs, or matrices (cf. Figure 5.1). SMSs are reusable, since each strategy exposes a small meta-model pattern that, when bound to a target meta-model, results in a tailored visualization for the target meta-model instances. SMSs are also flexible, as they can be used to understand (meta-)models at any level of abstraction, as well as entire modeling ecosystems. Moreover, SMSs can be organized into dashboards combining SMSs to perform multiple model understanding tasks simultaneously.

This chapter presents the theoretical foundations of SMSs, an extensive catalog of SMSs in Appendix A, and a recommender to suggest suitable SMSs for a given meta-model. The approach is implemented atop Dandelion (Chapter 4).

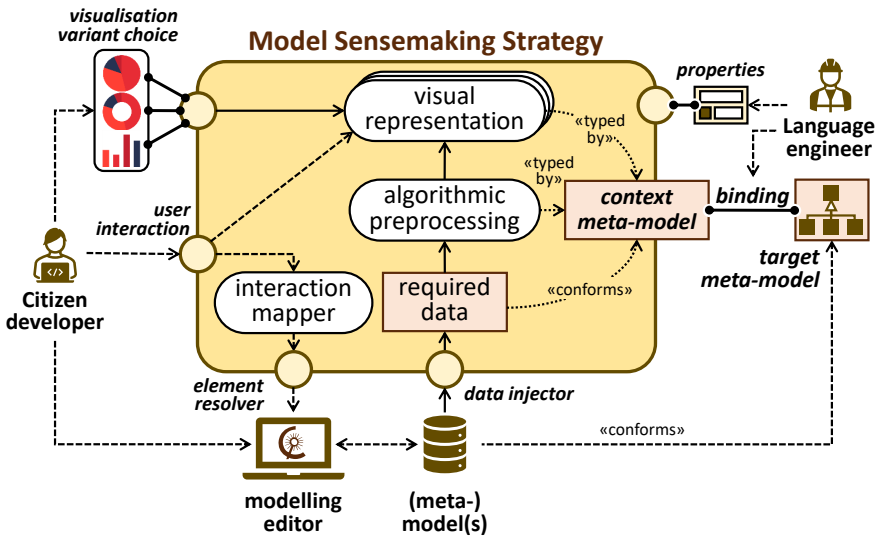


Figure 5.2 Components of a model sensemaking strategy (SMS).

To demonstrate the applicability of the approach, the effectiveness of SMSs has been evaluated within UGROUND, as models in the company can reach hundreds of thousands of elements and are defined using ROSE [40], a proprietary technology that interprets models to yield the final running system [204]. The goal was to develop a workbench to facilitate understanding of UGROUND models (to help developers of applications) and the ROSE language itself (to understand how the language is used and help in its future evolution). As later evaluated in Section 8.2, Dandelion was able to facilitate the understanding of ROSE, the entire UGROUND modeling ecosystem, and particular models.

5.2 Approach

The presented approach relies on the definition of model sensemaking strategies (SMSs) that can be reused and customized for specific DSLs. Figure 5.2 provides an overview of their main components.

An SMS yields an interactive visual representation for (graph-based) data according to a visual metaphor, e.g., plots, bar charts, or adjacency matrices. In this model-centric approach, data is extracted from the (meta-)model(s) being explored via a data injector. This data may undergo an algorithmic preprocessing step, if necessary. Some SMSs support multiple visualization variants, which can be switched at run-time. For example, Figure 5.2 shows some supported visualization alternatives for the Categorical SMS: pie, donut, and bar chart. All the steps depend on the context meta-model of the SMS, which the SMS exposes to make it reusable. Finally, SMSs can define and expose properties to refine the resulting visualizations (e.g., specifying

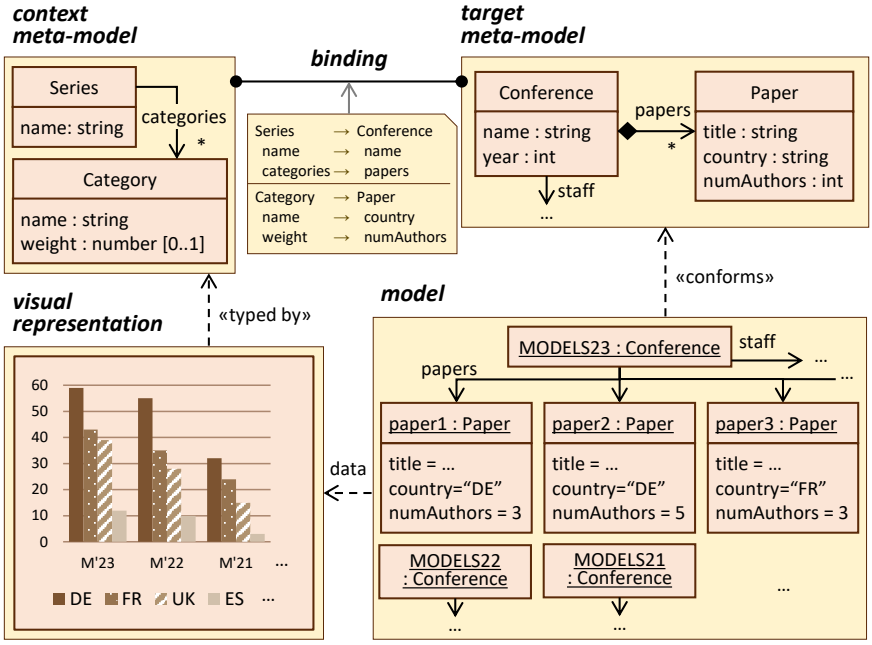


Figure 5.3
Application example of the Categorical SMS.

the axes titles in a bar chart).

DSLs can reuse an SMS through a binding. This model artifact specifies how the SMS context meta-model elements are mapped to those of the DSL target meta-model. For efficiency reasons, SMSs do not rely on explicit model-to-model transformations from DSL models to the format of the SMSs. Instead, they use notions from generic programming [219], where the SMS can be seen as a generic component that is instantiated by a binding to the target meta-model. As the next section shows, this target meta-model can be a domain meta-model (specifying the abstract syntax of a DSL), or the linguistic meta-model of the modeling environment. In the first case, the SMS is applicable to instances of the meta-model. In the second, it is applicable to each (meta-)model built within the environment. To support this approach, a recommender has been developed that computes possible bindings and provides a language to compute derived expressions.

SMSs provide interaction support at run-time. The interaction mapper specifies the (reactive) behavior of the visualizations upon user interaction. It establishes a bidirectional communication with the modeling editor, bridging the gap between the impacted model elements and their representation in the SMS. For example, an interactive tree-view SMS becomes a navigation utility, enabling users to explore the model by clicking its nodes.

While modeling with DSLs, citizen developers can use the SMSs configured (and bound) by a language engineer, which are presented in dashboards to allow exploring multiple sensemaking tasks at once.

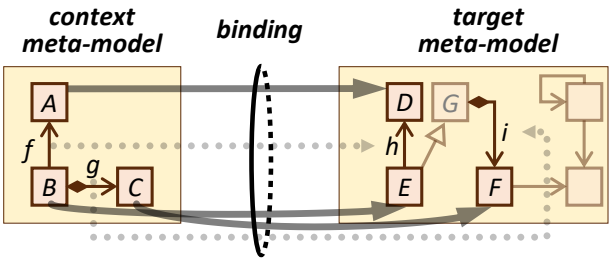


Figure 5.4
Binding a context meta-model to a target meta-model.

5.3 Applying sensemaking strategies

The key component for applying an SMS is its binding, which maps every concept of the SMS's context meta-model to elements of the target meta-model —see Figure 5.4.

As an example, suppose the steering committee of the MODELS conference wants to study the origin countries of the submitted papers per edition of the conference. The *Categorical* SMS fits this intent. Figure 5.3 depicts the application of this SMS to an example meta-model (with conferences and associated papers), and the resulting visualization for a sample model —more on this SMS in Section 5.4.1. The SMS is applied via a binding, which, once established, allows the visualization of any conformant model using the strategy. Moreover, if the SMS supports multiple visualization variants, these can be switched at run-time.

5.3.1 The context meta-model

The context meta-model defines the application pattern of an SMS. It has to be bound to a target meta-model so that the SMS can be used to visualize meta-model instances. Each element of the context meta-model (i.e., classes, attributes, and references) can be deemed conceptual "holes" to be populated by elements of the target meta-model. Context meta-models usually contain few elements, enabling their application to a wide range of target meta-models in different domains.

5.3.2 The binding

The binding determines how to apply an SMS to (the instances of) a target meta-model. It is a structure-preserving mapping $b : C \rightarrow T$ relating each class, attribute, and reference of the context meta-model C to elements of the target meta-model T. This approach closely follows [154], [220] for the definition of the well-formedness conditions presented in Table 5.1.

Non-injective mappings are supported, e.g., binding two classes c_1 and c_2 in C to one class c' in T (i.e., $b(c_1) = b(c_2) = c'$). The same holds for attributes and references. Generally, a well-formed binding b ensures that,

Classes	If c is a class in C, then $b(c)$ is also a class in T.
Class subtyping	It is preserved and reflected: c_1 is a subtype of c_2 in C (written $c_1 \leq c_2$) iff $b(c_1) \leq b(c_2)$.
Attributes	If a is an attribute defined or inherited in class c in C, then $b(a)$ is also an attribute inherited or defined in class $b(c)$. The type of the attribute must be preserved or refined in the binding: $b(a).\text{type} \leq a.\text{type}$. For instance, an attribute of type double can be bound to an integer.
References	If r is a reference from class c_1 to class c_2 in C, then $b(r)$ is a reference from class c_1' to c_2' in T, such that $b(c_1) \leq c_1'$ and $c_2' \leq b(c_2)$.
Composition	It is preserved: if r is a composition in C, then so must be $b(r)$.
Cardinalities	If f is an attribute or reference in C with a multiplicity interval $[l..h]$, then $b(f)$ should have multiplicity $[l'..h']$, with $l \leq l'$ and $h' \leq h$.

Table 5.1 Well-formedness criteria for bindings.

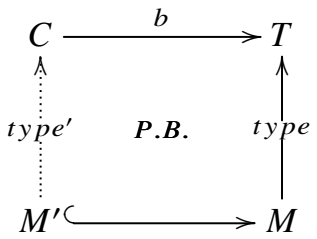
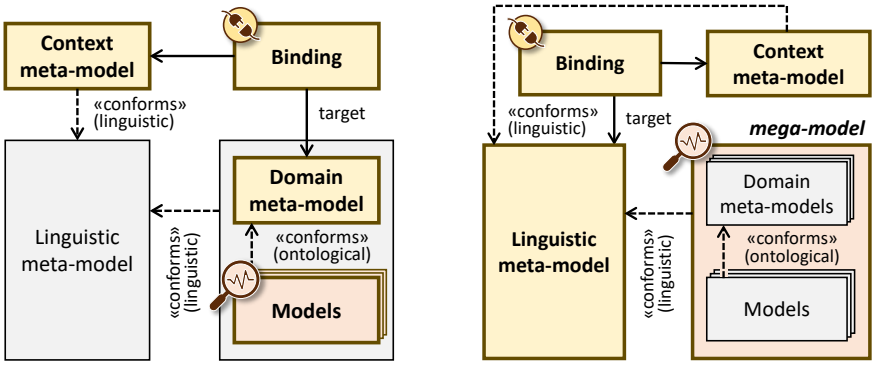


Figure 5.5 Pullback: extracting a model M' of the context meta-model C via a binding b.

for any model M conformant to T, slicing M w.r.t. the bound elements $b(C)$ yields a valid model M' of C, as Figure 5.5 depicts. Slicing a model M w.r.t. a "smaller" meta-model can be expressed using the categorical notion of *pullback* (P.B.) [63], [221]. Figure 5.4 exemplifies these binding criteria. For example, reference g can be mapped to i because i is inherited by E, which is bound from the owner of g. Formally, $b(g) = i$, which is correct since $b(B) = E \leq G$ (the source of i), and for the target of i, F, it holds that $F \leq b(C) = F$, as required by the well-formedness criterion for references.

In practice, a structural mapping may be too restrictive. Hence, this approach enables the specification of derived property bindings using an expression language. For example, imagine that Paper.numAuthors in the target meta-model of Figure 5.3 did not exist, and instead a Paper.authors reference to a class Author was introduced. The binding of Category.weight could, then, be expressed with / this.getReference("authors").target.length. In this case, the keyword this represents the bound Paper object. Section 5.5 will give more details about this language.



(a) Domain-specific SMS.

(b) Domain-agnostic SMS.

Figure 5.6
SMS classification by target meta-model.

5.3.3 The target meta-model

SMSs can be applied to (meta-)models independently of their level of abstraction. The only requirement is establishing a sound binding. This setting is especially suitable for multi-level modeling as, in this modeling paradigm, (meta-)models can conform in two ways to other models: via ontological or linguistic conformance [222], as detailed in Section 2.1.5. This leads to two types of SMS applications: domain-specific and domain-agnostic, depending on the type of conformance to the target meta-model.

Figure 5.6 depicts both cases. On the one hand, domain-specific SMS applications emerge from ontological conformance. As shown in Figure 5.6a, the binding targets a specific domain meta-model, thereby allowing the visualizations of any (ontologically) conformant model. Figure 5.3 exemplifies the application of a domain-specific SMS. On the other hand, domain-agnostic SMS applications are bound to the linguistic meta-model of the modeling framework, as depicted in Figure 5.6b. As, by construction, every (meta-)model conforms (linguistically) to the linguistic meta-model, this allows the visualization of any (meta-)model. This approach is helpful in two situations:

To understand (complex) meta-models. By targeting the linguistic meta-model, domain-agnostic bindings treat domain meta-models as (linguistic) instances. Therefore, they can be applied to any meta-model, regardless of their domain or complexity. For example, Figure 5.10 displays an SMS representing a meta-model as a heat map, while Figure 8.11 shows SMSs counting the number of elements within a meta-model; their division into concrete/abstract classes and enums using a bar chart; the prevalence of attribute names using a cloud map; and the distribution of meta-classes as data- or connection-centric using a scatter plot. Since these SMSs are domain-agnostic, they can be used to understand models as well.

To understand whole modeling ecosystems (mega-models). Models at any level of abstraction are (linguistic) instances of the linguistic meta-model. This means that bindings of agnostic SMSs can gather data about element instances through the expression language. For example, an agnostic SMS can be used to understand how a language is used, by presenting a proportional area chart, where meta-classes are represented as rectangles with size proportional to their number of instances (cf. Figure 5.1).

5.3.4 Strategy properties

SMSs feature properties to customize the resulting visualizations. Each SMS defines its own set of properties, which can be mandatory or optional, and are typed. For instance, all the SMSs define a title property (a mandatory string), and SMSs displaying Cartesian axes allow the definition of X and Y labels (optional strings). Unlike bindings, properties are independent of the target meta-model and are populated by value.

5.4 Catalog of sensemaking strategies

Appendix A presents a catalog of 10 SMSs, which can be displayed using 20 visualizations. Each SMS is described following a consistent format organized into sections, similar to traditional software design patterns [185]. Specifically, each SMS description includes its intent (i.e., the goal of the strategy), presentation metaphor, visualization variants, and properties. Motivating examples accompany the explanations.

A summary of these SMSs is presented in Table 5.2. Similar SMSs have been grouped by common presentation metaphors (PM), including numerical charts and plots; SMSs grouping elements into categories; metric-based SMSs; SMSs for models with time; and SMSs targeting model structure.

To maintain focus, only two SMSs are detailed: *Categorical* (Section 5.4.1) and *Weighted Hierarchy* (Section 5.4.2). The remaining SMSs are described in Appendix A.

5.4.1 Categorical

Aggregation is a powerful technique for summarizing large amounts of data: elements that share a feature are grouped together, partitioning datasets into categories. The *Categorical* SMS exploits this technique in the realm of models to visualize categories and their incidence.

Intent The *Categorical* SMS aims to understand the partitioning of objects into a set of (possibly unknown) categories. Some relevant sensemaking tasks include: "*What categories emerge from the data?*" and "*What is the*

PM	SMS	Visualization variants	Description
D	*Numerical*	Line graph, area chart, scatter plot	For (x, y) coordinates.
D	*Numerical w/ frequency*	Bubble chart	For (x, y) coordinates with a frequency.
G	*Categorical*	Vertical/horizontal bar chart, (semi-circle) donut chart, pie chart, proportional area chart	To partition data into categories.
M	*Metric distribution*	Boxplot	To depict the main percentiles of a metric.
M	*Free metric*	Highlighted number, icon and number	To visualize an unconstrained value.
M	*Bounded metric*	Angular gauge, gauge chart	To visualize a value bounded in a range.
M	*Literal metric*	Word cloud	To represent frequency in a textual field.
T	*Time-based*	Gantt chart	To show timed tasks with start and finish dates.
S	*Connectivity*	Adjacency matrix/heat map, chord diagram	For cross-referenced objects.
S	*Weighted hierarchy*	Treemap	For nested objects.

LEGEND: PM = presentation metaphor: D = data; G = grouping; M = metric; T = time; S = structural.

Table 5.2 Overview of model sensemaking strategies and their visualization variants.

incidence of each category?". The last task can be refined into discovering the most and least frequent categories.

Presentation metaphor This is a grouping-based SMS. Other examples under the same metaphor include cluster analysis and pattern matching.

Context meta-model It contains two classes, Series and Category, both identified by name (cf. Figure 5.3). Each series is associated with a set of categories, refining the scope of the analysis. Categories expose an optional numeric weight, which determines the contribution of the bound object to its category tally. This weight is assumed to be 1 if left unspecified. Objects sharing a Category.name are aggregated into the same category, and the visualization is replicated for each series.

Visualization variants Vertical and horizontal bar charts, (semi-circle) donut, pie, and proportional area charts (see Figure 5.11a).

Properties The SMS introduces three string properties: title, X_label, and Y_label —the former being mandatory, and the latter two optional. These only apply to bar charts, the visualization variants that display axes labels.

Motivating example The example in Figure 5.3.

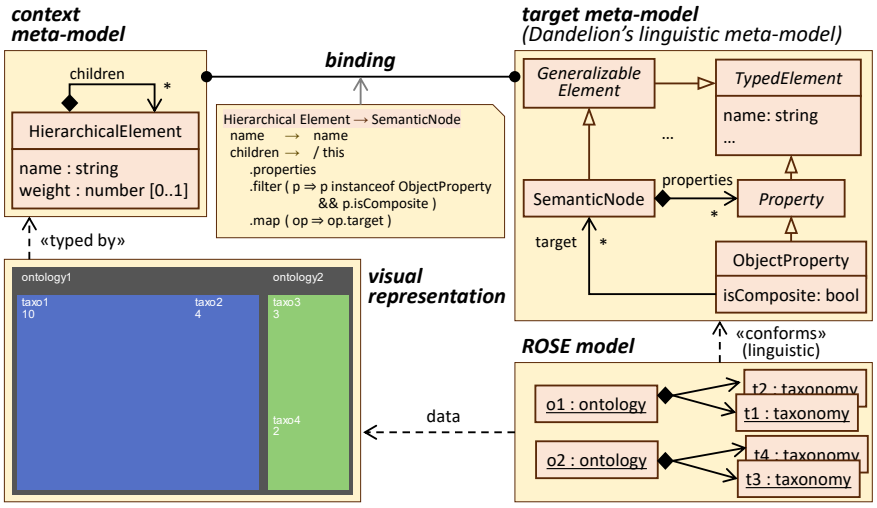


Figure 5.7 Weighted Hierarchy SMS, bound to Dandelion's linguistic meta-model, and used to understand a ROSE model.

5.4.2 Weighted hierarchy

Containment plays a central role in enabling abstraction in Software Engineering. It describes the relationship between two objects in which one object is a part of —or belongs to— the other. It creates hierarchical structures where the parent objects (the containers) contain child objects (the containees). Conceptually, the lifespan of the containees is contingent on that of the container: when the container is deleted, so are the containees. The *Weighted Hierarchy* SMS exploits the support of composition (a restrictive form of association) in many DSLs to visualize recursive containment relationships, where each element can be attributed a weight.

Intent The *Weighted Hierarchy* SMS aims to understand (multi-level) hierarchies emerging from containment relationships, and the relevance of their elements. The main sensemaking tasks it supports are "*What is the structure of the examined component?*" and "*What is the relevance of the components?*".

Presentation metaphor This is a structural-based SMS. The other SMS that belongs to this group is Connectivity, which relaxes the composition to an association.

Context meta-model The strategy fuses the concepts of containers and containees into a single class, HierarchicalElement (cf. Figure 5.7). These have a name and an optional numeric weight, which is the children collection's size by default. What characterizes the SMS is that hierarchical elements are related to themselves via a children composition, supporting recursive containment relationships.

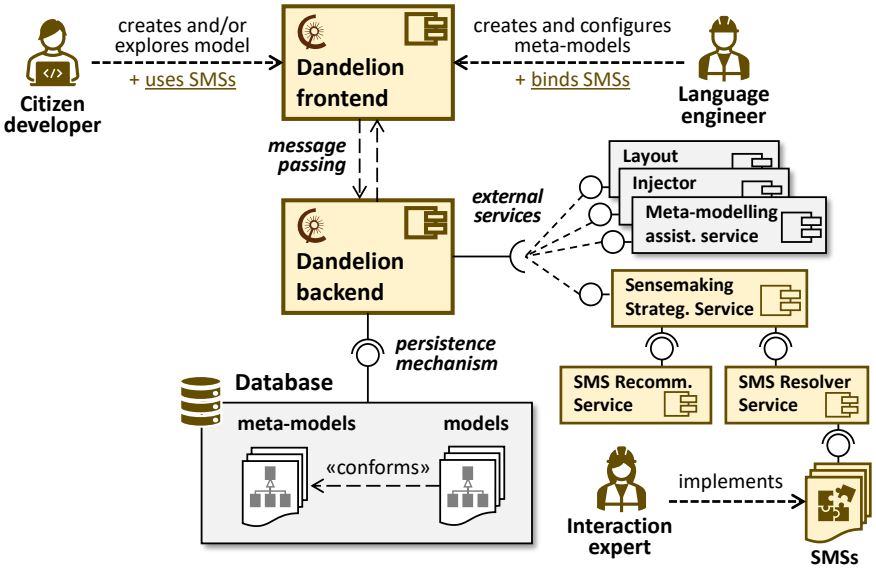


Figure 5.8
Extending Dandelion's architecture to support SMSs.

Visualization variants The SMS can be visualized with Treemaps spanning multiple levels.

Properties The SMS introduces a title property.

Motivating example This SMS can be used to understand the structure of (meta-)models. Figure 5.7 depicts an agnostic application of the strategy on Dandelion's linguistic meta-model (cf. Figure 4.2). The figure shows a fragment of it, displaying SemanticNode (Dandelion's concept representing both classes and objects) and ObjectProperty, which is the concept representing both references and links. The binding maps HierarchicalElement to SemanticNode, and children to an expression that obtains the target SemanticNodes of each composition relation. This SMS application can then be used to analyze arbitrary (meta-)models, such as particular UGROUND ROSE models, as illustrated in the figure.

5.5 Architecture

The architecture of Dandelion presented in Section 4.2.1 has been extended to support SMSs, as shown in Figure 5.8. In particular, with the incorporation of support for SMSs, language engineers and citizen developers can bind and use SMSs, respectively. SMSs are created by an interaction expert.

SMSs are integrated into Dandelion through the Sensemaking Strategies Service. Dandelion already supports a series of external services, such as a layout service to arrange graphical elements on the canvas; injectors, to

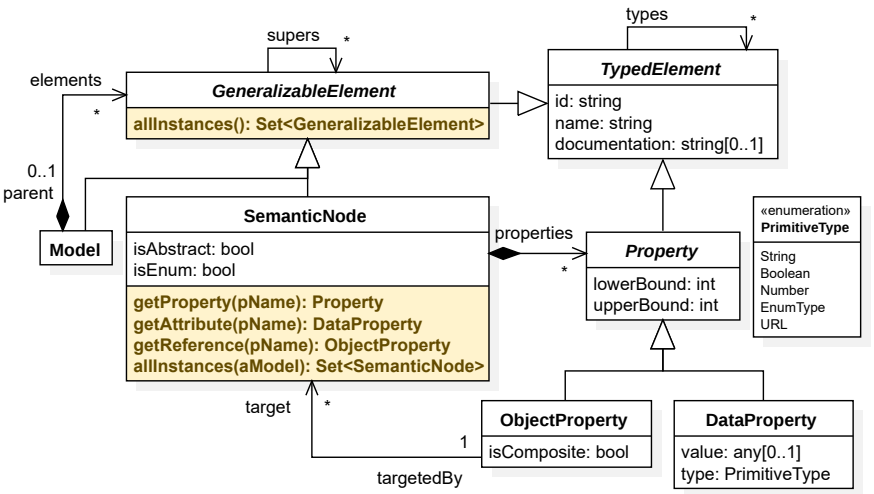


Figure 5.9 Dandelion's linguistic meta-model, with operations for derived bindings.

import/export models from/to other formats, such as EMF [26]; or meta-modeling assistants, like Droid [197], which recommends attributes and references when building a meta-model. The new service is split into a Recommendation Service, serving the SMS recommender and a Resolver Service, to execute SMSs.

The SMS recommender takes as input an SMS's context meta-model and a target meta-model (Figure 5.4) and finds the set of sound bindings between them. Finding such bindings is an instance of the NP-hard subgraph isomorphism problem [223], which is computationally demanding. To avoid a combinatorial explosion of recommendations, the recommender only considers the bindings of classes, and the binding of attributes and references is delegated to the user. Depending on the feasibility of these partial bindings (i.e., whether every attribute of the context meta-model can be bound to, at least, one attribute of the target meta-model), they are tagged perfect or incomplete. The latter are also useful, as users can still populate the non-bindable attributes with derived values through the dedicated expression language.

The SMS resolver interprets SMS applications and yields interactive diagrams. It requires identifying the type of SMS, a binding to a target meta-model, the values of the SMS properties, and a visualization variant choice. The catalog of SMSs that Dandelion presents can be extended by an interaction expert.

The execution of the resolver comprises three phases: data extraction, algorithmic preprocessing, and construction of the visual representation. First, all the relevant model elements dictated by the binding are retrieved. This task is delegated to an internal interface, IDandelionRepository, to exploit Dandelion's integration with different data providers.[1] If there are

[1]This work follows a very similar approach to the Epsilon Model Connectivity Layer:

derived attributes, they are evaluated here. Next, the resolver may perform an algorithmic preprocessing step, if needed. For example, metric boxplots require computing descriptive statistics, while categorical strategies have to aggregate data. The last step is constructing a visual representation (e.g., bar charts or boxplots) with the data obtained in the previous steps. Typically, SMSs expose multiple visualization variants, which are populated in this step. Two popular libraries are employed for this task: D3.js [224] and Apache ECharts [225]. As this step is decoupled from the rest of the resolver, integration of new visualization libraries is eased.

Property bindings can be derived thanks to a dedicated expression language. This language is implemented by extending Dandelion's linguistic meta-model (Section 4.2.2) with additional operations (Figure 5.9). In particular, all the attributes (e.g., TypedElement.name) are accessible via getters, and navigability is supported. SemanticNode implements a crucial operation for derived bindings: allInstances(), which returns all the instances of a given SemanticNode or Model, allowing thus the extraction of information from lower levels. Additionally, SemanticNode overloads allInstances() to obtain the node's instances within a particular model. Finally, this expression language has been implemented using TypeScript. Consequently, it supports (higher-order) functions such as map or filter; and length for the size of a collection.

5.6 Tool support

SMSs have been integrated into Dandelion via the strategies dashboard: a panel shown next to the modeling canvas (cf. Figure 5.10).[2] Strategies support different operations: maximizing to a full-screen view, switching visualization variants, and forcing a refresh. They can also be rearranged via drag and drop in the dashboard.

The creation of an SMS takes place in a multi-step wizard Figure 5.11. First, a suitable SMS and a visualization variant are selected. To guide the decision, the number of suitable bindings is shown next to each SMS (coming from the SMS recommender), as shown in Figure 5.11a. Next, the wizard displays a description and the context meta-model of the SMS together with all the suggested (available) bindings. The last step is completing the selected partial binding —possibly with derived values— and setting values to the appropriate properties of the SMS, as shown in Figure 5.11b. The strategy is then added to the dashboard.

`https://www.eclipse.dev/epsilon/doc/emc`.

[2]Edges in the modeling canvas are hidden for clarity.

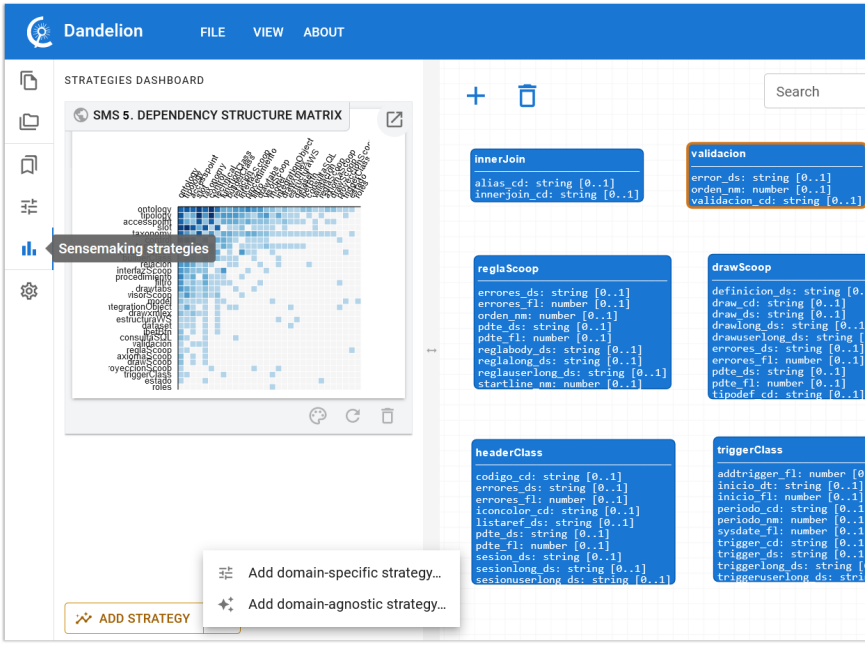
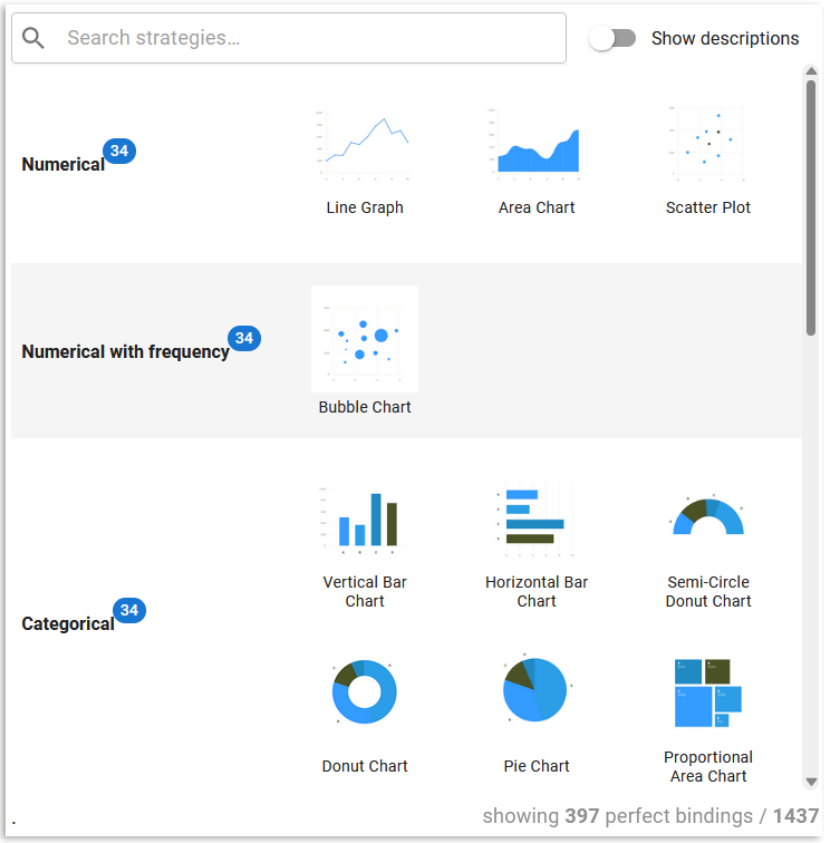


Figure 5.10
ROSE meta-model in Dandelion, showing a strategies dashboard with a dependency structure matrix SMS.

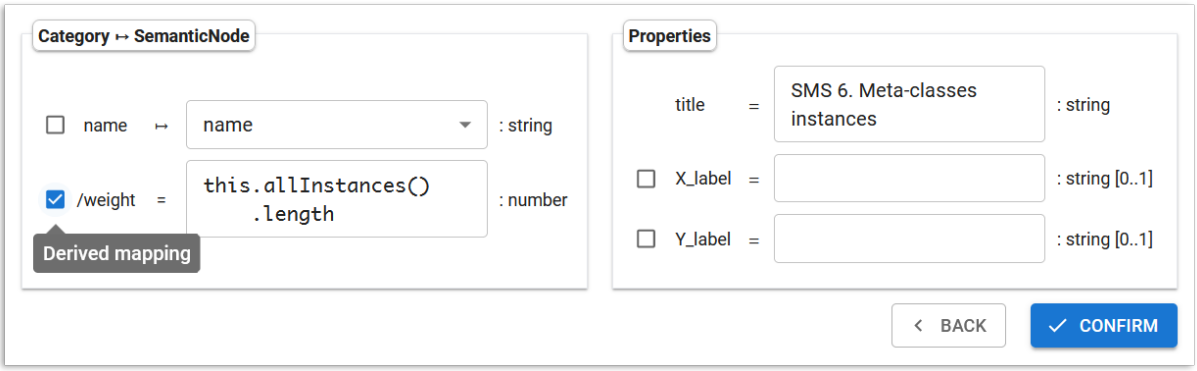
5.7 Summary and conclusion

This chapter has introduced the notion of *model sensemaking strategy* (SMS), a reusable, generic component for creating flexible, purposeful visualizations of large and complex models. SMSs are instantiated via bindings to either domain-specific or linguistic meta-models, enabling their application both to individual DSLs and to entire modeling ecosystems. The chapter has detailed their integration into Dandelion, presented a catalog of 10 strategies with 20 reusable visualization variants, and described a recommender system that helps identify suitable bindings for a given meta-model. The approach also supports derived bindings via an expression language, further enhancing its flexibility. SMSs will be evaluated in Section 8.2, showing that the proposed tooling is scalable, effective for understanding large modeling ecosystems, and suitable as a foundation for building and operating low-code platforms in industrial settings.

The next chapter presents Dandelion+, which extends Dandelion from a graphical language workbench to a low-code engineering platform that manages both the structure and the behavior of low-code applications on a model-driven foundation.



(a) SMSs recommendation panel.



(b) Configuration wizard for a domain-agnostic SMS. Left: binding Category to SemanticNode using a derived mapping. Right: properties editor.

Figure 5.11 Multi-step wizard for creating an SMS in Dandelion.

Chapter 6

Modeling Low-code Platforms: Structure and Behavior

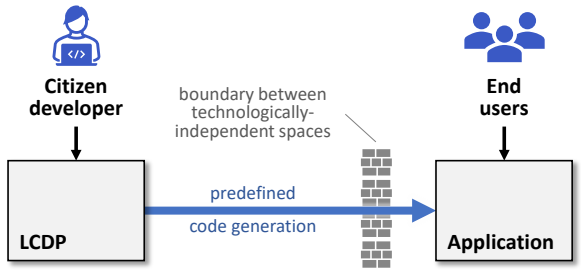
This chapter presents Dandelion+, a model- and workflow-driven approach for engineering low-code platforms, with a focus on capturing application structure and behavior. The chapter relies on a running example (6.2), which is used to motivate the approach (6.3). The core of the chapter explains how to model the structure (6.4) and behavior (6.5) of low-code platforms. Finally, the architecture and tool support of Dandelion+ are presented in 6.6. This work has been published in a journal article [33].

6.1 Introduction

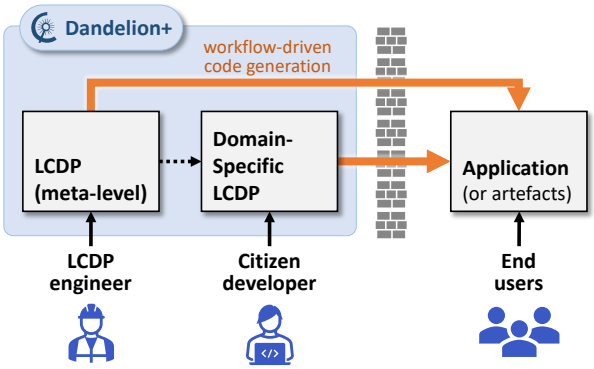
As outlined in Chapter 1, society increasingly demands more software, delivered faster. However, the available workforce of professional developers cannot keep up with this demand, and many end users lack formal training in programming to build their own solutions. To address this issue, two complementary approaches have emerged: Model-Driven Software Engineering (MDSE) and low-code development. As discussed in Section 2.3, MDSE has been proposed to enhance productivity by increasing the level of abstraction and enabling automation [18], typically targeting professional developers. Conversely, Low-code Development Platforms (LCDPs) aim to empower citizen developers by providing user-friendly (often graphical) languages and tooling to carry out programming tasks and build applications [32].

Many LCDPs exist today, including OutSystems [61], Mendix [169], and Microsoft Power Apps [171]. They follow the development process depicted in Figure 6.1a. First, citizen developers use an LCDP to build the blueprints of an application. Then, a predefined code generator is executed to generate the application. Finally, the application is deployed to the cloud, becoming available to end users. However, since most platforms are proprietary and

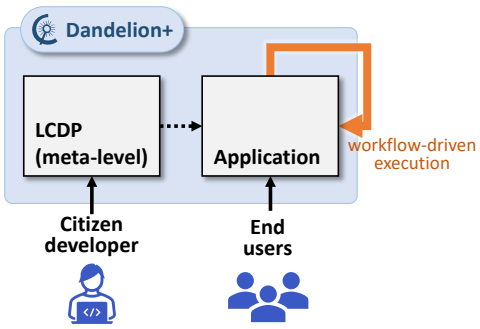
Figure 6.1 Approaches for building applications within low-code environments.



(a) The regular LCDP approach, using a predefined code generator.



(b) Dandelion+ approach, enabling the construction of domain-specific LCDPs by replacing the fixed generation step with domain-specific modeling and workflow execution.



(c) An alternative approach, also supported by Dandelion+, that does not depend on code generation but directly runs the final application within the platform.

closed-source, the generation step is usually fixed and platform-specific, hindering interoperability and causing vendor lock-in.

As discussed in Section 2.3, LCDPs can benefit from adopting MDSE techniques, including modeling and meta-modeling (to describe application structure), as well as model transformations and code generation (to describe application behavior). On the one hand, explicit meta-models for LCDPs would enable interoperability. On the other hand, explicit meta-modeling support could overcome the limitation of most LCDPs today, so that they could be used to build domain-specific LCDPs (e.g., low-code platforms for defining mobile apps, games, or data science workflows [226]), which can then generate apps (or artifacts) for the end users via code generation (cf. Figure 6.1b). As Figure 6.1c shows, code generation is not the only way to define applications. Instead, an MDSE-enabled LCDP can rely on modeling and transformations to directly define final applications and run them within the same environment without generating code.

To support these scenarios involving the construction of LCDPs and applications, merely porting MDSE technologies to the web may be insufficient. On the one hand, specific low-code concepts should be incorporated into structural (meta-)modeling languages (e.g., roles, users, platforms, or files). On the other hand, model management operations within LCDPs should be orchestrated using dedicated languages that also incorporate low-code concepts, such as managing entities or consuming external APIs.

To tackle these issues, this thesis proposes Dandelion+, a cloud-, MDSE-based low-code platform that can be used to build domain-specific LCDPs and final applications. Dandelion+ is founded on MDSE principles, featuring an agnostic meta-modeling language that is complemented by concepts specific to low-code. To describe the behavior of LCDPs and their resulting applications, Dandelion+ offers a workflow language —called PlatFlow— that orchestrates both MDSE tasks (e.g., model validation, model-to-model, or model-to-text transformations) and low-code-specific tasks (e.g., managing entities or calling APIs). This chapter presents the rationale behind Dandelion+, along with its architecture and tooling. To validate the proposal, an evaluation will be presented in Section 8.3, where Dandelion+ is confronted with other low-code tools and extant industrial approaches to develop low-code platforms.

The contributions of this chapter are:

1. An MDSE-based architecture for low-code development, which can be used to define both domain-specific LCDPs and applications.
2. PlatFlow, a workflow language designed to drive the behavior of low-code applications.
3. Dandelion+, a tool that realizes the previous ideas.

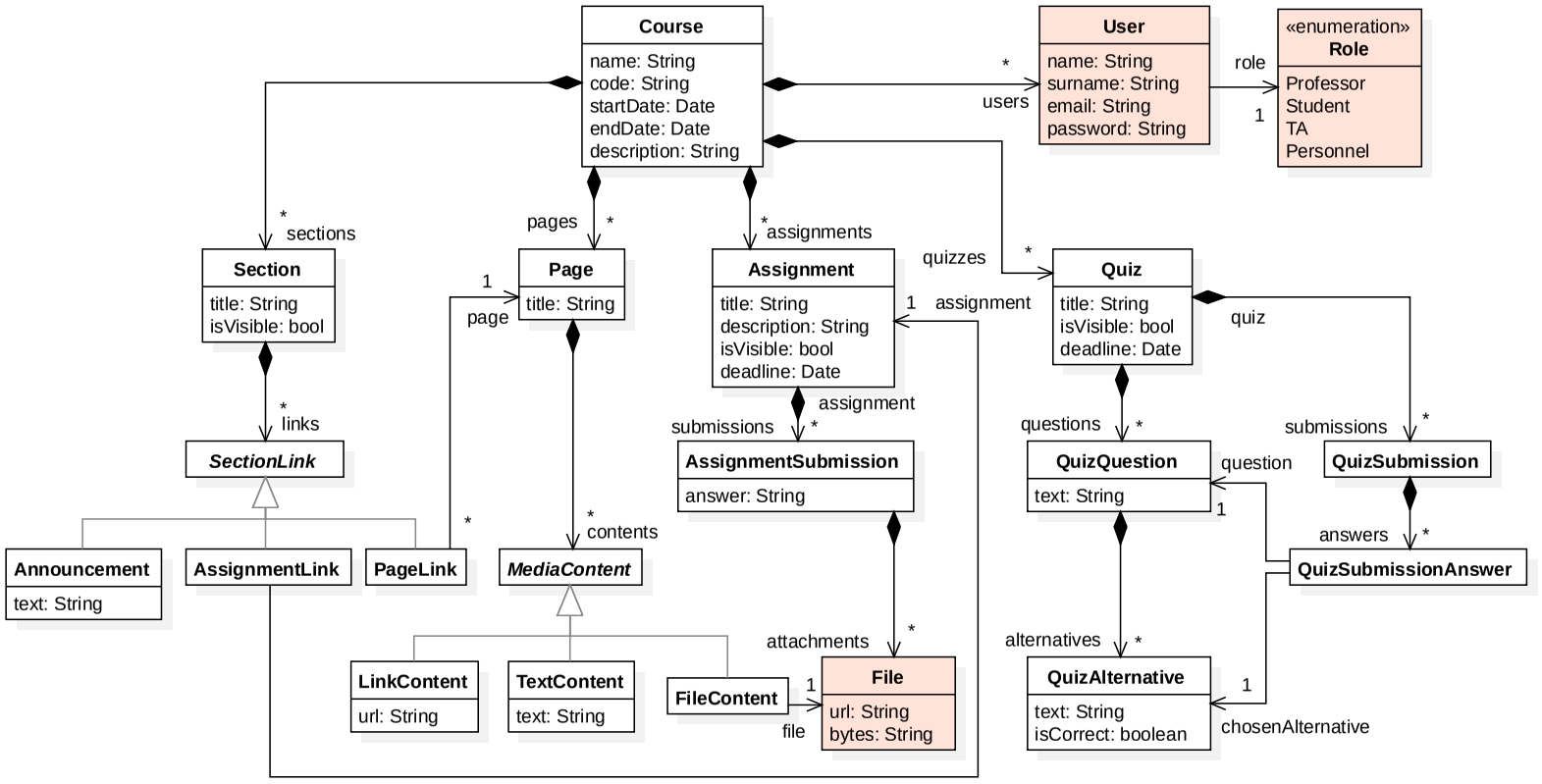


Figure 6.2 Class diagram of *Mudel*, a fictitious learning management system. Problematic classifiers are highlighted.

6.2 Running example

This section presents the running example and identifies the limitations encountered in its development when using current low-code approaches.

The domain of the running example is learning management systems (LMSs) —platforms where academic activities are carried out. Some of the most prominent examples include Moodle [227] and Canvas [228]. These platforms typically feature activities and learning tools (e.g., quizzes and assignments), interactive spaces for lecturers and students (e.g., announcements or discussion forums), and academic management facilities (e.g., uploading and downloading documents, grading quizzes, or conducting surveys) [229].

'Mudel' is a fictitious LMS that will be used as a running example throughout this chapter to illustrate the concepts and mechanisms of Dandelion+. Mudel supports common features of LMSs, including managing courses where announcements can be posted and learning material can be linked, creating and submitting assignments, and designing and answering quizzes.

Figure 6.2 proposes a possible class diagram that captures the structure of Mudel. The central concept is the Course, in which Users can be enrolled with different roles, namely: professors, students, teaching assistants (TAs), and external personnel. The main page of each course is divided into Sections, which may contain Announcements posted by professors and may link pages and assignments designed for the course (i.e., AssignmentLinks and PageLinks). Pages contain text (TextContent) as well as other media such as links to external web pages (LinkContent) or file attachments (FileContent), making them appropriate for lectures or notes. Professors can propose exercises to students through Assignments, and students can submit their work via an AssignmentSubmission, which may enclose file attachments. Professors can also create Quizzes with Questions, each with several QuizAlternatives in turn. Students can answer them via QuizSubmissions, which choose one alternative per question. Finally, sections, assignments, and quizzes can be hidden or made visible with the isVisible property.

While this class diagram adequately captures the essence of the domain of LMSs, the following issues in standard domain modeling languages (e.g., UML class diagrams) can be identified when it comes to their use within LCDPs:

I1. Access control is insufficiently modeled. Low-code platforms need access control to restrict the actions that users can perform. However, although Roles are modeled in Mudel, their permissions are not. As a consequence, access control is insufficiently described and cannot be enforced. This leads to undesired scenarios, such as students creating

courses, or modifying submissions of peer students. Note that typical domain modeling languages (e.g., Ecore or UML class diagrams) do not consider access control. Even the use of integrity constraint languages such as OCL [209] would not solve the issue, and may lead to *ad hoc* solutions. Therefore, mechanisms that reify roles and permissions [230] are required to enforce access control over the elements of the domain model.

I2. Ownership is not modeled. In regular modeling languages, ownership is not modeled; i.e., *who* creates an entity is not relevant. However, ownership is crucial in some domains developed with low-code platforms. The reason is that users themselves may be responsible for managing entities, and therefore it is important to discern who owns a given entity. In the case of Mudel, modeling the ownership of submissions is necessary to know who made them and to limit students' access to only their own. Therefore, platforms must keep track of the ownership of the entities defined therein, expose programmatic interfaces to manage entities' ownership, and offer access control mechanisms that exploit this information.

For clarification, the diagram in Figure 6.2 could be improved with associations between User and some classes to model ownership (e.g., the professor creating a Course or the student submitting an AssignmentSubmission). However, these ownership relations and the involved classes (e.g., User) would require a dedicated treatment in the modeling language to convey their special semantics to the low-code platform.

I3. Equivalent platform- and domain-specific entities are not bridged appropriately. LCDPs manage a constellation of entities, including pages, users, data schemas, and the data itself. These entities can be divided, in broad terms, into platform-specific and domain-specific entities. The former includes pages and configurations about the platform, while the latter includes entities that belong to the domain of the DSLs (e.g., Course or Quiz in Mudel). However, these sets are not disjoint: the platform manages some entities, which DSLs may also include to relate them with other domain concepts. This is the case of User, Role, and File: while these concepts are clearly platform-specific (i.e., they are central to the functioning of any low-code platform), they may also belong to some DSLs. For instance, in Mudel, assignment submissions may attach files that the platform should handle especially and appropriately. However, as it is, the class diagram in Figure 6.2 does not account for this connection. In particular, users and roles should not be secluded within DSL definitions, as the platform must manage them for authentication (i.e., logging in) and authorization (i.e., access control). Similarly, files can be part of the low-code platform definition itself as files with code snippets for its configuration. In conclusion, modeling languages for low-code development must provide mechanisms

that bridge domain-specific entities with some platform-specific ones, such as users, roles, and files, avoiding entity duplication and ensuring correct management.

I4. Behavior is not modeled. The running application consists of pages that typically contain forms and buttons that perform actions upon submitting or clicking on them. For instance, a student may post a question in a forum, and a professor may grade a quiz. However, the class diagram alone does not capture this information. Therefore, it is essential to provide mechanisms to model behavior, possibly through high-level workflows.

I5. Applications are not modeled. Besides access control, this (meta-)model is insufficient to generate a running LMS without assumptions. For instance, the LMS will contain non-obvious pages such as the one presenting quizzes together with their questions and alternatives. Therefore, there is a need to model pages and to display different entities and other elements of the application.

Moreover, the following issues can be identified when it comes to the development of Mudel in conventional LCDPs:

I6. Facilities for data manipulation and artifact generation are insufficient. As discussed in Table 3.3 from Section 3.3.3, most LCDPs in the market support some (meta-)modeling facilities. However, they lack mechanisms to manipulate the data they manage and to produce artifacts from it using model transformation and code generation, respectively. This directly refers to the scenario depicted in Figure 6.1b. Mudel has several realizations of this scenario. For instance, professors may generate a static site from the sections of a course, generate code for other tools, or export quizzes using a Moodle-specific format for exchanging quizzes, such as the *Moodle XML format*,[1].

I7. Importing/exporting mechanisms are not extensible. LCDPs support different import/export mechanisms. For instance, some LCDPs can manipulate JSON documents while others, like OutSystems [61], integrate with Excel spreadsheets. However, the support is limited given the lack of data manipulation facilities (I6). This disparity of capabilities can hinder the adoption of a particular LCDP, and produce vendor lock-in. For example, this would preclude importing quizzes in Moodle XML format into the example platform. Therefore, it is necessary to provide extensible mechanisms for importing and exporting data.

[1] `https://docs.moodle.org/405/en/Moodle_XML_format`

Although issues I1–I7 have been identified in the domain of LMSs, they are general enough to be extrapolated to other domains and are likely to affect development in any LCDP. The following sections present an approach to overcoming these limitations, using Mudel as an illustration.

6.3 Approach

The approach of Dandelion+ builds upon Dandelion (Chapter 4) by broadening its scope from language definition to platform engineering. In particular, while Dandelion focuses on defining the static aspects of DSLs (i.e., their abstract and graphical concrete syntax), Dandelion+ provides the necessary infrastructure to manage users, permissions, and executable workflows. Consequently, Dandelion+ is equipped to host and execute full-blown applications.

6.3.1 Design principles

Dandelion+ is based on the following design principles:

P1. Dandelion+ has a model-driven foundation. As discussed in Section 2.3, the adoption of MDSE techniques into low-code platforms brings benefits, including overcoming the issues of scalability (i.e., supporting many entities), openness (i.e., supporting different formats, enhancing interoperability and preventing vendor lock-in), and heterogeneity (i.e., supporting artifacts expressed in different formats). Dandelion+ follows a model-driven approach to design DSLs that capture platform domains, as well as the models that are instances of these DSLs.

P2. Dandelion+ has a uniform ecosystem with a reflective interface. Low-code platforms work at different levels of abstraction, including platform definition (e.g., users, roles, or the platform itself), DSL definition (i.e., meta-models), and DSL usage (i.e., models). When developing complex applications, it is common to encounter scenarios that involve the whole modeling ecosystem and permeate all these levels of abstraction (e.g., dashboards). Therefore, a *unified* interface that bridges all these levels is desirable. Moreover, applications may be *reflective*, i.e., they may modify the platform itself at runtime.

Dandelion+ achieves reflectiveness by explicitly modeling each aspect of the platform, and enabling its runtime manipulation via model management languages. Technically, Dandelion+ relies on the Eclipse Epsilon family of languages for this purpose (cf. Section 2.1.8). Another benefit of using Epsilon throughout the platform is persistence decoupling. LCDPs in the market have different persistence mechanisms, potentially yielding vendor lock-in.

Although Dandelion+ provides an implementation in Elasticsearch [123], the persistence layer is handled by the Epsilon Model Connectivity Layer (EMC). This layer acts as an intermediary, making the platform persistence-agnostic and shielding entities and logic from underlying storage mechanisms.

P3. Platforms are structured using a low-code-specific linguistic meta−model. Issues I1–I3 show that a standard domain modeling language (e.g., UML) is not expressive enough to handle key aspects of low-code platforms such as access control, ownership, and bridging domain and platform entities. Dandelion+ addresses these issues by relying on a linguistic meta-model tailored to low-code platforms. In particular, the structure of the platform is managed through a *platform* meta-model, ownership and entity bridging are managed through a bespoke *domain modeling* meta-model, and access control is managed through a *roles* meta-model.

P4. Application behavior is implemented using workflows. As stated in issues I4 and I5, defining applications and their behavior is essential in any non-trivial low-code platform. In Dandelion+, applications are modeled using an *applications* meta-model, and their behavior is described using PlatFlow, a platform-aware graphical language to define and configure domain-specific applications using workflows. In order to improve the experience of citizen developers, workflow development takes place in the browser, sparing the need for installing desktop applications or plugins. Workflows include nodes for different operations, like model transformation, code generation, and integration with external services. PlatFlow works on top of Epsilon (cf. principle P2) to shield the implementation from technology specificities. The components of PlatFlow enable data manipulation in a systematic and controlled way (I6). To address I7, PlatFlow allows importing and exporting data in different formats via serializers (drivers) that leverage the Epsilon Model Connectivity Layer (EMC). Moreover, Dandelion+'s domain modeling meta-model allows the homogenization of data in different formats, thereby ensuring interoperability.

6.3.2 Top-level overview

Figure 6.3 presents a top-view of the approach. Every instance of Dandelion+ contains several platforms, which are isolated development environments. This allows their use by different tenants without interfering with each other, e.g., a single installation can serve multiple customer-specific platforms in a SaaS scenario. Each platform manages its users, whose permissions are determined by their roles. Platforms also define DSLs, which can be instantiated in models, just like in standard MDSE. Furthermore, platforms can define complex applications, which are driven by workflows that can

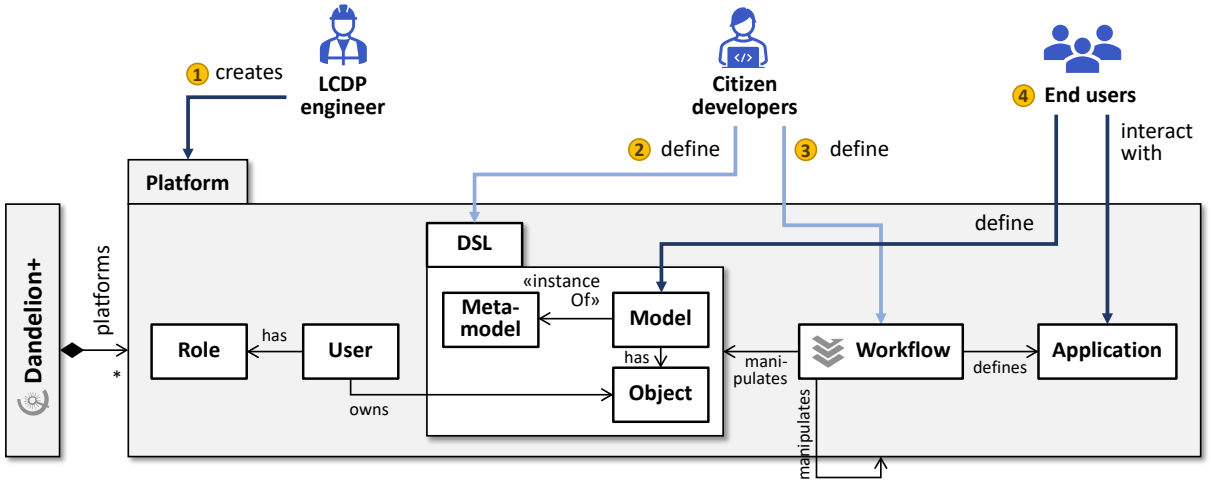


Figure 6.3 Top-level structure of Dandelion+ and interactions with stakeholders.

either manipulate or make use of DSLs and models, as well as the entire platform.

Dandelion+ supports three main stakeholders, who incrementally build the low-code platforms. The steps are numbered in the figure. First, the LCDP engineer creates a platform (1). Automatically, an *Administration* role is created, together with an administrator user, along with predefined components for its operation. Additionally, the LCDP engineer can register more roles and users. Then, those citizen developers registered by the LCDP engineer can authenticate into the platform and create domain-specific languages (2). To do so, they can register meta-models into the platform either using a textual syntax or via an integrated graphical meta-model editor. Then, citizen developers can create workflows (3). Finally, end users can either modify models manually or interact with applications (4). The experience of both citizen developers and end users is shaped by their assigned roles.

The next sections describe the two main ingredients of low-code development in Dandelion+. Section 6.4 presents the approach to defining the structure of low-code platforms (i.e., platforms, DSLs, roles), whereas Section 6.5 details the infusion of behavior into applications using a workflow language.

6.4 Structure of low-code platforms

This section presents the approach followed to design the structure of low-code platforms. To do so, the entities encompassing low-code platforms and their relationships are formalized following a model-driven approach in a

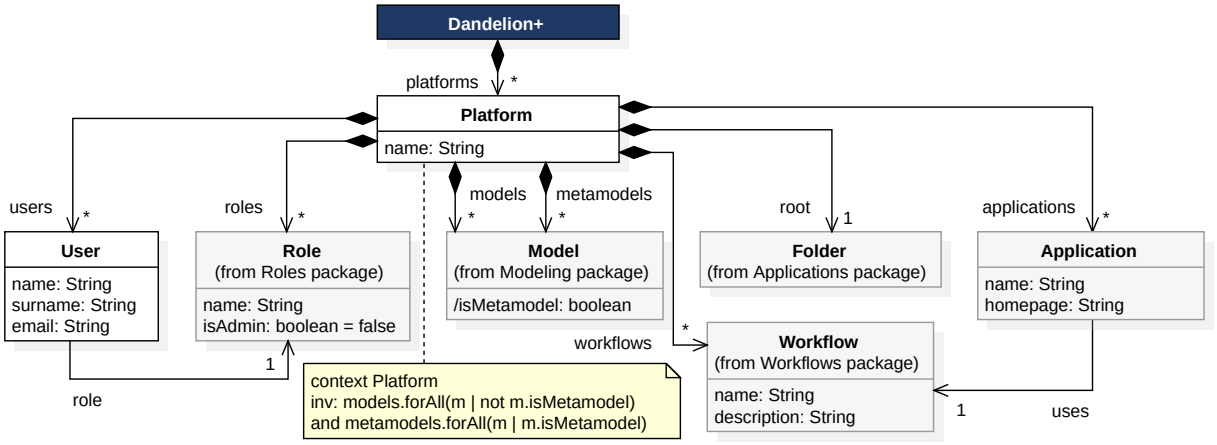


Figure 6.4 The platform package.

linguistic meta-model that satisfies the principles detailed in Section 6.3.1. The meta-model is broken down into packages, which are examined in the next subsections. In particular, the core of this meta-model is the platform package (6.4.1), which uses the rest of the packages: the domain modeling package (6.4.2), which allows defining DSLs within platforms; and the roles package (6.4.3), which permits defining user permissions.

6.4.1 Platform meta-model

The platform is the topmost concept in Dandelion+. Everything in Dandelion+ is contained within a platform. As such, platforms provide independent and isolated development environments that coexist under one instance of Dandelion+.

Figure 6.4 presents the platform package of Dandelion+'s linguistic meta-model. Platforms have a name and encompass several concepts. In particular, platform stakeholders are reified as Users, allowing their authentication therein. The experience of these users within the platform is shaped by their assigned Role. These same users can define and work with DSLs, whose construction platforms support through meta-models and their instances (both can be represented using Models, as explained later). DSLs can be managed in a low-code fashion using Workflows, which can be promoted to full-blown Applications available for the end users. Finally, platforms can store Files, which can contain executable code (e.g., for defining specific parts of workflows) or any other type of static content, such as images or documents. For this reason, each platform points to the root Folder of its associated file system.

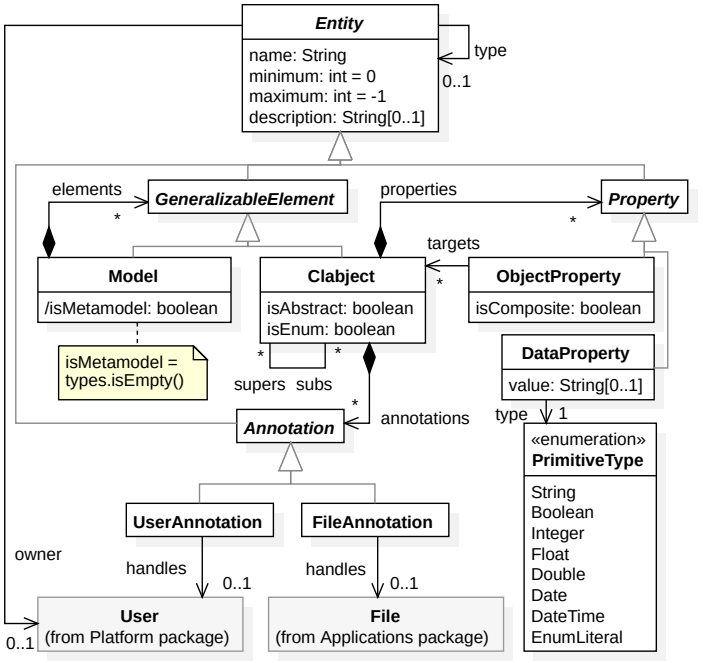


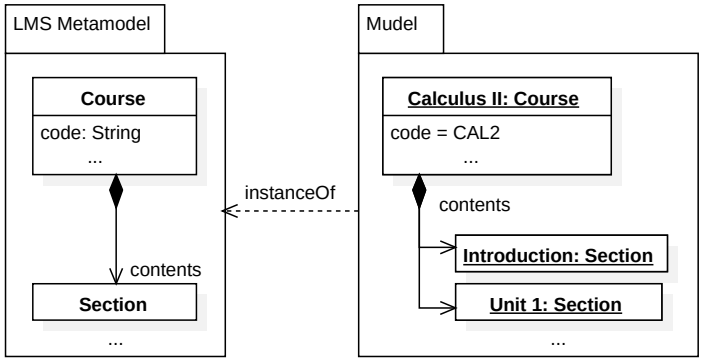
Figure 6.5
The domain modeling package.

6.4.2 Domain modeling meta-model

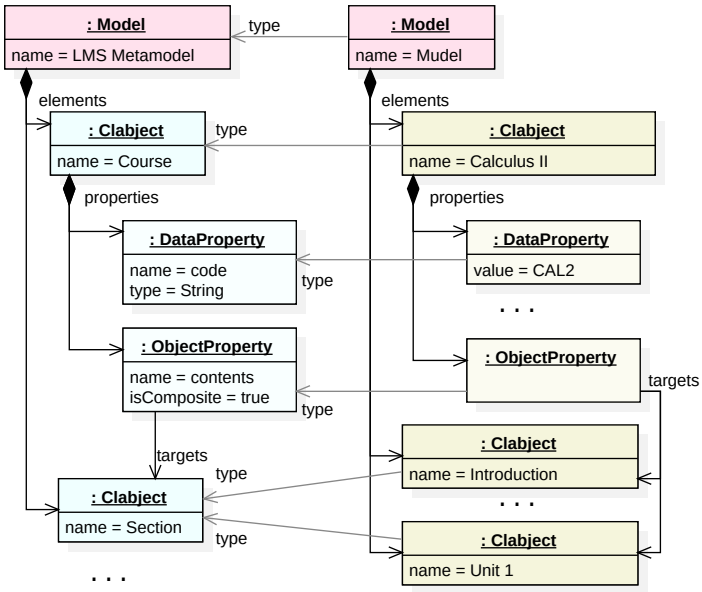
Dandelion+ uses a domain modeling meta-model to represent domains and define DSLs. The definition of these DSLs is specified in meta-models, and their instances are represented in models. This meta-model is a continuation of the harmonizing meta-model designed for Dandelion (Figure 4.2), which has been adapted and augmented to cater to the requirements of operational low-code platforms.

Figure 6.5 presents the domain modeling meta-model. The central concepts are Model and Clabject. On the one hand, Models reify both models and meta-models. Their elements are objects and meta-classes, respectively. On the other hand, Clabjects are used to represent these objects and meta-classes, populating or defining properties, respectively (see Section 2.1.5 for the rationale behind this nomenclature). Properties can either be encoded through DataProperty, for primitive types, or ObjectProperty, for associations or containments. Models, clabjects, and properties are all subclasses of Entity, which has a name, may have a description, and may be typed. Entities also define minimums and maximums, whose semantics depend on the concrete realization: in clabjects, they constrain their number of instances, whereas, in properties, they define multiplicities.

Figure 6.6 exemplifies how to express a part of Mudel's meta-model in Figure 6.2 and how to instantiate it using the modeling meta-model. Specifically, it represents the LMS meta-model by defining concepts like Course and Section, as well as an instance named *Mudel* of said meta-model



(a) An excerpt of Mudel's meta-model and an instance thereof.



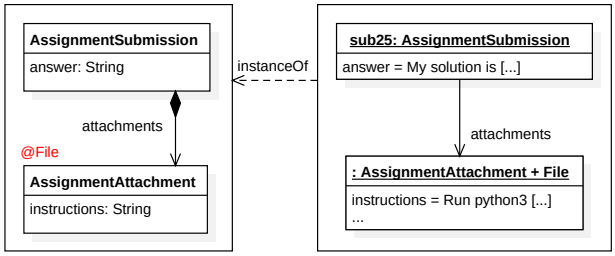
(b) Representation using the domain meta-model.

Figure 6.6
Expressing a meta-model and one of its instances using the modeling meta-model.

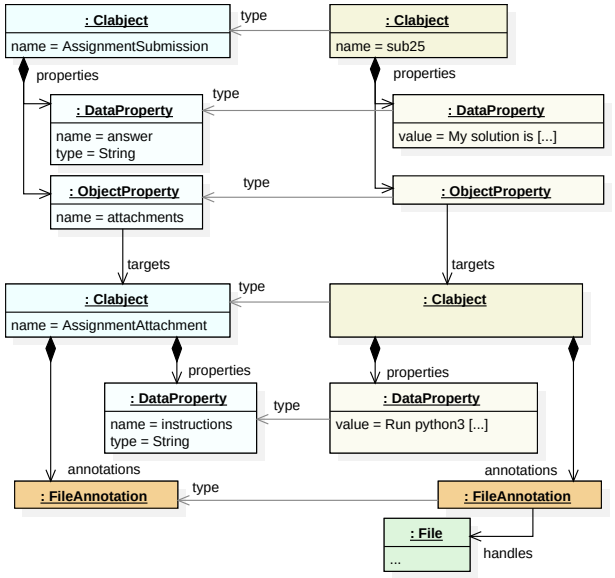
that contains a Calculus II course with some sections. Please note that instantiation is symmetrical: Models (meta-models/models) contain Clabjects (meta-classes/objects) which, in turn, define/populate Property elements. Type conformance is preserved thanks to the Entity.type relationship.

One of the issues identified in the running example is that users and files must be modeled differently to be usable from a platform standpoint while being expressed normally in the meta-model as meta-classes. The approach followed is to decorate the impacted meta-classes with Annotations, considering the case of users and files with UserAnnotations and FileAnnotations, respectively. Both meta-classes and objects can be annotated. In particular, objects of an annotated meta-class point to the concept (i.e., user or file) they handle.

Figure 6.7 Supporting files in a meta-model using annotations from the domain modeling meta-model.



(a) An excerpt of Mudel's meta-model showing the use of annotations for files.



(b) Representation using the domain meta-model, benefiting from annotations.

Figure 6.7 demonstrates how to apply these annotations to represent AssignmentSubmissions and Files from the running example. The meta-class AssignmentSubmission is represented as a Clabject, with a data property for its answer and an object property for the attachments. Files are instead handled differently. In Dandelion+, it makes sense to promote them to a meta-class (i.e., AssignmentAttachment) annotated with a FileAnnotation. The objects that instantiate this meta-class follow the regular procedure for instantiation, but are also annotated with FileAnnotations that point to files. With this approach, the expressivity of the initial class diagram is maintained (i.e., files are part of the domain language), the platform can handle files correctly, and additional properties over users or files can be defined (e.g., AssignmentAttachment.instructions).

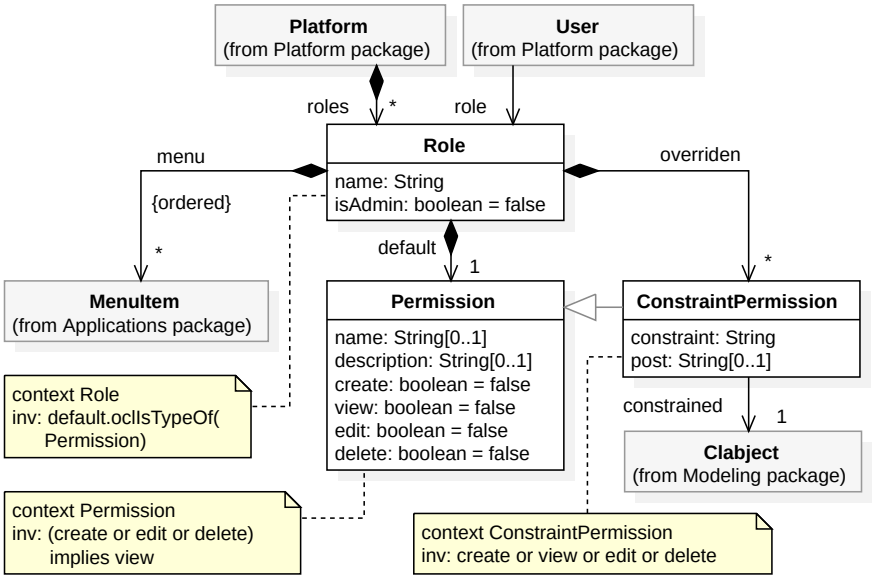


Figure 6.8
The roles package.

6.4.3 Roles meta-model

Stakeholders participate in low-code platforms with different goals and scopes of action. Therefore, implementing access control is crucial. Dandelion+ manages this through roles.

Figure 6.8 presents the roles meta-model. Each platform defines a series of Roles, which are assigned to Users. Roles have a name and can be administrators, who are granted full control to modify the platform, including managing roles and users. Each role is associated with a series of applications, which constitute its menu in the browser. A role declares Permissions, which determine whether the creation, query, editing, or deletion of entities is allowed (i.e., create, view, edit, and delete). Specifically, every role must define a default permission, which is applied to all the meta-classes in the system. This behavior can be superseded via ConstraintPermissions through Role.overridden. In particular, ConstraintPermissions are linked to the meta-class they target (constrained), and define two fields: constraint and post. The former expects an EOL Boolean predicate indicating whether the permission should be applied. That is, whenever an object of type constrained is found, constraint determines whether the object is made available or not. The latter expects any EOL code and allows modifying objects immediately after retrieval. This mechanism can be leveraged for convenience to perform transient, local model transformations. In both fields, variables 'this' and 'currentUser' are available, referring to the object being queried and the user performing the action, respectively. If a clabject type is pointed to by several overridden ConstraintPermissions, Dandelion+ only grants those permissions (i.e., create, view, edit, delete) for which no

constraint evaluates to false.

Table 6.1 summarizes Mudel's roles and their permissions following this meta-model. In particular, professors can do everything except manage students' submissions; students can only view content on the platform, as well as create and edit their submissions; TAs can modify courses and their contents; and personnel has access to all the information therein but in a read-only manner.

Table 6.1
Roles and permissions of Mudel.

Role	Create	View	Edit	Delete
Professor · *(Default)*	•	•	•	•
· AssignmentSubmission	○	•	○	○
· QuizSubmission	○	•	○	○
· QuizSubmissionAnswer	○	•	○	○
TA · *(Default)*	○	•	○	○
· Course	○	•	•	○
· Section	•	•	•	•
· Page	•	•	•	•
· AssignmentSubmission	○	○	○	○
· QuizSubmission	○	○	○	○
· QuizSubmissionAnswer	○	○	○	○
Student · *(Default)*	○	•	○	○
· AssignmentSubmission	•	•	•	○
· QuizSubmission	•	•	•	○
· QuizSubmissionAnswer	•	•	•	○
Personnel · *(Default)*	○	•	○	○

LEGEND: • granted, ○ not granted.

However, this permission specification is too coarse-grained; the domain of Mudel is more restricted. For instance, the additional restrictions gathered in Table 6.2 apply to the student role. They restrict an action (in italics) over a meta-class given a condition (i.e., a predicate). These more refined restrictions can be modeled using ConstraintPermission: the permission is enforced if and only if the constraint is satisfied. Moreover, the post field can be used to modify the object in place after retrieval.

Figure 6.9 shows the implementation of some of these restrictions for the student role using the roles meta-model. In the model, the default permission enables viewing everything. Then, ConstraintPermissions override this general permission for particular clabject types. In particular, R1 enables viewing only available Sections (those making this.isVisible true); R5 restricts the view to owned AssignmentSubmissions; R8 enables editing within the deadline; and R4 hides the correct alternatives (i.e., the solutions of the assignment) via a post attribute. Please note that these modifications are not

#	**Meta-class**	**Predicate** ("can only be...")
R1	Section	*viewed* iff they are visible.
R2	Assignment	*viewed* iff they are visible.
R3	Quiz	*viewed* iff they are visible.
R4	QuizAlternative	*viewed* with isCorrect hidden.
R5	AssignmentSubmission	*viewed* if the object is owned.
R6	QuizSubmission	*viewed* if the object is owned.
R7	QuizAnswer	*viewed* if the object is owned.
R8	AssignmentSubmission	*edited* before the deadline.
R9	QuizSubmission	*edited* before the deadline.
R10	QuizSubmissionAnswer	*edited* before the deadline.
R11	AssignmentSubmission	*created* before the deadline.
R12	QuizSubmission	*created* before the deadline.
R13	QuizSubmissionAnswer	*created* before the deadline.

Table 6.2
Restrictions of the Student role in Mudel.

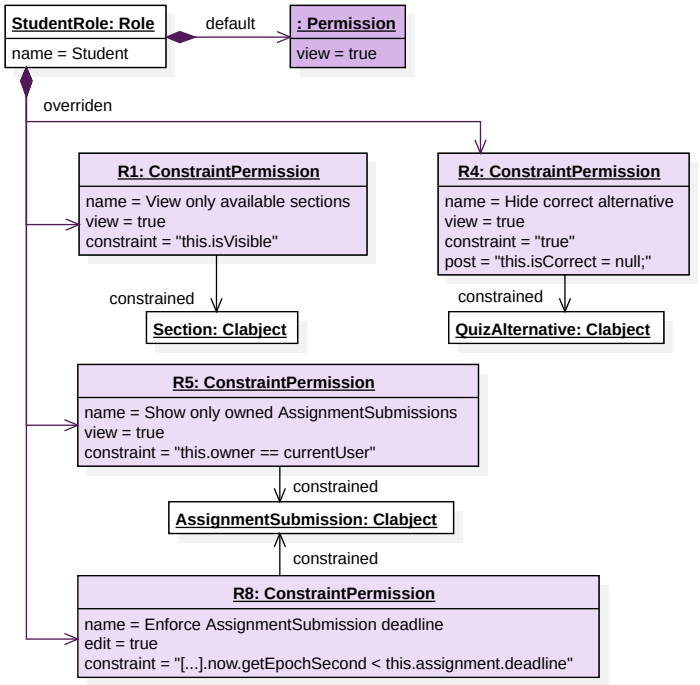


Figure 6.9
Implementing permissions for the Student role using the roles meta-model.

persisted, but computed to create a view.

Finally, Figure 6.10 illustrates the integration of roles within Dandelion+. First, the end users interact with the platform and trigger the query of some data. Upon resolution, the query yields a set of objects (Clabjects) filtered according to the user's role and permissions. In case of inconsistency, the most restrictive permission applies. Moreover, the filtered objects shape the view presented to the end user. Finally, recall that ConstraintPermission.post

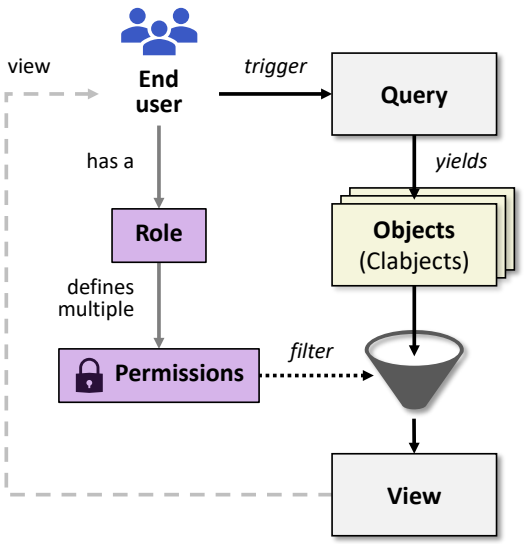


Figure 6.10
Views processing according to the permissions assigned to the current user's role.

modifications must be local (e.g., modify the value of an attribute) and are not persisted. This way, the view-update problem from databases [231] is circumvented.

6.5 Behavior of low-code platforms

The meta-models presented in the previous section enable the definition of the structure of low-code platforms, from the definition of DSLs and their instantiations to access control policies. However, although this approach is sufficient for simple use cases (such as CRUD dashboards offered by Google AppSheet [172]), rich applications typically depend on components with an associated behavior. That is, they require executing a set of operations (i.e., algorithms) to fulfill their operational requirements. For instance, reports need to query, analyze, and aggregate data for their visualization; and forms may perform validations over their input fields before storing the provided data into new entities.

This section presents the approach of Dandelion+ to model behavior, which includes workflows and applications. First, 6.5.1 introduces PlatFlow, a language for defining workflows in low-code platforms, and the execution of the workflows defined in this language is explained in 6.5.2. Next, 6.5.3 presents some exemplary workflows for the Mudel platform. Finally, the meta-model for deploying applications on Dandelion+ using workflows is presented in 6.5.4.

6.5.1 PLATFLOW

PLATFLOW is a workflow language tailored to specifying and executing behavior in low-code platforms. The following subsections present its design rationale (6.5.1), the structure of the workflows defined with it (6.5.1), and a catalog of its available nodes (6.5.1).

Design rationale

PLATFLOW is founded on domain adequacy and technological decoupling for workflow execution. On the one hand, PLATFLOW is meant to define behavior directly in the browser. To achieve this, a drag-and-drop graphical editor is provided, featuring nodes that interact with the platform where workflows are executed (e.g., managing entities). Additionally, the editor is designed to be platform-aware: many nodes include dropdowns and search boxes to quickly locate entities within the current platform, facilitating workflow development. On the other hand, PLATFLOW's execution is decoupled from the persistence layer used by Dandelion+ (or the LCDP workbench it integrates with). This is achieved thanks to a heavy reliance on Epsilon, which is decoupled from the peculiarities of the underlying technology by the Epsilon Model Connectivity Layer Section 2.1.8.

Workflow structure

A workflow is an executable entity that, given a set of inputs passed as parameters, produces a result. Workflows consist of an exposed interface and implementation logic. As with the rest of Dandelion+, workflows in PLATFLOW are developed following a model-driven approach. Figure 6.11 presents the meta-model that captures their structure.

Regarding their interface, Workflows have a name, a description, may define input parameters, and have an output format. First, InputParameters have a name and are typed. In particular, they can have a primitive type (i.e., PrimitiveParameter, typed by a PrimitiveType from Figure 6.5) or may reference platform-specific entities, such as users, files, models, or metamodels (ReferentialParameter). Second, the enumeration OutputFormat specifies how to render the output of the workflow when executed. It can be textual, including plain text, HTML, Markdown, or code snippets; or structured, including files and validation reports from EVL.

Regarding their implementation logic, workflows consist of Nodes connected by Edges. In order to correctly redirect the information between nodes, these expose input and output Ports, which can be named. Most nodes have one or two input ports and an output port. Moreover, some nodes have extensible input ports, allowing the creation or removal of input ports

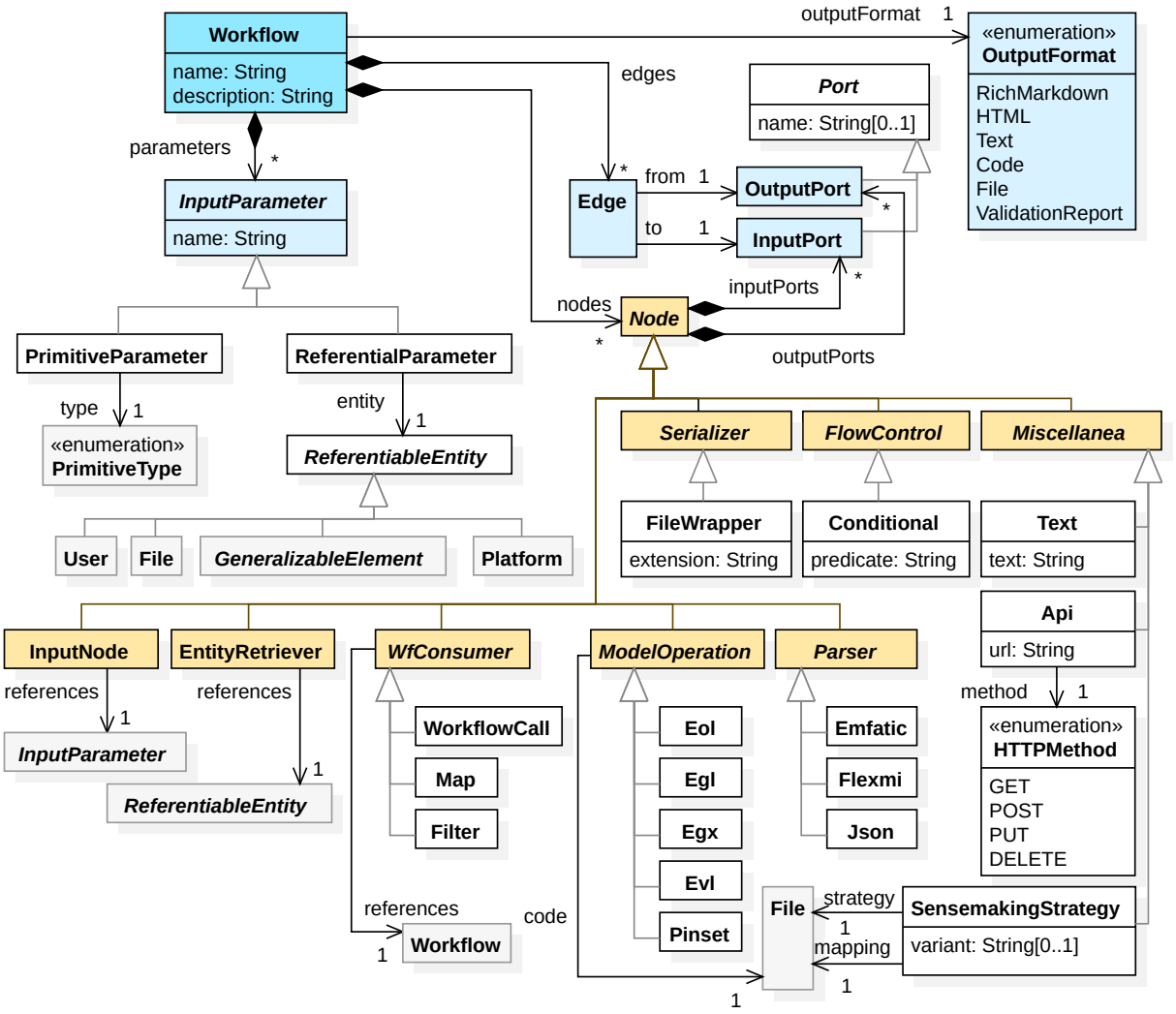


Figure 6.11 Workflows package.

at will. Appendix B attaches the constraints of the meta-model, including those regarding the number of input and output ports of each node.

From a computation standpoint, workflows are similar to regular software programming functions. However, beyond regular programming, workflow nodes permit defining high-level operations over platform entities via a graphical editor, reducing the need to write code.

Workflow nodes

PLATFLOW supports the following concepts:

Input nodes. This type of nodes is used to connect the value of an input parameter to other nodes of the workflow.

Entity retrievers. EntityRetriever nodes load entities within the current platform for their subsequent manipulation in the workflow. The entities that can be managed include users, files, models, meta-models, objects, meta-classes, and the current platform itself. There are as many nodes as entity types, and each node has a dropdown to find the desired entity (except for the current platform retriever node).

Workflow consumers. These nodes leverage other workflows for their execution, enabling workflow reuse through the references association. Three types can be distinguished:

- WorkflowCall. This node executes another workflow defined in the system. The input parameters of the workflow are passed via named input ports. Therefore, the node requires a one-to-one correspondence between the input ports of the node and the parameters of the referenced workflow. That is, they must match in quantity and by name. The node's output is the result of executing the referenced workflow with the provided inputs.
- Map. This node applies the referenced workflow to each element of an input collection, yielding a new collection of transformed elements. The referenced workflow must have exactly one input parameter, and the input collection is the flattened union of all values received through the input ports. The node outputs a collection containing the results of applying the referenced workflow to each input element.
- Filter. This node is similar to Map, but instead of transforming elements, it selectively retains only those satisfying a specified predicate. The referenced workflow must have exactly one input parameter and return a Boolean value. The input collection is constructed the same way as in Map, and the output collection contains only the elements for which the referenced workflow evaluates to true.

Overall, WorkflowCalls enable the composition of workflows, thereby enabling reuse. In addition, the Map and Filter nodes act as higher-order functions, enabling loop-like behavior in workflows.

Model operations. These nodes wrap Epsilon [232] language calls to perform model operations (Section 2.1.8). All of them accept an arbitrary number of input ports, which are promoted to context variables within the scripts, and require their code to be specified in a File. Following this approach, code snippets can be shared among different workflows. First, Eol executes Epsilon Object Language [77] code, implementing arbitrary operations over any entity. Second, Egl and Egx make use of the Epsilon Generation Language and the EGL Co-Ordination Language [72], respectively, to generate artifacts from templates: Egl populates only one template, while Egx may produce multiple artifacts. Finally, Evl validates models against EVL scripts [41], generating a summary of the violated constraints, and Pinset [233] generates tabular structures out of models.

Parsers. Parser nodes make use of existing parsers and Epsilon drivers[2] to structure textual information. First, the Emfatic parser consumes textual descriptions of meta-models in Emfatic and outputs a Dandelion-specific meta-model. Second, the Flexmi parser consumes a meta-model and a FLEXMI [234] description of a model to load it into the workflow. Finally, the Json parser produces a queryable in-memory model from a JSON structure.

Serializers. The FileWrapper node produces a file with a specified extension resulting from joining the contents of the input ports. It can be used, for instance, to aggregate the different outputs of a workflow into a compressed file (e.g., a zip file), thereby allowing users to download multiple artifacts at once. This node requires setting the workflow's OutputFormat to File.

Flow control. The Conditional node modifies the flow control of the workflow by switching between two branches depending on the value of its predicate (a Boolean expression in EOL).

Miscellanea. First, Text is a utility node that makes a text available in a workflow, without the need of creating a file. Second, the API node allows making HTTP requests to REST APIs via a URL. The node accepts an arbitrary number of input ports, whose values populate the body of the request. The node accepts methods GET, POST, PUT, and DELETE.

Sensemaking strategies. The model sensemaking strategies (SMSs) presented in Chapter 5 can be exploited in Dandelion+ via a workflow node. Given their domain and level agnosticism and the model-driven foundations of Dandelion+ (cf. P1 and P2 in Section 6.3.1), SMSs are a good fit for

[2] https://eclipse.dev/epsilon/doc/emc/#emc-drivers

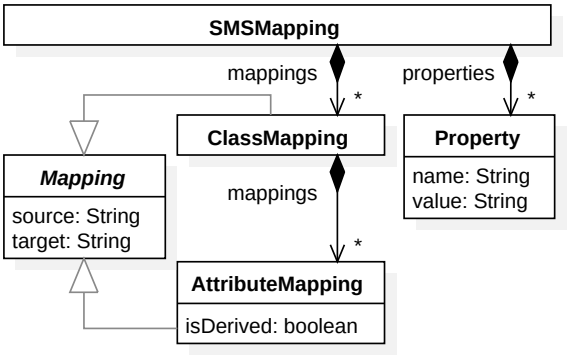


Figure 6.12
Sensemaking strategies mapping meta-model.

understanding Dandelion+ entities: from entire Platforms, to users, DSLs, meta-models, or models.

The SensemakingStrategy node allows exploiting SMSs within workflows. Its use requires three ingredients, following the approach presented in Section 5.3:

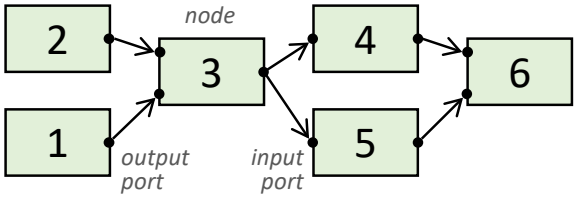
1. The target (meta-)model, which is the artifact under study for which a visualization is produced.
2. The context meta-model, which shapes the strategy by describing the structure of the visual artifact (e.g., containing meta-classes like Bar, Point, or Boxplot).
3. A mapping that establishes a correspondence between the target (meta-)model and the context meta-model to populate the SMS.

The node receives the target (meta-)model via input ports, and applies a sensemaking strategy to it. Both the context meta-model and the mapping are described in Files: the former is written in Emfatic, and the latter is defined in a dedicated DSL, whose abstract syntax is governed by the meta-model presented in Figure 6.12. In particular, mappings relate meta-classes from the context meta-model to the target meta-model via ClassMappings, each containing AttributeMappings which may be derived. Non-derived mappings map attributes directly by name, whereas derived mappings use an EOL expression for the target where 'this' refers to the current instance. For instance, an AtributteMapping for a source 'label' attribute could be mapped to a non-derived mapping to a target 'name', or to a derived mapping with the expression 'this.name.toUpperCase()'.

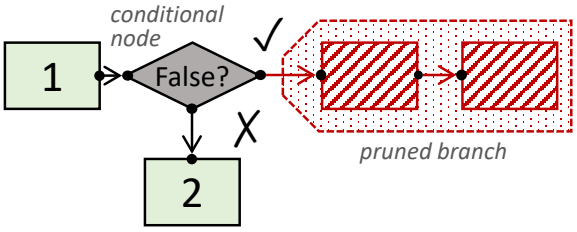
Finally, mappings may define properties (e.g., the title of the chart or the labels of the axes) as pairs name/value.

6.5.2 PLATFLOW execution

Workflows are executed according to the dependencies imposed by the edges between the ports of their nodes. In particular, for a workflow to be



(a) A possible schedule of the execution for the nodes of a workflow — if an output port of N_1 is connected to an input port of N_2, then N_1 must be scheduled before N_2.



(b) Pruning unexecuted conditional branches: as the predicate of the conditional does not hold in this execution, the nodes pending from the true branch are discarded.

Figure 6.13 Workflow execution.

well-formed, the graph induced by its nodes and edges must form a directed acyclic graph (DAG). The schedule of execution of nodes is then computed via a topological sort [235] of this DAG, as shown in Figure 6.13a. The execution engine, then, produces the schedule incrementally to prune the unmet branches every time a Conditional node is reached, as demonstrated in Figure 6.13b.

The result of the workflow is the output of the last executed node. Internally, data transfer relies on variable injection. In particular, when a node is executed, the runtime engine automatically populates its execution scope with variables from two sources simultaneously: workflow parameters and input ports. The names of the injected variables correspond to those of the input parameters or input ports, respectively.

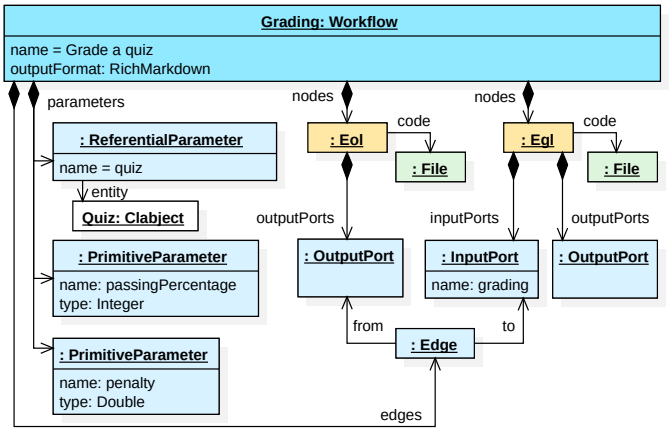
6.5.3 Exemplary workflows

This section exemplifies PlatFlow through the implementation of three workflows tailored to the Mudel platform: Workflow #1 grades the submissions of a quiz (6.5.3), Workflow #2 generates a dashboard for the personnel of the platform (6.5.3), and Workflow #3 produces a downloadable version of the dashboard generated by Workflow #2 (6.5.3).

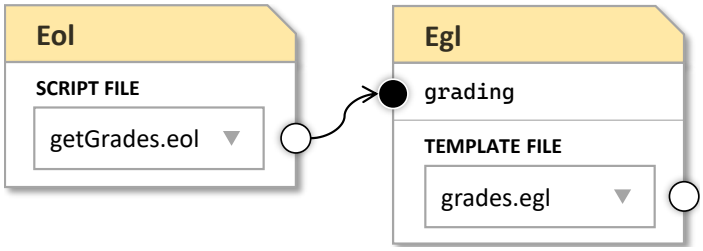
Workflow #1: quiz grading

Figure 6.14 presents Workflow #1, a workflow that reports on the grades of the submissions of a quiz. The workflow is displayed in abstract syntax in Figure 6.14a, and in the concrete syntax that the stakeholders of Dandelion+ use in Figure 6.14b.

As Figure 6.14a shows, the interface of the workflow has three input parameters: the quiz, a passingPercentage (i.e., a number from 0 to 100), and a penalty factor (e.g., 0.25) that is applied to wrong answers. The quiz parameter points to the Quiz meta-class, allowing the professor to select the quiz to grade, while the rest are (typed) numeric PrimitiveParameters.



(a) Internal representation using PLATFLOW.



(b) Equivalent graphical representation of the workflow.

Figure 6.14 Workflow #1: grading quizzes report.

Additionally, the output format of the workflow is set to RichMarkdown to render the generated report appropriately.

The workflow implementation logic involves two nodes, as shown in Figure 6.14b. The first is an Eol node that runs an EOL script that grades the QuizSubmissions of the selected quiz (cf. Listing 6.1). In particular, it tallies the number of right, wrong, and unanswered questions per submission. By accounting for the penalty factor (cf. line 13), the script also determines their grades. Note that the script can use variables such as quiz (lines 2 and 3) or penalty (line 13), since input parameters are automatically injected into the execution context of every node. Moreover, the script demonstrates how EOL's dot operator can be exploited to navigate Dandelion+ (meta-)models. Two examples are quiz.questions (line 2) and answer.chosenAlternative.isCorrect (line 20).

The last step generates a report using an Egl node. To do so, the node populates the template in Listing 6.2. This consumes a grading variable (cf. line 4) that contains the result of the previous node and is passed to the Egl node via a named input port with the same name (cf. Figure 6.14). This template has the particularity of producing code in MDX[3] (an XML-like

[3] https://mdxjs.com

Listing 6.1 Script *getGrades.eol* for grading quizzes.

```
var res = new Sequence;
var total = quiz.questions.size().asReal();
for (submission in quiz.submissions) {
  var tally = submission.tally();
  var right = tally.right;
  var wrong = tally.wrong;
  var unanswered = total - right - wrong;
  res.add(Tuple {
    "student"    = submission.owner,
    "right"      = right,
    "wrong"      = wrong,
    "unanswered" = unanswered,
    "score"      = 100.0 * ( right - wrong * penalty ) / total
 });
}
return res;

operation QuizSubmission tally() : Tuple {
  var right = self.answers.select(a
              | a.chosenAlternative.isCorrect).size();
  var wrong = self.answers.size() - right;
  return Tuple { "right" = right, "wrong" = wrong };
}
```

embedded language) to render special widgets in the output. This approach permits enriching an otherwise static result (i.e., just Markdown) with interactive widgets, including tables, buttons, or tabs.

Finally, this workflow executes first the EOL node, followed by the EGL node. Since the EGL node is the last to be executed, its output becomes the output of the entire workflow, i.e., the report in Markdown, as specified in the Workflow object in Figure 6.14a. For instance, when the workflow is executed on a quiz, with a passingPercentage of 50, and a penalty of 0.25 as input parameters, it generates a report like the one in Figure 6.15.

Workflow #2: personnel dashboard

Workflow #2 implements the dashboard from Figure 6.16, where the personnel can get an overview of the status of the platform. Figure 6.17 presents its structure.

Interface-wise, the workflow does not have any input parameter; instead, the elements it works with are directly obtained within the workflow. The output format of the workflow is set to RichMarkdown to render MDX tags, as in the previous example. As per its implementation logic, the workflow execution has three phases. First, the current Platform is obtained by an EntityRetriever node. Second, the platform object is passed to an EOL node,

Listing 6.2 Template *grades.egl* for grading quizzes.

```
# Grades of quiz [%=quiz.name%]
<Table>
  <Row> <Th>Student</Th> <Th>Score</Th> <Th>Answers</Th> <Th>Passed</Th> </Row>
  [% for (grade in grading) { %]
    <Row>
      <Cell>[%=grade.student.name%]</Cell>
      <Cell>[%=grade.score%]</Cell>
      <Cell>*OK* [%=grade.right%],
            *KO* [%=grade.wrong%],
            *NA* [%=grade.unanswered%]</Cell>
      <Cell>[%=grade.score >= passingPercentage ? "Passed" : "Failed"%]</Cell>
    </Row>
  [% } %]
</Table>
```

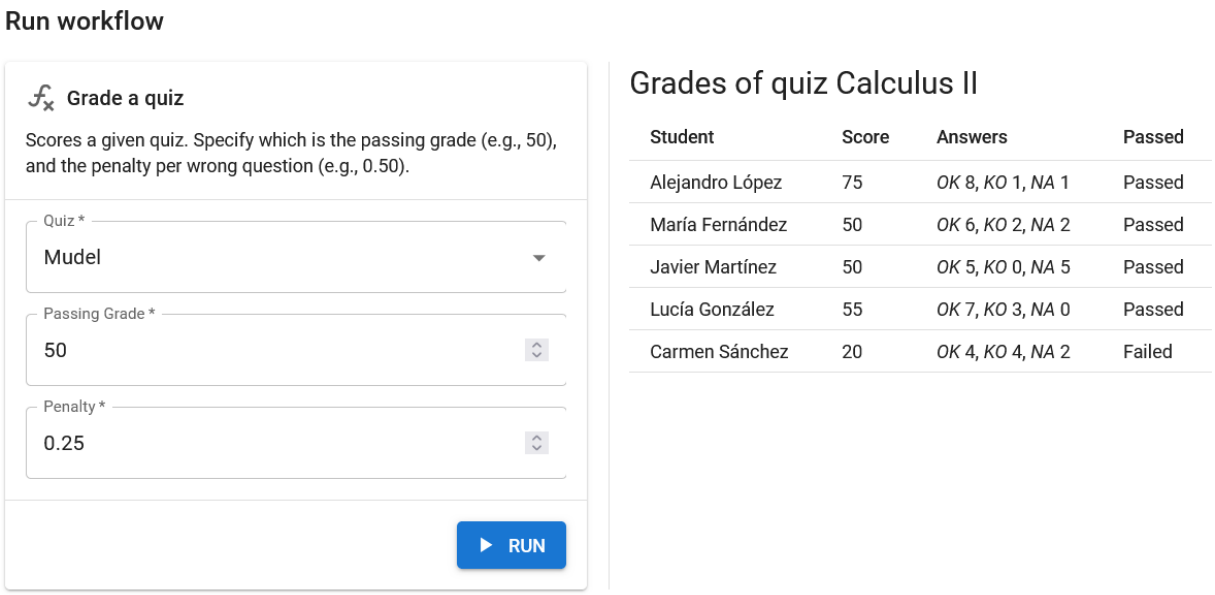


Figure 6.15 Output of Workflow #1 on a Calculus II quiz, with a passing grade of 50 and a penalty of 0.25.

which extracts information about the platform, and two SensemakingStrategy nodes, which produce visualizations about the current platform. Finally, these results are passed to an Egx node which, by means of a coordination script and templates, generates a series of HTML files that constitute the dashboard and its subpages.

To implement a sensemaking strategy, three elements are necessary: a target (meta-)model, a context meta-model, and a mapping (Chapter 5). For instance, the first SMS generates a timeline graph of the courses within the platform, with their start and end dates. Its target in this case is platform:Platform, and the context meta-model is in Listing 6.3. Context

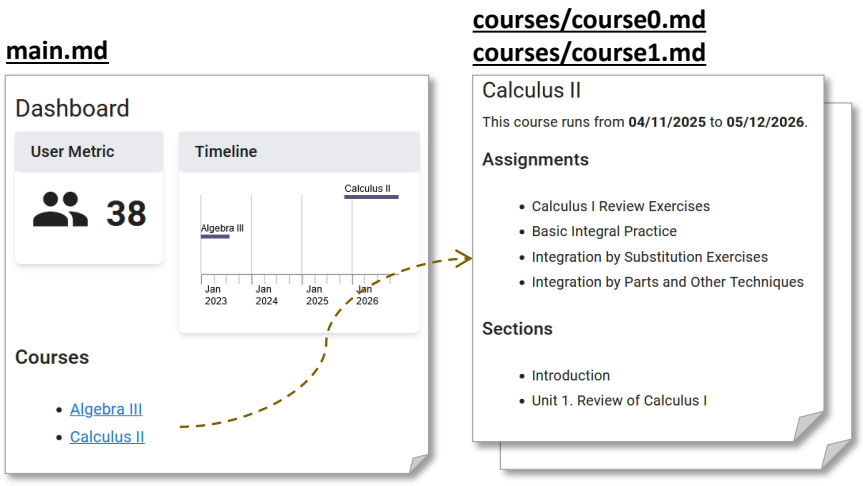


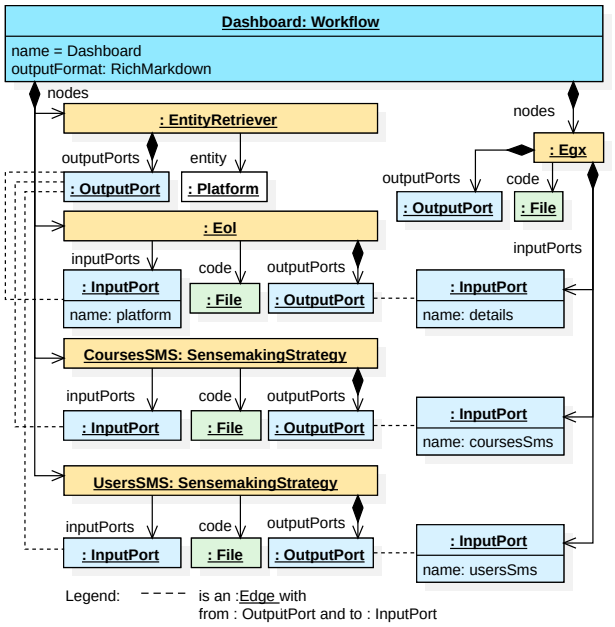
Figure 6.16 Output of Workflow #2 in multiple pages.

meta-models declare the concepts used by the sensemaking strategy, and each strategy defines its own meta-model. The timeline context meta-model only contains one meta-class, Period, that must be instantiated as many times as periods displayed in the timeline. Finally, the mapping is in Listing 6.4. In this case, it only contains one mapping rule (lines 1–6) that creates a period per course. Note that "this" on line 1 refers to the target (meta-)model, i.e., the current platform, and lines 3–5 map attributes of Course (i.e., name, startDate, endDate) to attributes of Period. Additionally, lines 8–10 define properties of the visualization, namely its title. Using all this information, the node produces an image that can be embedded in any HTML page.

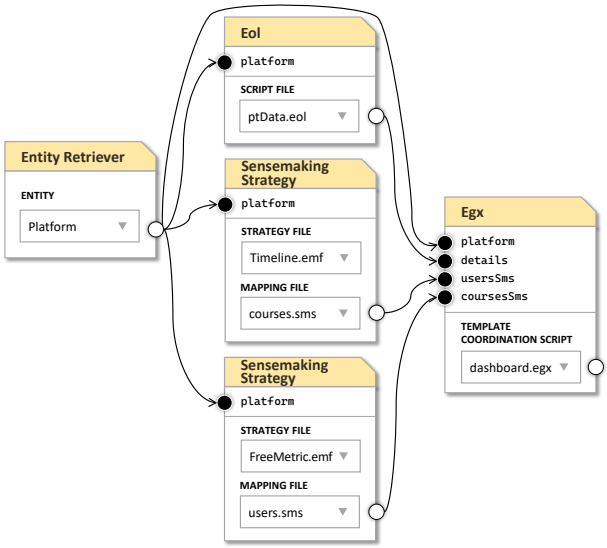
Listing 6.3 Meta-model *timeline.emf*.

```
1 package timeline;
2
3 class Period {
4   attr String label;
5   attr Date start;
6   attr Date end;
7 }
```

Regarding the Egx node, it generates several HTML files based on a series of templates (cf. Listing 6.5). In particular, it generates a main.md file, and as many Markdown reports as courses on the platform (e.g., courses/course0.md, courses/course1.md, etc.). In order to be representation-agnostic, Egx nodes always return a Java/Epsilon Map that maps the URIs of the generated files to their contents (see Listing 6.6). The workflow simply returns this map.



(a) Internal representation using PLATFLOW.



(b) Equivalent graphical representation of the workflow.

Figure 6.17
Workflow #2: generating a dashboard for Mudel personnel.

Listing 6.4 Mapping *courses.sms*.

```
map "this.models.first()!Course.all"
to "Period" {
  label <- name
  start <- startDate
  end <- endDate
}

properties {
  title: "Courses timeline"
}
```

Listing 6.5 Coordination script *dashboard.egx*.

```
pre {
  var courses = platform.models.first()!Course.all;
}

rule Main {
  parameters: Map {
     "details"    = details,
     "usersSms"   = usersSms,
     "coursesSms" = coursesSms
  }
  template: "dashboard/main.egl"
  target:   "main.md"
}

rule Subpages transform course in: courses {
  parameters: Map { "course" = course }
  template:   "dashboard/course.egl"
  target:     "courses/course" + courses.indexOf(course) + ".md"
}
```

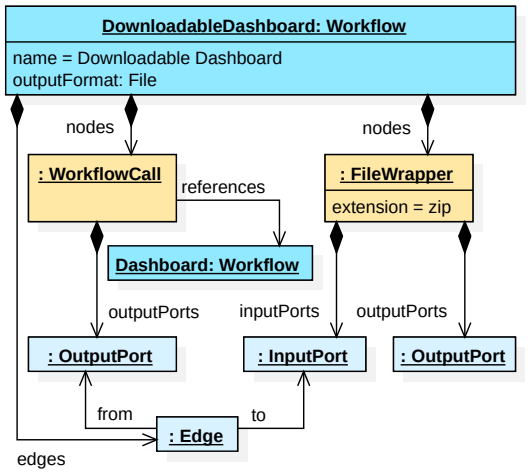
Workflow #3: downloadable personnel dashboard

Low-code platforms sometimes need to support downloading artifacts instead of just displaying them. For this reason, Workflow #3 shows how to download the files produced by a workflow. Figure 6.18 presents its structure.

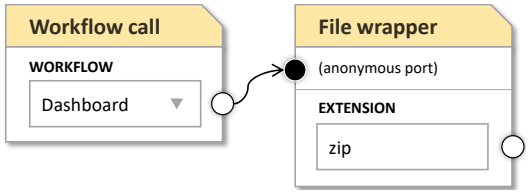
The workflow replicates the functionality of Workflow #2, with the difference that the generated report is downloaded as a compressed file containing a static web page with its contents. To achieve this, the workflow uses two nodes. First, the WorkflowCall node allows reusing logic defined in other workflows by calling them directly. In this case, the Dashboard workflow is referenced (cf. Figure 6.17), and no information is passed as input, since the called workflow does not have input parameters. Upon execution of this new DownloadableDashboard workflow, Dashboard is called and its

Listing 6.6 Example of a map returned by an Egx node.

```
{ "main.md": "# Dashboard ...",
  "courses/course0.md": "# Calculus II ...",
  "courses/course1.md": "# Algebra III ...", ...
}
```



(a) Internal representation using PLATFLOW.



(b) Equivalent graphical representation of the workflow.

Figure 6.18 Workflow #3: generating a downloadable dashboard for Mudel personnel.

result is passed to a FileWrapper node that bundles the information passed to it into a File. In most cases, the specified extension will leave the file as is (e.g., json, yaml, or xml). However, the zip extension is handled in a special way, as it expects a Map object as produced by an Egx node (cf. Listing 6.6), and generates a compressed file with the contents of the map. Finally, the output format of the workflow is set to File to display a download button correctly.

6.5.4 Application deployment meta-model

While all workflows can be executed standalone, some of them can be deployed as running parts of the Dandelion+ interface. This is achieved with applications. Figure 6.19 models applications, together with the support for file systems.

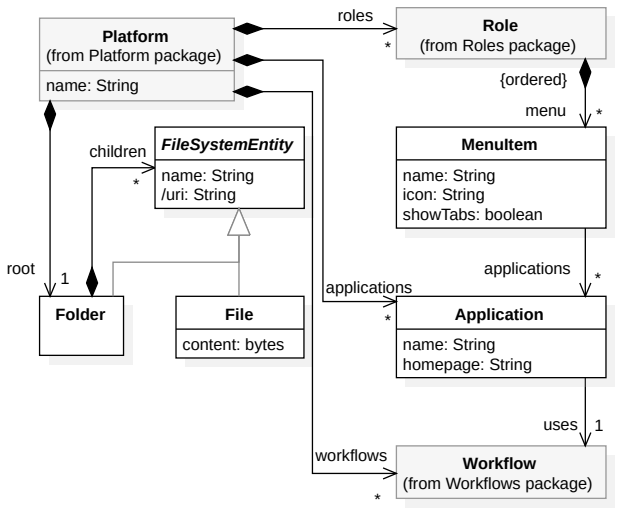


Figure 6.19
The applications package.

Regarding file systems, each platform has one, and they comprise Folders, which may be nested, and Files. This way, files with any content may be stored in an ordered fashion, from scripts to documents and images. Both Files and Folders are identifiable via a URI in code or in the workflow editor.

Regarding workflows, some can be deployed as Applications. In this context, an application is an indivisible collection of pages that fulfill a purpose. For this reason, only workflows that produce collections of pages are eligible for deployment. In PLATFLOW, the node that generates such pages is Egx:[4] it produces a map from strings to strings, which can be interpreted as a mapping URL $\mapsto$ ⟨page contents⟩, i.e., a collection of pages. For instance, whenever Workflow #2 (6.5.3) is executed, it produces a map as the one in Listing 6.6. Therefore, the only required information to deploy the workflow is to set its entry point to main.md via Application.homepage. Please note that pages of any format can be served: from HTML, to Markdown or any other format (e.g., txt).

Figure 6.20 shows how to model the application side of the PersonnelRole. The role has a menu consisting of three items: Dashboard, Projects, and Models. Each of them is tied to a single application that deploys a workflow. In particular, the former deploys Workflow #2 (6.5.3), whereas the latter provide pages for the projects (meta-models) and models of the platform. These last workflows are created by default.

[4]To be more precise, only workflows that return a map with strings as keys and values can be deployed as applications.

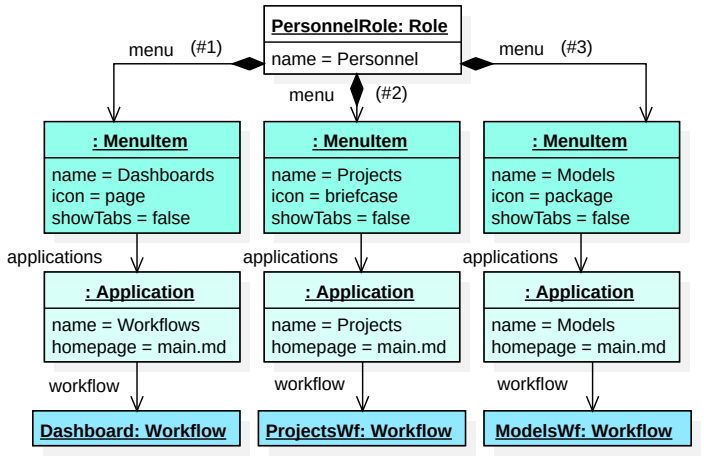


Figure 6.20
Example of workflow deployment in applications.

6.6 Architecture and tool support

This section presents the architecture (6.6.1) and tool support (6.6.2) of Dandelion+.

6.6.1 Architecture

Figure 6.21 presents the architecture of Dandelion+, which is heavily inspired by that of Dandelion (cf. Section 4.3). The main difference is that the frontend now entails multiple pages (i.e., the tool is no longer a single-page application), and that the backend provides multiple external services to support the new features of Dandelion+.

More specifically, the architecture of Dandelion+ is divided into a frontend and a backend component.

The frontend exposes a web-based application available for all stakeholders (i.e., LCDP engineers, citizen developers, and end users). Technologically, the frontend makes use of React [211] to build the user interface together with MUI,[5] a React component library. The graph-based editors are implemented using React Flow,[6] and the Files page makes use of the Ace Editor.[7] Finally, the frontend communicates with the backend via a REST API.

As for the backend, it is the heart of the tool, as it manages platforms and all their elements (e.g., (meta-)models, workflows, or roles). It is implemented in Java, and it integrates with a number of services. Namely: a PlatFlow service, to define and execute workflows; a layout service by the Eclipse Layout Kernel (ELK) [214] to arrange elements automatically in the diagrams; a Sensemaking Strategies service to process the sensemaking

[5] https://mui.com
[6] https://reactflow.dev
[7] https://ace.c9.io

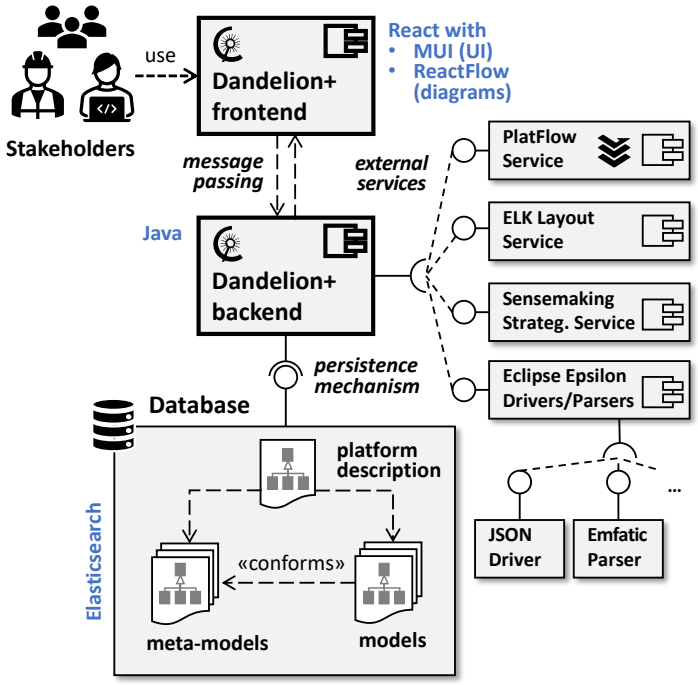


Figure 6.21
Architecture of Dandelion+

strategies defined in the workflows (cf. node SensemakingStrategy in 6.5.1); and Epsilon tooling, including drivers and parsers, to interact with different technologies, such as JSON, Emfatic, or FLEXMI.

Finally, persistence is based on Elasticsearch [123]. This technology was chosen in Dandelion for its scalability and speed (cf. Section 4.3). In this backend, all the elements related to platforms are persisted and handled.

6.6.2 Tool support

This section showcases the functionality of Dandelion+, which is offered as a website. To do so, it illustrates how to develop the Mudel platform (Section 6.2) from scratch, following the process presented in Figure 6.3. In particular, this section explains how Dandelion+ manages platforms (6.6.2), how to develop the structure (6.6.2) and behavior (6.6.2) of a platform.

Managing platforms The entry point of the tool is the top-level homepage, which lists all available platforms and permits logging into them (cf. Figure 6.22).

From there, the administration page (Figure 6.23) is also available, allowing administrators to manage the platforms defined in the system. The page provides metrics about them, such as the number of users, meta-models, or models. Its access is protected by a password set during the installation of Dandelion+.

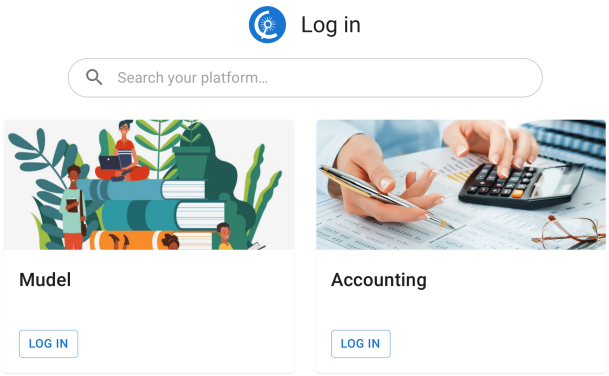


Figure 6.22
Top-level homepage.

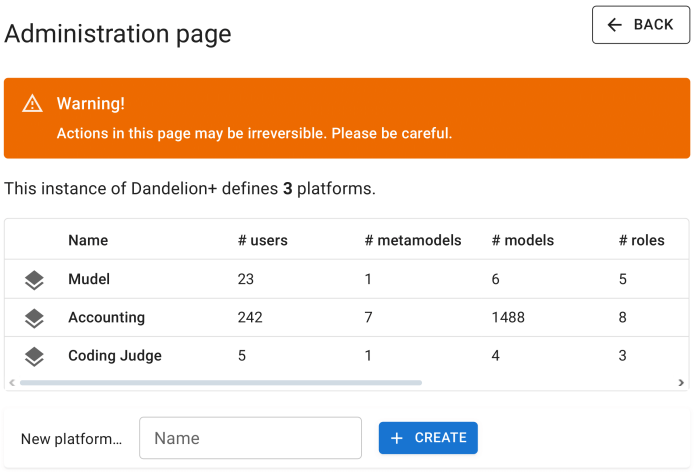


Figure 6.23
Administration page.

Once a platform is created from this page, the system automatically populates multiple entities for its correct functioning. Namely: an administrator Role; a User with that role, allowing the LCDP engineer to log in and start developing the platform; workflows for predefined pages (e.g., Projects, Models, Workflows...); and applications for these workflows.

Developing the structure of a platform To develop the structure of a platform, users need to log into it. The first page users encounter is the platform *Homepage* (cf. Figure 6.24). All pages have a header and a navigation menu, which links to the applications defined in the platform. The ones included by default are: the *homepage*, *projects*, *models* (i.e., meta-models and models), *workflows*, *files*, *applications*, a *chatbot*, and *platform settings*.

The Projects page allows managing meta-models. Dandelion+ features two flavors for meta-model editing: tabular and graphical (cf. Figure 6.25). Both allow manipulating concrete and abstract classes, enumerations, properties, literals, associations, and containments.

On the one hand, the tabular editor (cf. Figure 6.25a) permits hiding or

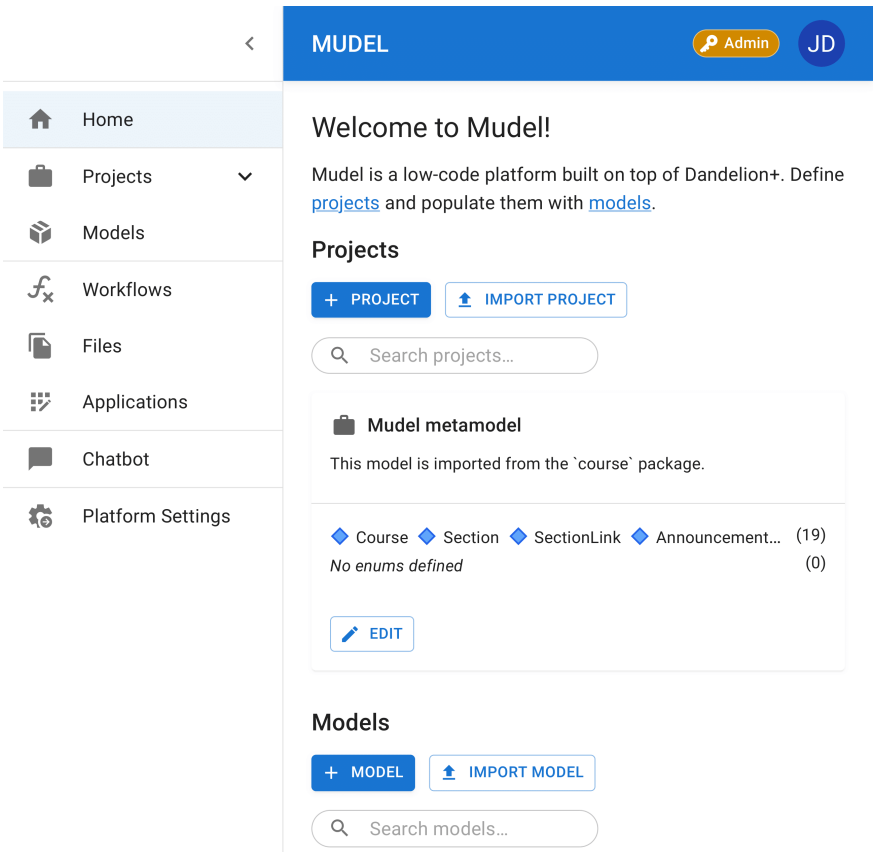
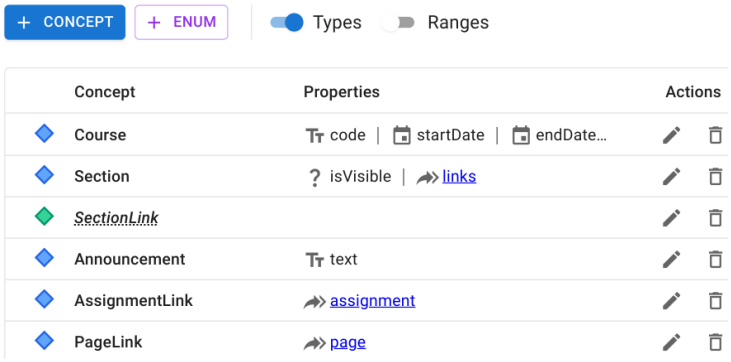


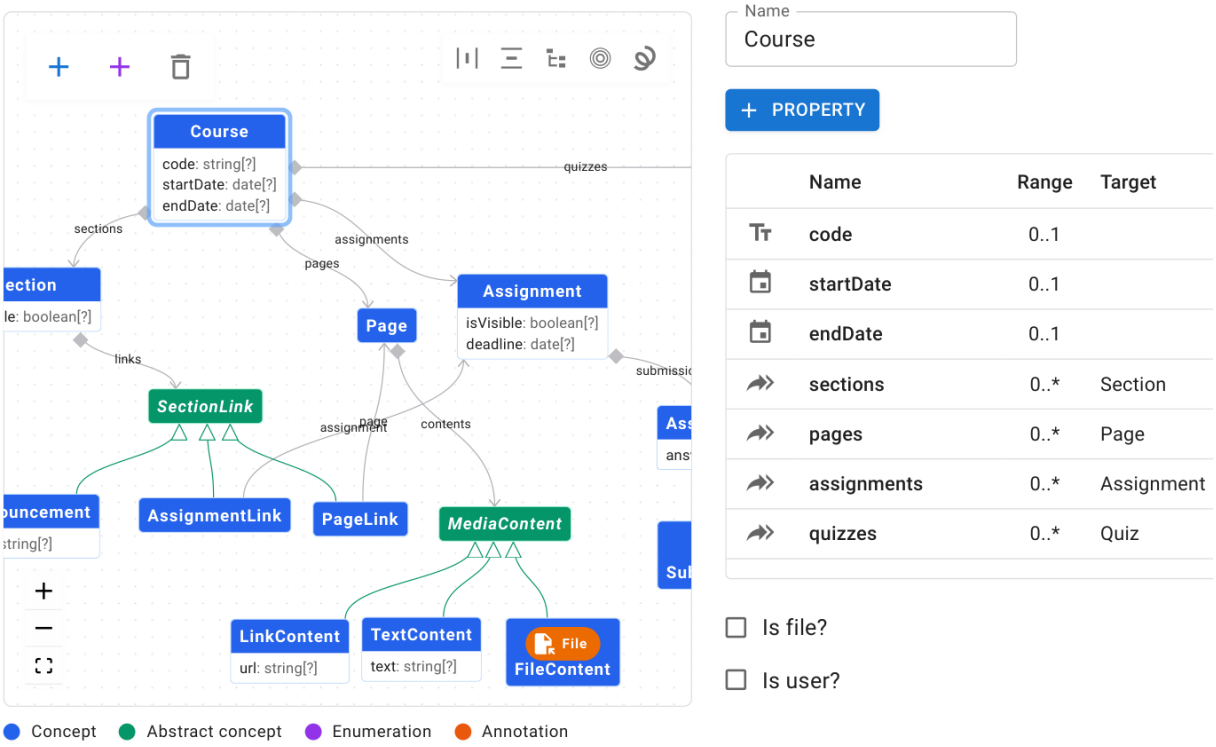
Figure 6.24
Homepage of the Mudel platform.

showing both types and ranges (i.e., multiplicities), to better adapt the level of detail of exploration of a meta-model. Note that, although the tabular representation is appropriate when meta-models are very dense, it poses a problem regarding the graph-like nature of meta-models. In particular, acknowledging the relations between classes may be difficult. To alleviate this issue, relations are marked as clickable links which point to their referred types. For instance, when clicking links on the class Section, the page scrolls down to the class SectionLink and highlights that row momentarily. On the other hand, the graphical editor (cf. Figure 6.25b) is interactive, allowing users to move elements within the canvas, as well as pan and zoom. It also features automatic layout facilities out of the box, including horizontal, vertical, tree-like, circular, and force-based arrangements. Classes can be marked as abstract and decorated with User and File annotations. These are reflected in the diagram (e.g., FileContent in the figure).

Once a meta-model has been created, the user can click on the Models link to instantiate it. Figure 6.26 presents the model editor, where the objects are arranged tabularly and organized by types, displaying all their properties. Types can be hidden or made visible via checkboxes. The panels containing



(a) Tabular form of the meta-model editor.



(b) Graphical form of the meta-model editor.

Figure 6.25 Meta-model editor.

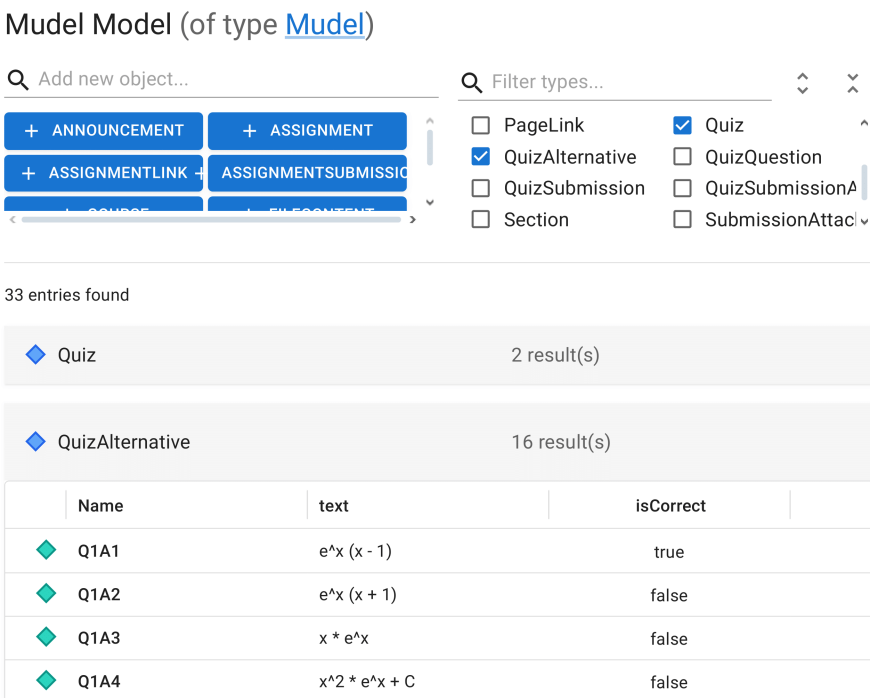


Figure 6.26
Model editor.

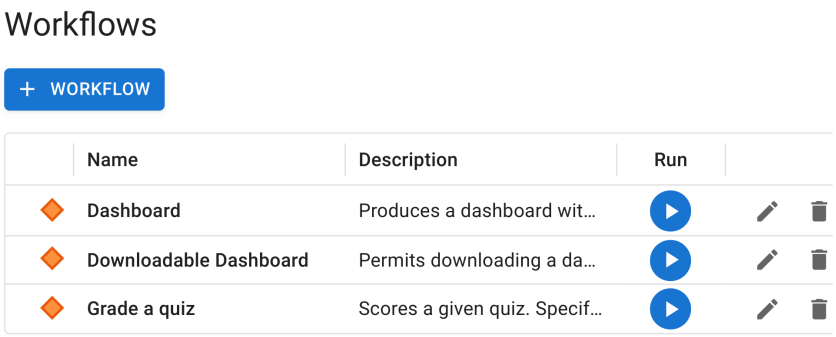
the objects of each type can also be expanded or collapsed to examine the model. Moreover, existing objects can be modified, deleted, and new ones can be created.

Developing the behavior of a platform The *Workflows* page permits managing the workflows defined in the platform (cf. Figure 6.27a). In particular, it permits modifying them, deleting them, and creating new ones.

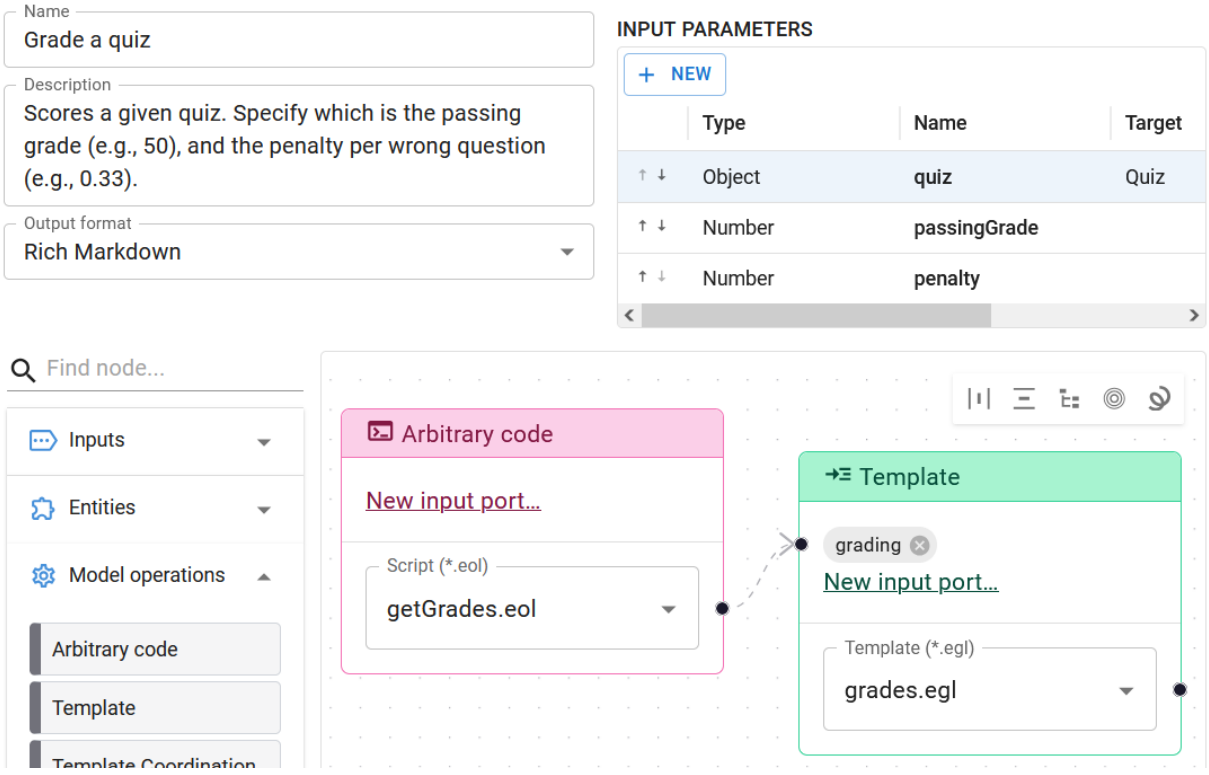
Figure 6.27b showcases the workflow editor. It allows giving workflows a name, a description, input parameters, output type, and implementation. Input parameters can be typed with either primitives (e.g., string, numbers, or Booleans) or Dandelion+ types, as presented in the linguistic meta-model in Section 6.4 (e.g., model, meta-model, role...). The visual editor allows drag and drop of nodes on the left, which are distributed in the categories presented in 6.5.1. Some nodes include search boxes that filter through the entities of the corresponding type by name for easy selection.

Workflows can be tested iteratively while running them from the *Workflows* page (e.g., Figure 6.15). As shown, the workflow is rendered as a form where input parameters are mapped to inputs, checkboxes or selects according to their typing. The visual appearance of the result of an execution depends on the output type.

The *Files* page allows manipulating the files used in the workflows as shown in Figure 6.28. In particular, the page lists all files and folders, and



(a) Workflows page.

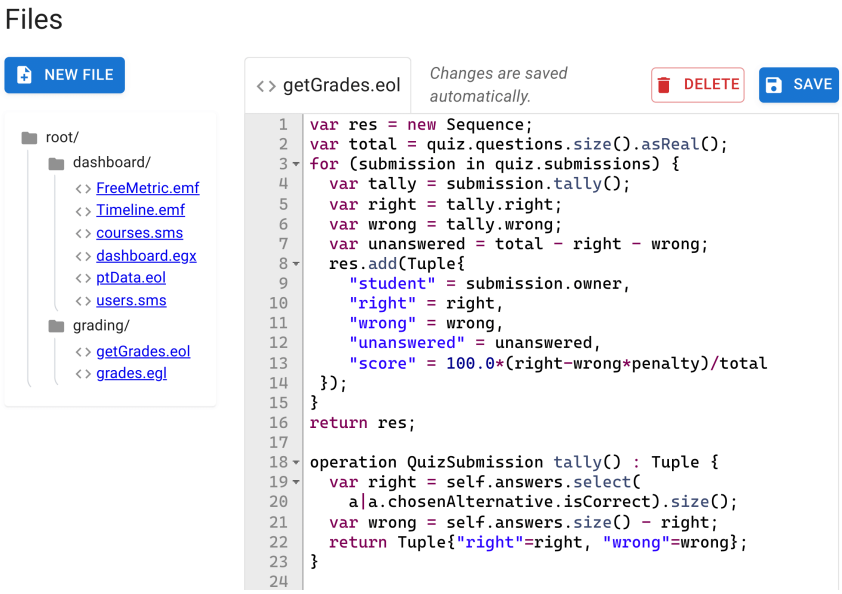


(b) Workflow editor.

Figure 6.27 Workflow management.

the right-hand-side editor allows editing the files as well as saving them.

Figure 6.28
Files page



In order to lower the entry barrier for citizen developers, Dandelion+ integrates with an LLM-based assistant powered by LowcoBot, which will be presented in Chapter 7. This chatbot is directly embedded into the tool and is aware of the platform's structure and capabilities. It assists users by answering natural language queries regarding navigation and platform usage (e.g., *"Where can I define a new role?"*), acting as an interactive guide (see Figure 7.6). More details of the integration are provided in Section 7.6.

Finally, the *Platform Settings* page permits managing roles and their menus, as well as users.

6.7 Summary and conclusion

This chapter has presented Dandelion+, a model- and workflow-driven low-code engineering platform for defining DSL-based low-code platforms and the applications they host. Dandelion+ is built on a model-driven foundation and relies on a low-code linguistic meta-model —organized into platform, domain modeling, and roles packages— to capture the main concepts of DSL-based platforms. The behavior of platforms and applications is specified through PlatFlow, a platform-aware workflow language tailored to low-code development. PlatFlow provides nodes that manipulate Dandelion+ entities, orchestrate MDSE tasks (such as model-to-model transformations and code generation), integrate external services, and invoke model sensemaking strategies to visualize information.

As will be demonstrated in the evaluations presented in Section 8.3, Dandelion+ has been assessed both against representative commercial and academic low-code platforms and within UGROUND's development process. The results show that Dandelion+ provides key features for constructing complex domain models, such as inheritance, files, and object ownership, and offers a more expressive workflow-based behavior specification toolkit, compared to competing LCDPs that lack the necessary expressiveness for this use case. Moreover, Dandelion+ can improve UGROUND's industrial process thanks to the rapid prototyping capabilities and model-driven nature of workflows.

The next chapter introduces LowcoBot, which continues to explore the intersection of MDSE and low-code but focuses on providing conversational assistance for low-code platforms through model-driven chatbots.

Chapter 7

Conversational Agents for Low-code Platforms

This chapter presents LowcoBot, a model-driven code generator that produces chatbots catered to low-code platforms. In particular, it explains the approach followed (7.2), presents the meta-model of the tool (7.3), the artifacts it generates and consumes (7.4), its architecture (7.5), and a use case applying it to Dandelion+ (7.6). This work has been published in a workshop [39].

7.1 Introduction

In the previous chapters, low-code platforms have been explored as a means to empower citizen developers to create applications that require writing less code. In particular, grounding these platforms on MDSE foundations has been shown to bring benefits: from enabling scalable graphical language workbenches (Chapter 4) and defining level- and domain-agnostic visualizations of platform elements (Chapter 5), to enabling customizable application generation processes (Chapter 6). However, while MDSE facilitates the development of low-code platforms and their usage, these platforms are still very complex in nature. That is, low-code platforms exhibit high essential complexity [236] due to the wide range of features they provide, which can be daunting for newcomers and cumbersome for experienced developers working with large projects.

Meanwhile, the field of natural language processing (NLP) has undergone significant advances in the last decade, enabling alternative ways to interact with systems through text or voice. As a consequence, chatbots (or *conversational agents*) [237] have proliferated and been adopted in a plethora of domains, including IoT, customer service, and e-commerce [238]. More recently, the discovery of the transformer architecture (2017) [239] and the subsequent breakthroughs in deep learning have revolutionized the

field of chatbots. In particular, large language models (LLMs) have enabled the development of chatbots able to understand a wide range of user intents and to answer them appropriately [240]. Finally, the ubiquity of smart voice assistants like Amazon Alexa or Apple Siri and the popularity of LLM-driven chatbots like ChatGPT [6] have brought chatbots closer to the general public and made them part of everyday life.

Therefore, a twofold potential can be observed for the application of chatbots in low-code platforms. First, chatbots can provide alternative interactions that may enhance the navigability of low-code platforms. Second, as chatbots have become ubiquitous, their incorporation does not constitute an entry barrier for citizen developers. Thus, chatbots can help users get accustomed to platforms, navigate them, and use their features. However, interfacing with low-code platforms is not straightforward, as they involve many interconnected components, including web interfaces (i.e., the frontend), APIs (i.e., the backend), and linguistic concepts (e.g., user, role, or project).

For these reasons, this chapter presents LowcoBot, a tool that generates chatbots catered to low-code platforms. LowcoBot follows a model-driven approach whereby the platform components are modeled in a structured and technology-agnostic manner, permitting the generation of chatbots for web-based low-code platforms that expose a REST API, independently of their domain. Although the current implementation relies on a stack involving Python, Streamlit, and LangChain, the model-driven design makes it possible to generate code for other programming languages or frameworks by implementing appropriate code templates. This chapter also reports on an application of LowcoBot by building a chatbot for Dandelion+ (Chapter 6) as a step towards the validation of the tool.

LowcoBot is open source and its code is freely available on GitHub.[1]

7.2 Approach

This section presents the components required to create a chatbot for a low-code platform and the approach followed by LowcoBot to generate them.

Figure 7.1 presents an overview of the approach. The chatbot is presented as a chat interface for the end user, which is backed by an LLM agent with an initial prompt that interacts with a set of defined tools. These tools enable communication with the low-code platform through its REST API, whose preexistence is assumed in this approach. While the chat interface, the LLM, and the REST API can be easily implemented or consumed, the contents of the LLM agent vary significantly between platforms, while maintaining

[1] https://github.com/LowcoBot/lowcobot

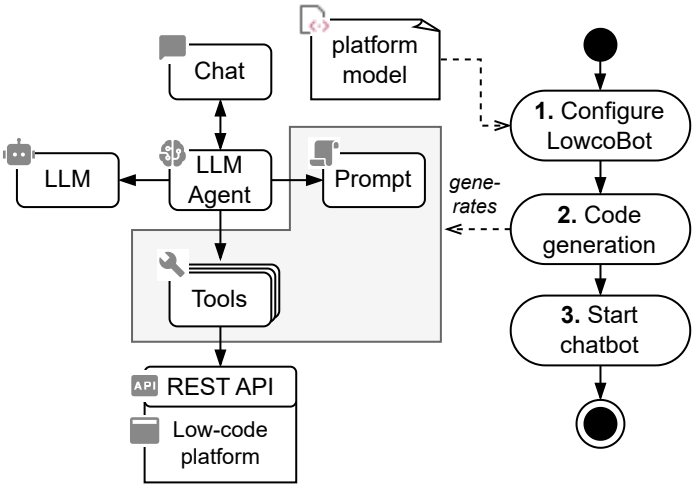


Figure 7.1
Overview of the approach.

the general architecture. Therefore, tools and prompts can be automatically derived from models of the platform, making them suitable for automation. LowcoBot exploits this fact to provide a code generation solution, taking a model-driven engineering approach, to automatically populate tools and the initial prompt from a low-code platform model specification.

More precisely, LowcoBot follows three steps. First, the tool consumes a specification of the targeted low-code platform. Next, the tool generates the necessary artifacts to communicate with the platform's REST API. Finally, the chatbot is generated, so it can be executed and interacted with by the end users. These steps are described in more detail below:

Step 1. Configure LowcoBot. First, the user of LowcoBot models the low-code platform for which to generate a chatbot using the tool's meta-model (presented in the next section), and supplies the created model to the tool. The more completely the model covers the features exposed by the platform, the more comprehensive the generated chatbot will be.

Step 2. Code generation. Once the platform model is provided, all the necessary artifacts required by the chatbot can be generated. In particular, to produce appropriate textual responses to user requests, the chatbot makes use of a large language model (LLM). Moreover, as cloud-native environments, low-code platforms expose REST APIs to operate with them via HTTP requests. These two components, however, must be mediated by an LLM agent following the ReAct pattern [241], enabling it to reason about the user's request and act by invoking the appropriate API tools. As the "heart" of the chatbot, the agent must know the internals of the low-code platform, and allow the LLM to make use of the REST API's endpoints to resolve the requests.

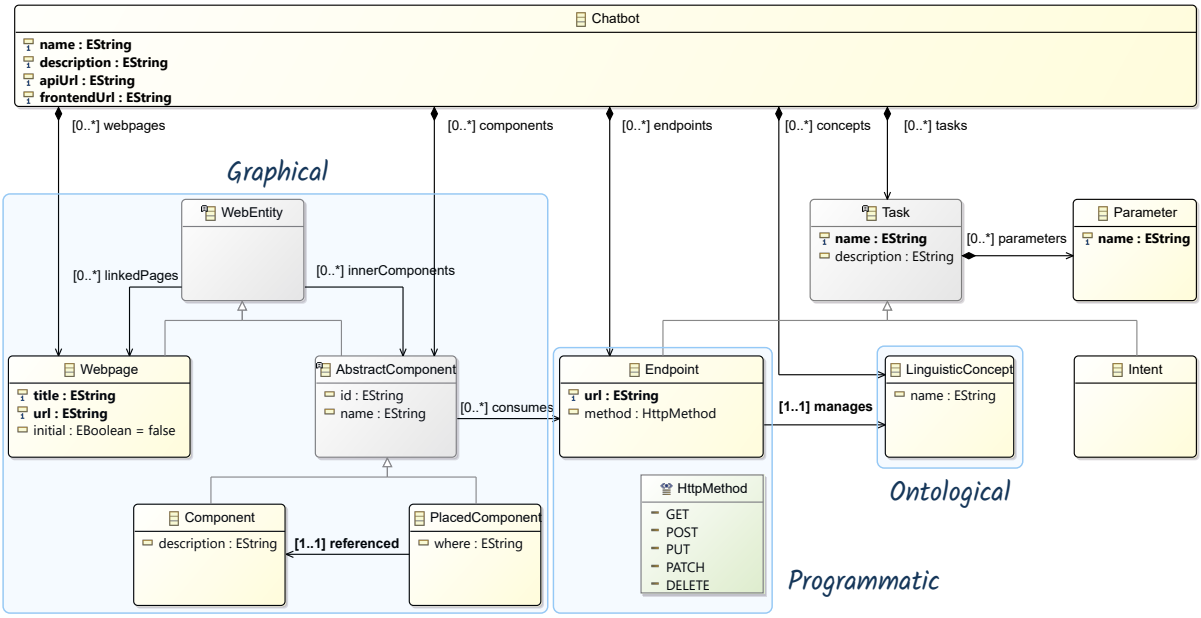


Figure 7.2 LowcoBot meta-model, relating the graphical, programmatic, and ontological spaces of a low-code platform.

In order to answer platform-related questions, the context of the LLM is augmented with the initial prompt and custom tools. On the one hand, the initial prompt is a textual artifact that is fed initially to the LLM to instruct it how to behave. In particular, it should be crafted with the appropriate context so that the LLM can provide satisfying completions for the expected intents of the chatbot. On the other hand, tools allow the LLM to trigger the execution of arbitrary code specified on the chatbot side. Typical LLM tools include code interpreters, arithmetic calculators, or weather forecast providers. In the context of low-code platforms, tools serve as the foundational blocks bridging the LLM and the low-code platform's API.

Step 3. Start the chatbot. Finally, the generated prompt and tools are wired with scaffolding code, resulting in a functional chatbot. Users, then, can interact with it via a chat interface, consisting of a textual input and the conversation history. This chatbot can be used independently, or embedded into an existing low-code platform, blending into the platform's interface.

7.3 LowcoBot meta-model

The meta-model of LowcoBot captures the relevant elements in a low-code platform to produce a conversational chatbot for it. In particular, it focuses on

three key but interwoven technological aspects found in low-code platforms: graphical, programmatic, and ontological.

Graphical – the web interface Low-code platforms are web applications that are accessed through a browser, and they typically span across multiple webpages. As these platforms follow a low-code approach, they heavily rely on visual metaphors such as tables, diagrams, or forms that are reused in different widgets across webpages. This modular approach based on reusable and encapsulated components aligns with the current fashion of web application frameworks, such as React [211] or Angular [242]. Therefore, modeling the graphical space of a low-code platform involves capturing the webpages of a platform, the links between them, and the components that are contained within them.

Programmatic – the REST API User operations in the web interface must be communicated back to the server side to make them effective. Low-code platforms typically expose their set of functionalities through REST APIs, which can be either consumed by the web interface (thus, being transparent for non-tech users) or by third-party applications. Since APIs are the main entry point to the platform's functionality, the chatbot only exposes capabilities that are explicitly modeled as API endpoints. As a consequence, covering the programmatic side of low-code platforms makes it possible to know the operations that can be performed, and allows the chatbot's agent to use them to execute the user's requests.

Ontological – linguistic concepts Each platform has a different set of domain concepts that it handles (e.g., users, roles, or projects). In LowcoBot, these are captured in a lightweight way via instances of the `LinguisticConcept` meta-class, which essentially records their names. This yields a simple vocabulary for the platform rather than a full-fledged ontology. Even so, making this vocabulary explicit allows the generated chatbots to use the platform's own terminology and to condition tools on these concept names, which can help the LLM better interpret users' requests.

The resulting meta-model is presented in Figure 7.2. Note how the meta-model is not exhaustive: it does not fully model a platform's web interface, its API, or its domain vocabulary. Instead, it focuses on capturing the relationships between entities across the three technological spaces that are useful for generating a chatbot.

Chatbots for low-code platforms have a name and a description, and are determined by the platform's frontend and API URLs. Chatbots encompass webpages, components, endpoints, concepts, and custom tasks. First, webpages have a title and are accessible at a certain URL (relative to `Chatbot.frontendUrl`), and can contain components. Components describe widgets that appear on webpages. They can either be defined directly as `Components` (with a name and a description) or by referencing other components and "placing" them using a textual explanation with `PlacedComponent.where` (e.g., for a navigation bar, it can be clarified that it is " on the left side of the page"). Components can be either nested within other components or embedded in webpages, following the spirit of web components. In turn, components can consume API endpoints, which are exposed at a certain URL (relative to `Chatbot.apiUrl`), and have an HTTP method. Endpoints, as providers of the interface of the tool's backend, manage vocabulary concepts (e.g., 'user', 'role', or 'project'). Moreover, it may be desirable to expose endpoints as tools for solving user tasks. Custom `Intents` can also be specified. Finally, both endpoints and custom intents can specify parameters as inputs for their execution.

7.4 Generated artifacts

LowcoBot's meta-model can be instantiated in a model that contains the information needed to automatically generate the initial prompt and the tools of a chatbot for a particular low-code platform.

Initial prompt On the one hand, the initial prompt is a textual artifact that instructs the LLM to behave as a chatbot for the low-code platform in question. In particular, the chatbot's name and description specified in the model are injected into it, so the LLM has a preliminary idea of the platform with which it is interfacing, even if the platform is initially unknown to the LLM. The initial prompt also contains boilerplate instructions on how to structure its answers, together with a list of the available tools and their parameters. Listing 7.1 presents an excerpt of it.

Listing 7.1 Initial prompt of LowcoBot

```
1 You are a chatbot for a low-code platform called "[%=chatbot.name%]".
2 A description about you:
3 ------------
4 [%=chatbot.description%]
5 ------------
6 Respond to the human as helpfully and accurately as possible.
7 You have access to the following tools: {tools}
8 [...]
```

Tools On the other hand, tools allow the LLM to execute arbitrary code specified on the chatbot side. Thanks to being exposed in the initial prompt, the LLM can invoke them when deemed appropriate. LowcoBot leverages this to furnish the LLM with the following tools that help navigate, explore, or make use of the low-code platform:

- **Summary tool.** It allows the chatbot to answer questions like "*What is this?*" or "*What is ⟨tool's name⟩?*". It makes use of the platform's name, description, pages, and tasks to provide an overview of its parts. Additionally, the LLM is hinted to employ this tool when the user greets the chatbot, so they receive guidance from the beginning.
- **Navigation tool.** Exploring new platforms can be daunting for newcomers. For this reason, this tool answers questions like "*Where can I ⟨do something⟩ on the platform?*" To do so, it generates instructions on how to reach a certain part of the platform's webpage. Internally, the tool constructs a navigation graph as a flattened version of the model. Nodes represent webpages, components, endpoints, and vocabulary concepts (meta-classes), while edges capture the relationships between them (e.g., linkedPages, innerComponents, or AbstractComponent.consumes). The Navigation tool currently performs a breadth-first search (BFS) over this graph to find a short path from the homepage to the targeted entity, which is then rendered as step-by-step instructions for the user. Figure 7.3 illustrates an example of such a graph.
- **Capabilities tool.** Platforms manage different concepts, each with a different set of capabilities. For example, posts may be created on social media, but not deleted. This tool aims at guiding the user when asking "*Can I ⟨verb⟩ ⟨concept⟩?*", or "*What can I do with ⟨concept⟩?*". If this is possible, the tool will offer the possibility to perform the action and will inform about its location on the platform. Otherwise, it will state that this is not possible, while detailing alternative available associated intents or endpoints. To do this, the tool checks the defined endpoints, the concepts they manage, and their HTTP method (e.g., `GET` for read, `POST` for create, `PUT`/`PATCH` for update, `DELETE` for delete), assuming a RESTful convention, and the components (and webpages) where these endpoints are consumed to locate the corresponding actions in the UI.
- **Intents.** If the tool has a certain capability, the user expects to be able to use it. In particular, the `Endpoints` and API-agnostic `Intents` are promoted to tools, usable by the chatbot. Endpoints perform the appropriate HTTP requests when invoked, while agnostic intents delegate their implementation to the chatbot designer. Both can define

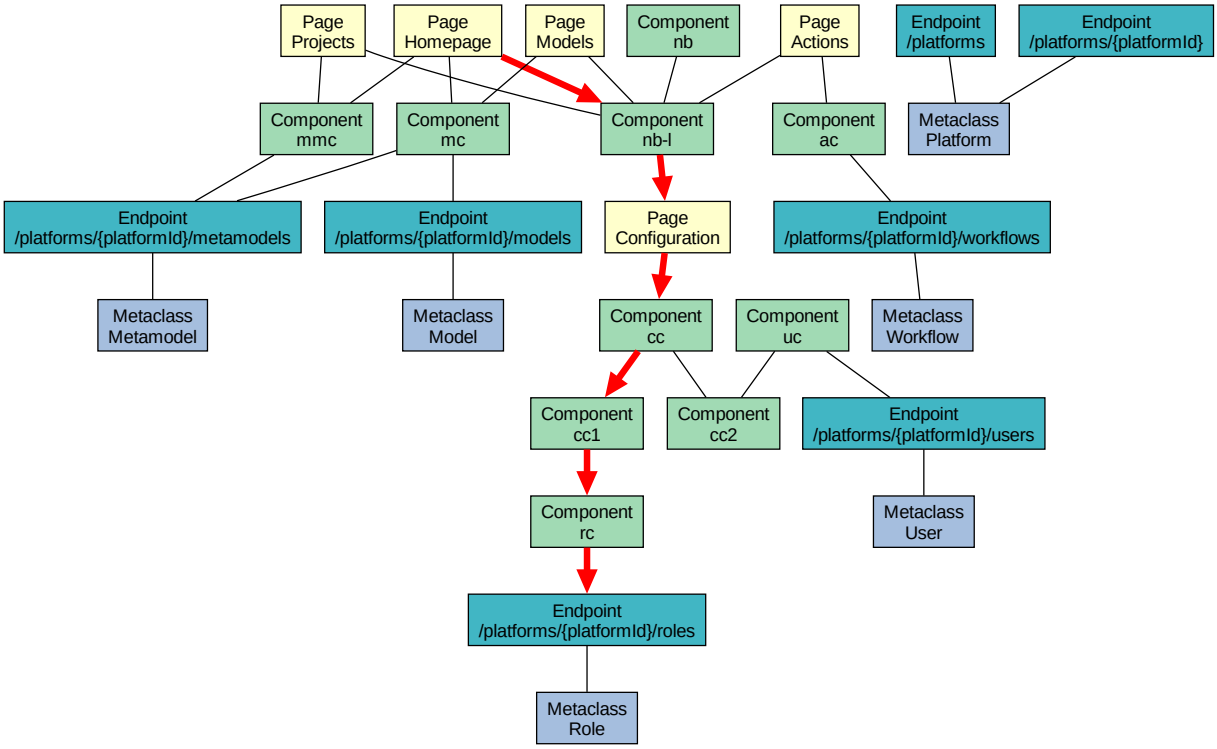


Figure 7.3 A LowcoBot model flattened to a graph, as consumed by the *Navigation* tool. Highlighted is the shortest path between the homepage and a `Role` endpoint. The path is computed to answer the question "*Where can I change roles?*".

parameters (cf. Task in Figure 7.2), which result in inputs integrated into the chat to be filled in by the user when the task is triggered (see the last interaction in Figure 7.6).

7.5 Architecture

LowcoBot's architecture comprises two technological spaces: the Eclipse environment and the Streamlit application (see Figure 7.4).

The chatbot configuration and code generation steps take place in Eclipse. In this environment, the user first designs the chatbot in a model conformant to LowcoBot's meta-model. The model is expressed in Flexmi [234] (an XML/YAML-flavored syntax for model specification), whereas the meta-model uses the Eclipse Modeling Framework (EMF) [26]. Next, this model is consumed in the code generation step, whose source code works atop Eclipse Epsilon [41]. In particular, it comprises several Epsilon Generation Language (EGL) templates and an EGL Coordination Language (EGX) script that orchestrates them. When the transformation is executed, it generates

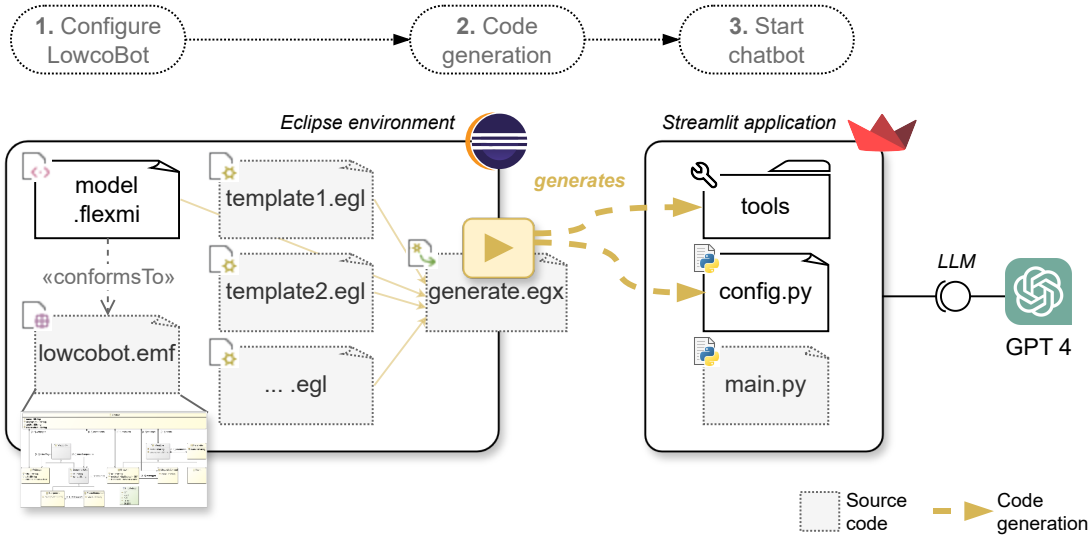


Figure 7.4 LowcoBot's architecture.

the chatbot's tools and the initial prompt. Note that, in order to extend LowcoBot with more tools, it suffices to create more EGL templates and link them properly in the EGX script.

The generated artifacts are incorporated into a Streamlit application. Streamlit[2] is a general-purpose Python framework for building web applications. In this implementation, a chat interface is provided, backed by a structured chat agent[3] defined in LangChain [243] —a framework for LLM development. LangChain is agnostic to the employed LLM, allowing it to be switched easily. The current implementation uses GPT-4, which is available via a pay-per-use API, and relies on LangChain to interface with it, as this framework provides built-in support for tool-augmented agents while remaining agnostic to the underlying LLM. LangChain also consumes the tools and the initial prompt generated in the previous step directly, as the templates generate LangChain-specific code. Additionally, the Navigation tool makes use of the NetworkX[4] Python library for graph traversal operations. Finally, Streamlit supports an embedded mode, allowing easy integration into any website via `iframe` HTML elements.

7.6 Use case

As a case study, LowcoBot is employed to generate a chatbot for Dandelion+ (Chapter 6), demonstrating the direct integration between the platform

[2] `https://streamlit.io`
[3] `https://python.langchain.com/docs/modules/agents/agent_types/structured_chat`
[4] `https://networkx.org`

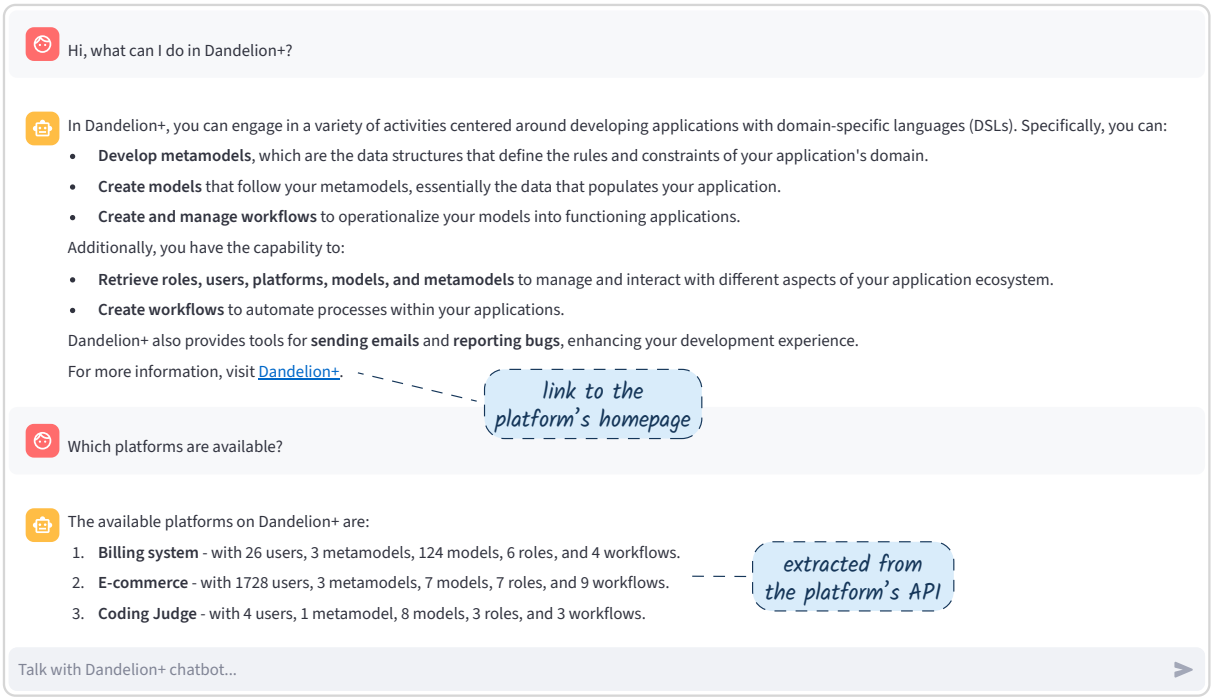


Figure 7.5 A chatbot generated by LowcoBot for Dandelion+.

definition and the conversational agent.

Dandelion+ is modeled using 36 entities, including 5 webpages, 10 components, 8 endpoints, 6 concepts, and 2 API-agnostic intents (i.e., sending emails and reporting bugs). This model, expressed in XMI/Flexmi, spans 51 non-empty lines of code (LOC). LowcoBot automatically generates 14 tools[5] and the initial prompt, which add up to 552 Python LOC (without empty lines or comments), yielding a code generation ratio of 10× (generated artifacts vs. specification LOC). The features of the generated chatbot are showcased in two screenshots:

On the one hand, Figure 7.5 shows an interaction where the user greets the chatbot and it responds with an overview of the chatbot (Summary tool). Then, the user asks for information about a low-code concept, which is answered using an endpoint intent. In particular, the tool associated with this last interaction makes a request to the Dandelion+ REST API to retrieve the available platforms in the system and construct an answer. On the other hand, the rest of the tools are demonstrated in Figure 7.6, where the chatbot is fully integrated into Dandelion+. First, the user asks for directions on where to perform a specific action within the web interface, and the chatbot

[5]Namely: Summary, Navigation, and Capabilities; 8 for endpoints; 2 for agnostic intents; and 1 for intent coordination.

responds with indications and links (Navigation tool). Then, the user makes a follow-up question regarding the capabilities of a concept, for which the chatbot informs the user and suggests their execution (Capabilities tool). Finally, the user triggers an API-agnostic intent (i.e., reporting a bug), which results in a form with the modeled parameters for the user to fill in and submit.

7.7 Summary and conclusion

This chapter has presented LowcoBot, a code generator for chatbots aimed at low-code platforms. LowcoBot is built following a model-driven approach, allowing it to capture the graphical, programmatic, and ontological sides of low-code platforms, in the form of an explicit vocabulary of platform concepts. From this model, the tool automatically generates chatbot tools that support users in getting accustomed to, navigating, and using the platform. In particular, the LowcoBot meta-model relates webpages and components, REST endpoints, and vocabulary concepts so that summary, navigation, capability, and task-specific tools can be generated from a single platform specification. The chapter has also presented an architecture that combines an Epsilon-based modeling and code-generation environment with a Streamlit/LangChain runtime that executes the generated LLM agent. The applicability of LowcoBot has been demonstrated by generating a chatbot for Dandelion+, generating a set of useful introductory and navigation tools with a generated-artifacts-to-specification ratio of 10× in lines of code.

The next chapter presents the evaluations carried out to assess the previous contributions of this thesis.

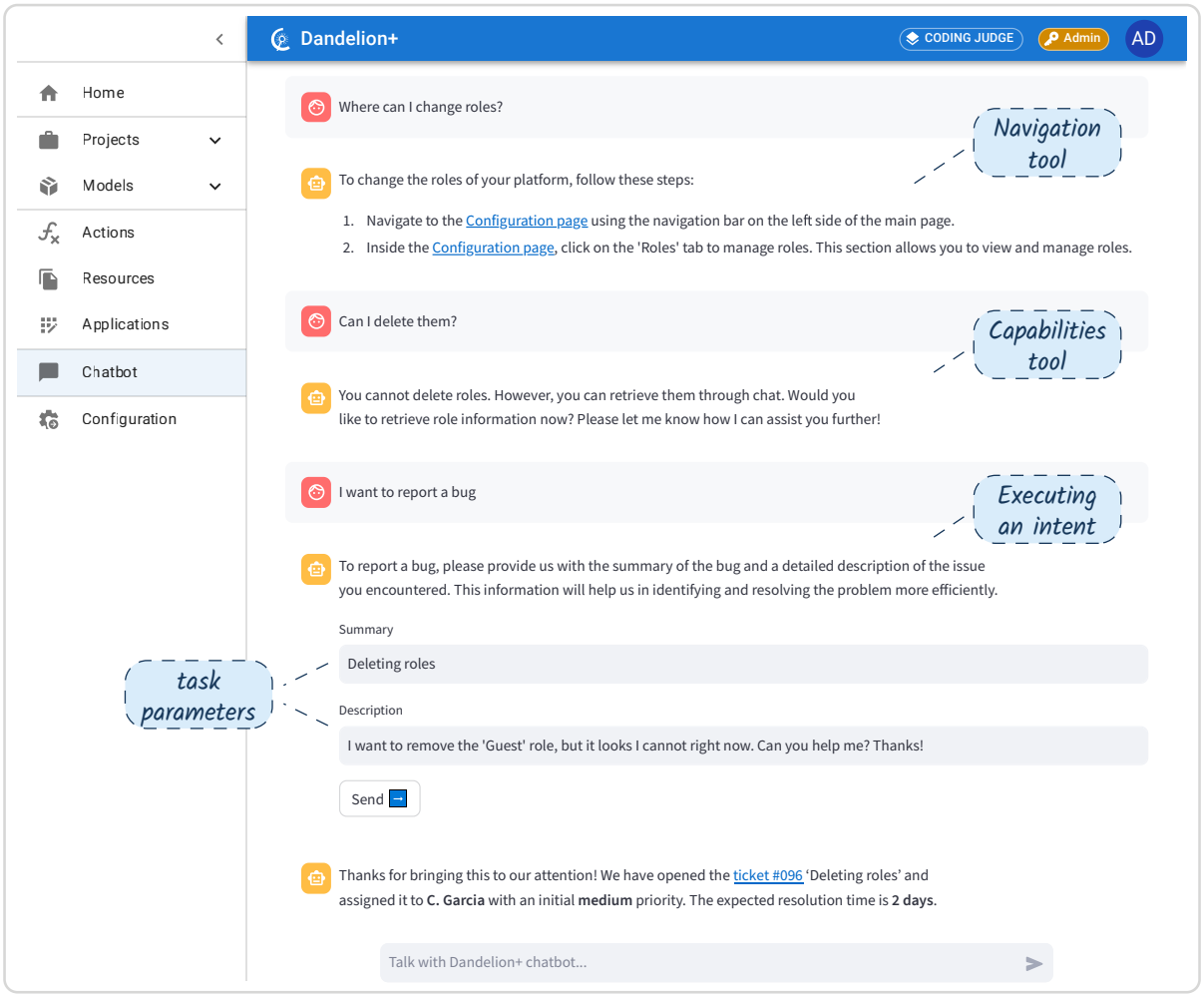


Figure 7.6 A LowcoBot chatbot integrated into the Dandelion+ platform.

Part III

Evaluations

Chapter 8

Evaluations

This chapter presents the evaluations of the main tools introduced as contributions of this thesis in Part II: Dandelion, model sensemaking strategies (SMSs), and Dandelion+. Most of the evaluations are conducted in an industrial setting within UGROUND [36], a software company that develops enterprise low-code solutions, and they rely on realistic data from the company. In particular, this chapter reports on studies that assess the scalability of Dandelion (8.1), the versatility of sensemaking strategies to understand modeling ecosystems (8.2), and the development of low-code platforms using Dandelion+ (8.3). Data from the evaluations are available at the tool's website.[1]

8.1 Scalability evaluation in Dandelion

This section reports on the evaluation of Dandelion (Chapter 4) in the context of developing a web IDE for UGROUND, driven by the company's needs.

A first set of experiments assesses the reactivity of Dandelion's model exploration mechanisms from a citizen developer viewpoint. It assesses the scalability to load and display large synthetic models —of up to one million elements— in the domain of process mining [244]. The evaluation considers Dandelion from two perspectives, leading to the following two research questions (RQs):

RQ1 *Can large synthetic models be loaded using Dandelion's pagination scalability mechanism?*

RQ2 *What is the practical limit of Dandelion's page capacities?*

Moreover, since Dandelion's research is driven by actual industrial needs, it also aims to evaluate whether Dandelion can be used as a frontend for

[1] `https://miso.es/tools/Dandelion.html`

UGROUND's LCDP, called ROSE [40]. Hence, in a subsequent experiment, the following research question is posed:

> **RQ3** *Can an existing industrial low-code technology be injected into Dandelion to visualize large industrial models?*

Experiment setting of RQ1 and RQ2 To address RQ1 and RQ2, synthetic models of event logs (instances of the meta-model in Figure 4.11) are generated to evaluate the scalability of the pagination mechanism. The following algorithm is employed to generate synthetic instances of event logs given their size s:

1. Create a log L and r Resources, where r is uniformly distributed between $0.008 \cdot s$ and $0.012 \cdot s$.
2. Repeat until s elements are created:

   With 10 % probability, create an Event and link it to L. Otherwise, create a Trace T with e Events, with e uniformly distributed between 2 and 150, and link them to T; for each event, assign its causedBy to a random Event in e and, with 0.8 % probability, assign its originator to a random Resource in r.
3. Return L.

The probability ranges and distribution numbers were determined experimentally. The aim was to generate realistic logs densely populated with traces connected to events, plus a few scattered standalone events and resources. This shape of models (with both densely connected and sparsely connected objects) is especially interesting to evaluate the limits of this pagination mechanism. In particular, the algorithm yields one log with approximately 81 % of traces with 2–150 events each, 10 % of resources, and 9 % of single events. Note that, although the resulting number of elements (and, therefore, nodes) is s, the effective size of the generated models is higher because of the associations between elements.

Experiments have been conducted on a Windows 11 Pro 64-bit laptop with an Intel® Core™ i7-7700HQ CPU @ 2.80 GHz and 16 GB of RAM, and the employed browser was Mozilla Firefox 109. To increase the reliability of the results, each measurement is repeated 10 times, and the median value is taken.

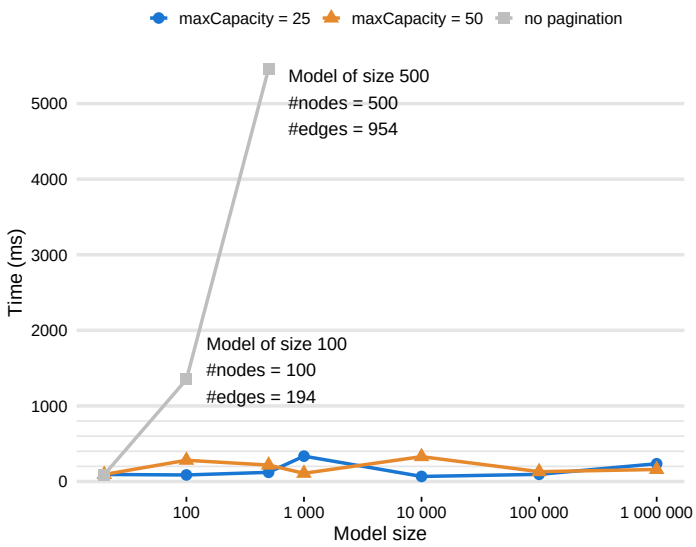


Figure 8.1
RQ1: Varying model size and page capacities.

8.1.1 RQ1: Loading large synthetic models

This experiment aims to assess whether Dandelion's pagination mechanism (cf. Section 4.2.4) is effective in loading large synthetic models. To this end, models of different sizes are created and the time required to visualize them is measured. A visualization without pagination is used as a baseline and is compared to two visualizations that rely on pagination. In particular, the experiment is carried out on models of size 20, 100, 500, 1 000, 10 000, 100 000, and 1 000 000; and paginations of a maximum capacity of 25 and 50. To make the measures more realistic, a Force-based layout (defined in Figure 4.4) is applied to the models. Figure 8.1 shows the results.

The graph shows that, for small page capacities of 25 and 50, loading times remain consistently below 400 ms, irrespective of the model size. Conversely, the time grows rapidly with the model size if no pagination is used, making this approach unsuitable for large models. Therefore, pagination is effective for loading large models. Arguably, this is because the mechanism relies on Elasticsearch, a scalable database that can quickly retrieve model subsets. Additionally, pagination exploits how the parent/elements relation between Models and their contained elements (cf. Figure 4.2) is persisted in the database. Please note that, for model sizes of 1 000 and 1 000 000, loading a page of size 25 resulted in slightly higher loading times than loading pages of size 50. This can be attributed to the low query volume, as well as to the real total content of the page (edges and proxy nodes), a factor that will be analyzed in the next research question.

Thus, RQ1 can be answered positively: large synthetic models of up to one million elements can be loaded reactively when using pagination with small page capacities.

8.1.2 RQ2: Limits of Dandelion page capacities

This experiment aims to evaluate the limits of pagination in terms of query resolution and layout computation and rendering. To this end, a large model of fixed size 100 000 is used to observe how varying page capacities affect the overall visualization time. Figure 8.2 shows the time it takes to query the database and render the retrieved elements using the Force-based layout for increasing page capacities.

The graph shows several tendencies. First, the layout rendering time increases with the page capacity. This is expected, as more nodes and edges have to be arranged. Second, the database querying time also increases, but much more slowly —note that the X axis (page capacity) is logarithmic. Layout rendering times show a logarithmic tendency, whereas database querying times show a linear tendency. A small decrease can be observed in the querying times for page capacities of 2 048 (about 200 ms lower). This inconsistency may be due to the database response time being very sensitive at low query volumes, thus being susceptible to subtle variations, such as caching. Third, the total time is dominated by the layout rendering time, thus becoming the bottleneck of the entire mechanism. Finally, the mechanism can be considered reactive (i.e., it offers a fast user experience) for page capacities up to around 512, as the total time is within a few seconds (≤ 3 s). For sizes of 1 024, 2 048, and 4 096 (omitted in the graph), the total time increases to around 7.6, 9.1, and 32.8 seconds, respectively. In particular, the browser often becomes unresponsive for page capacities around 4 096 or higher.

To better understand the real limitations of page capacities, it is necessary to consider the internal structure of models, which may group nodes and edges differently. Figure 8.3a shows the median of nodes and edges found on the pages while varying the page capacity. The X and Y axes represent the number of nodes and edges, respectively, and both employ a logarithmic scale. Each segment represents the content of a page. The left ends, the circles, represent the maximum capacity, as studied in Figure 8.2. The right ends, the triangles, account for the total number of nodes on screen. That is, the maximum capacity plus the number of proxy nodes. The length of the segment represents, therefore, the number of proxy nodes.

Figure 8.3b accompanies the graph with the sum of number of nodes and the number of edges, the number of proxy nodes, and the ratio between the total number of visual elements and the page capacity.

Some observations can be made. The most important observation is that the number of visual primitives on screen is higher than the configured page capacity, because proxy nodes and edges are not counted towards this capacity. For example, for page capacities of 512, the number of primitives is around 1 600; around 3 times higher. Additionally, the number of proxy

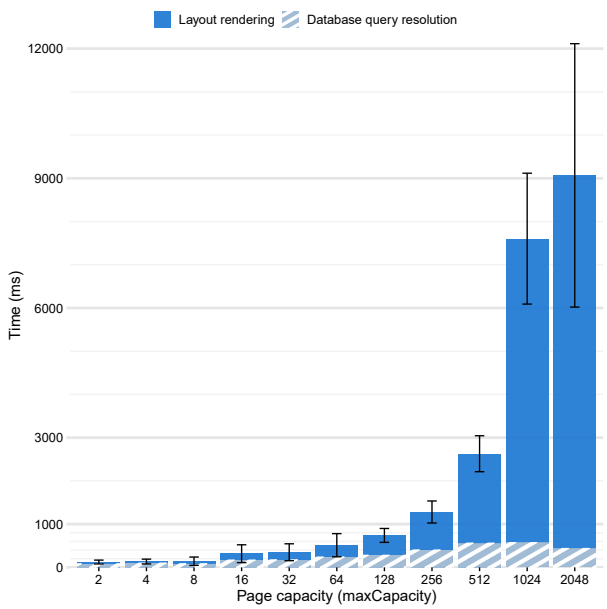
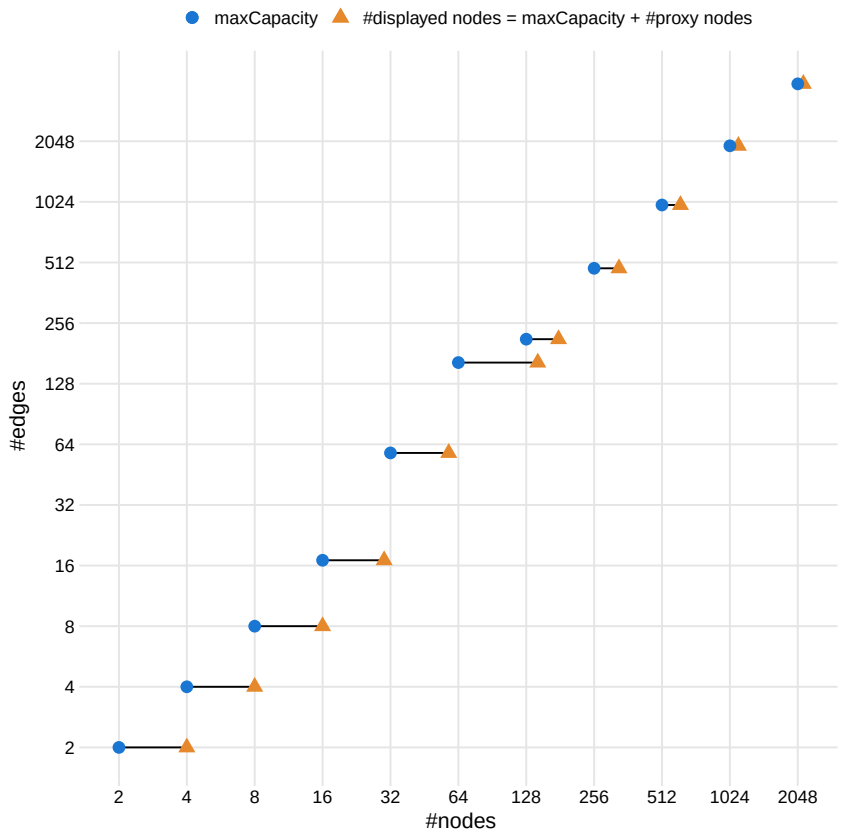


Figure 8.2
RQ2: Varying page capacity on a fixed-size model.

nodes grows with the page capacity. Larger pages increase the likelihood that traces (which are dense in edges) are split across several pages, and each split introduces extra proxy nodes.

To answer RQ2, the limits of page capacities are determined by four factors: the layout rendering time, the database querying time, the number of nodes, and the number of edges. These depend on the structure of the visualized model (i.e., its meta-model) and on the content of the specific pages being queried. For the event log meta-model considered in this experiment, a page capacity of 512 strikes a good balance between all the factors. This allows rendering pages within a few seconds, providing a reactive user experience. The total number of visual primitives is around 1 600. To visualize other meta-models, it is recommended to adjust the page capacity parameter starting from a few hundred until finding a good balance. Finally, pagination becomes very slow at page sizes of thousands, which corresponds to 3 000 or more visual primitives. These become the effective limits of the mechanism's reactivity and, therefore, should not be surpassed.



(a) Relation between page capacity and the number of nodes and edges.

maxCapacity	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1 024**	**2 048**
# displayed nodes	4	8	16	30	58	144	178	330	618	1 117	2 168
# edges	2	4	8	17	58	163	213	479	988	1 944	3 947
# nodes + # edges	6	12	24	47	116	307	391	809	1 606	3 061	6 115
# proxy nodes	2	4	8	14	26	80	50	74	106	93	120
(# n+# e) / maxCapacity	3.0×	3.0×	3.0×	2.9×	3.6×	4.8×	3.1×	3.2×	3.1×	3.0×	3.0×

(b) Table accompanying the graph with the sum of number of nodes and the number of edges, the number of proxy nodes, and the ratio between the total number of visual elements and the page capacity.

Figure 8.3 RQ2: Relation between page capacity and the number of nodes and edges.

8.1.3 RQ3: Loading industrial models

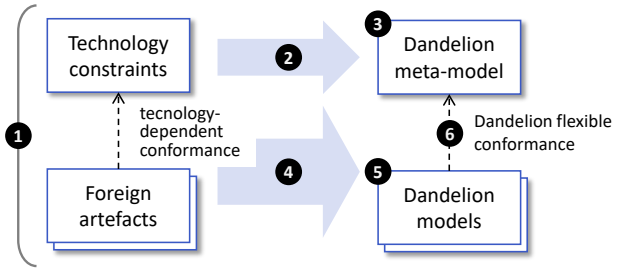
This research question evaluates Dandelion in a realistic setting, based on UGROUND's working environment. Hence, it reports on the creation of a frontend for UGROUND's LCDP using Dandelion, and on its use to manage realistic industrial models. Integrability with third-party technologies is central for language workbenches that target low-code. This is especially the case in industrial settings, where data can come from different sources, be represented in various formats, and persisted differently. This section explains the process of injecting foreign technologies into Dandelion and how the tool can help refine the process. To illustrate it, the integration with ROSE [40], an industrial model-based platform by the UGROUND company, is shown.

Figure 8.4a outlines the conceptual setting and Figure 8.4b depicts the process followed to inject a generic foreign technology into Dandelion. The aim is to port foreign artifacts and the constraints they are subject to so that they fit within the tool. Conceptually, two technical spaces need to be bridged [245]: UGROUND's ROSE and Dandelion. Since ROSE has no explicit meta-model, it has to be discovered, hence the process entails the creation of two injectors: the *schema injector* and the *data injector*. The former is responsible for extracting the meta-model of the data, while the latter is responsible for extracting the data itself. The process is iterative and can be repeated multiple times to refine the results.

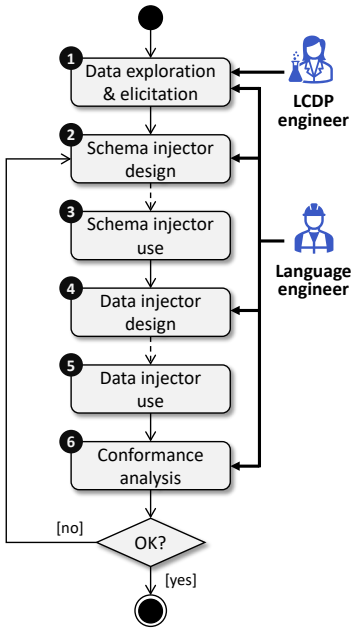
In the following, each step in Dandelion is explained —accompanied by common scenarios and challenges— and the process is illustrated with the injection of ROSE into Dandelion.

1. Data exploration and elicitation The goal of the first step is to understand the data to devise the best way to inject it into Dandelion. It involves two roles: the LCDP engineer and the language engineer. The former is typically fulfilled by a domain expert who is also knowledgeable about the technology used to persist the data (e.g., low-code artifacts), while the latter has expertise in modeling and is familiar with the infrastructure of Dandelion. Both have to make a synergistic effort to establish a common understanding of the data content and its structure. A practical methodology towards this goal is domain-driven design, which can help establish a ubiquitous language to distill the data domain [25].

The following questions can steer the discussion in the right direction: "*Which type of data is to be injected into the tool?*", "*What is it used for?*", "*Which database engine is used?*", "*How many elements are there?*", "*In which format is the data persisted?*", "*Does the data follow any schema?*" or, equivalently: "*What is the meta-model of the data, if any?*", "*How are inconsistencies dealt with?*".



(a) Overview of the conceptual setting.



(b) Process.

Figure 8.4 Process of technology injection into Dandelion. Numbers in a refer to the activities in b.

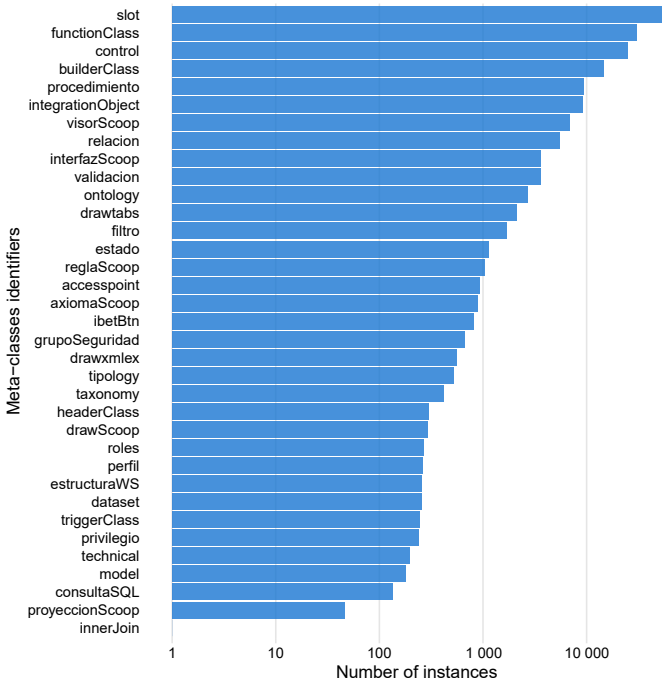


Figure 8.5
RQ3: Meta-classes frequency in ROSE data.

As a concrete example, the process is illustrated by injecting UGROUND's ROSE data into Dandelion. ROSE is an industrial model-based platform by the company UGROUND to develop business platforms, e.g., organizational digital twins. The types of elements ROSE works with are varied: from domain concepts and slots (i.e., fields/attributes) to business processes. All these components can be reused for different clients. So far, the data has been persisted in relational databases, but the company is migrating it to Elasticsearch, a document-based database, to improve performance and scalability. The data is distributed in multiple indices spanning millions of modeled concepts. For the scope of this experiment, one of these indices is considered, which contains more than 175 000 elements. Through communication, the language engineer elicits that the data is structured according to a technology-dependent schema (a *mapping* in Elasticsearch terminology) aiming to represent different model elements uniformly. However, the data is not governed by any explicit meta-model.

A preliminary inspection of the data permits identifying the types that elements conform to, as well as their number of instances, as shown in Figure 8.5. The figure depicts the number of instances of each meta-class, together with its frequency (please note the logarithmic scale of the X axis). This information can be used to focus on the most important meta-model elements first, deferring other elements to subsequent iterations.

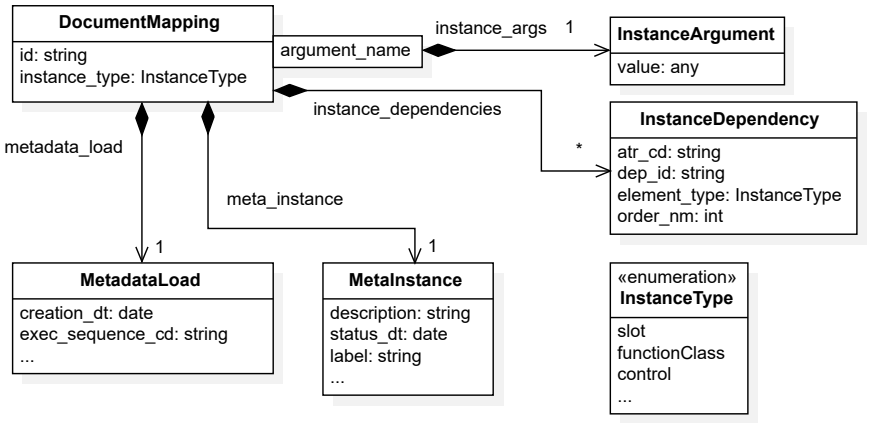


Figure 8.6
RQ3: The Elasticsearch mapping for ROSE data.

2. Schema injector design Once the LCDP engineer transfers all the pertinent knowledge about the data to the language engineer, the latter can derive a schema injector.

A schema injector is a piece of code that, when given access to data in a foreign technology, extracts its underlying meta-model. The difficulty of this step lies in how the data is structured. For modeling languages such as UML or EMF, establishing canonical mappings between their primitive modeling concepts and those of Dandelion is straightforward. Another common use case is injections from relational or document-based databases. In these cases, table definitions and document schemas can be leveraged. Note, however, that these usually do not suffice, as SemanticNodes, Models, and properties may be represented vastly differently even while conforming to the same schema. That is why elicitation is crucial; only that way can the language engineer precisely pinpoint the mappings to be established. Finally, the schema injector may result in an incomplete meta-model. To support this scenario, the strictness of model conformance can be relaxed through the tool's flexible modeling capabilities. Multiple iterations of the process can also further refine the meta-model.

In ROSE, the main asset for constructing the schema injector is the Elasticsearch mapping, which determines how documents are stored and indexed, and the fields they consist of. Figure 8.6 presents the schema used to persist ROSE artifacts.[2]

Each document in the database conforms to a DocumentMapping, which is identifiable by the pair (id, instance_type). The instance_type assigns a type to the document, which is drawn from the keywords identified in Figure 8.5. Documents embed properties in instance_args and instance_dependencies, and meta-data in MetadataLoad and MetaInstance.

[2]Elasticsearch mappings are expressed in a domain-specific JSON format. An approximate, simplified UML version of it is shown here for readability purposes.

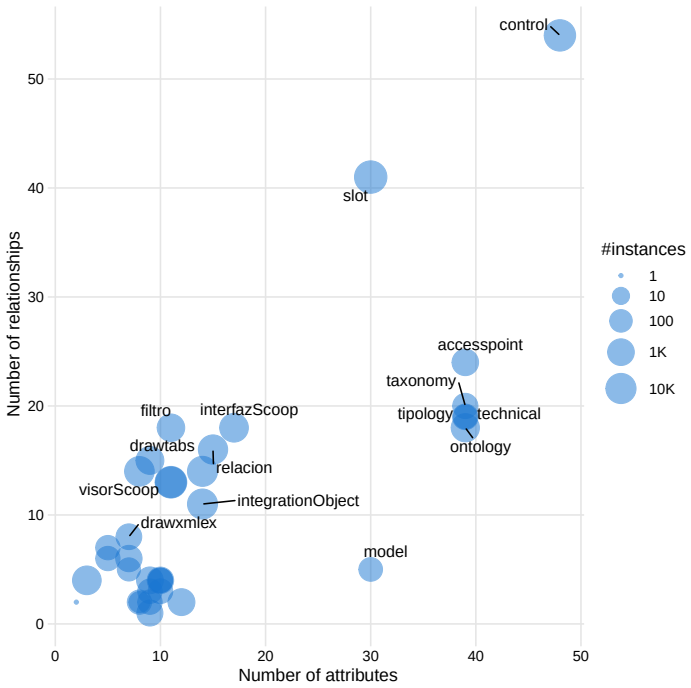


Figure 8.7
RQ3: Number of attributes and relations per meta-class in the resulting meta-model.

The following paragraphs explain how.

First, documents can express unconstrained valued attributes via instance_args. This is implemented as a map of key-value pairs, with the key being the property's name, and the value being of any type. Second, relations to other documents are indicated in instance_dependencies. In particular, dependencies have a name, atr_cd, and use the pair (dep_id, element_type) as the foreign key to point to the target document. The field order_nm is used to sort the dependencies when displayed in the user interface. Finally, MetadataLoad and MetaInstance encode meta-data, some of the fields being relevant to the ROSE execution engine (i.e., its interpreter). The most important ones are MetaInstance.description, a human-readable description —the name— of the document, and MetadataLoad.creation_dt, its creation date.

To create the schema injector, some additional information was required, as the data model lacks specificity in the properties. Additional queries were executed to determine the specific attributes and relations per InstanceType. In particular, it is necessary to know which InstanceArgument keys and which InstanceDependency.atr_cd populate each InstanceType. Figure 8.7 summarizes the results of these queries, relating three metrics per meta-class: the number of attributes, relations, and instances.

The resulting meta-model contains, therefore, as many SemanticNodes as identified meta-classes (i.e., InstanceTypes in the Elasticsearch mapping).

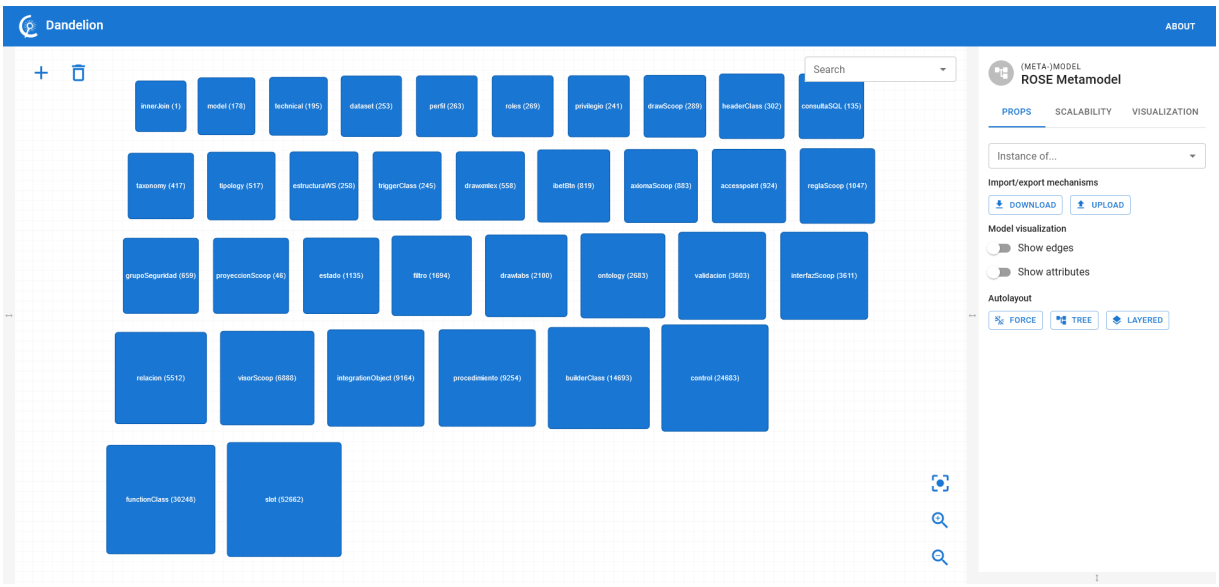


Figure 8.8 RQ3: The ROSE meta-model in the tool's editor. Edges and attributes have been hidden for readability. In parenthesis, the number of elements instantiated per meta-class.

Regarding properties, InstanceArgument gets mapped to DataProperty, where the name is argument_name, and the type depends on the argument's name, as it also encodes this information. For example, an InstanceDependency.atr_cd valued "column_nm" contains two pieces of information: the property's name, column, and its type, numeric. Moreover, InstanceDependency becomes ObjectProperty, where the target is determined by InstanceDependency.element_type. Finally, multiplicities are set to 0..1 in DataProperty, as InstanceArguments are single-valued, and 0..* in ObjectProperty, as InstanceDependency instances are multi-valued. Both cases support the absence of property values (i.e., their lower bound is 0), as the ROSE data model does not enforce their presence.

3. Schema injector use Once the schema injector is implemented, it can be used to extract a meta-model, which can be imported into the tool.

The resulting ROSE meta-model is highly complex —including dozens of meta-classes with many attributes and relations each— so the conventional meta-model representation is unsuitable. Instead, it is presented in a compact view where relations are hidden, and meta-classes are displayed in nodes whose size is proportional to the number of elements that instantiate them (see Figure 8.8). This quantity is also shown in the labels of the nodes in parentheses. Thanks to this integration, users can now use the tool to operate on the meta-model, inspecting its attributes and relations, or obtaining attribute recommendations over meta-classes.

4. Data injector design Although users can already use the meta-model in the tool, e.g., by instantiating or modifying it, existing models of the foreign technology may need to be migrated into Dandelion, too. For that, a data injector is needed, which permits extracting data from a foreign technology to make it conform to a Dandelion meta-model. This also permits leveraging the tool features such as pagination and flexible modeling support.

In the case of ROSE, each document yields a SemanticNode where instance_args and instance_dependencies are mapped to DataProperty and ObjectProperty, respectively. From the meta-data, only MetaInstance.description is preserved as the name of the SemanticNode, since the rest of the fields are not relevant within Dandelion. Conformance to the extracted meta-model is achieved via TypedElement.types. Specifically, SemanticNodes point to the corresponding meta-class dictated by their instance_type. Similarly, properties conform to those defined in the meta-class. The argument name is used for instance arguments, whereas atr_cd (the attribute name) is employed for instance dependencies. Finally, ROSE already has an InstanceType called model that nicely fits with the Model concept in Dandelion. In particular, elements target the models that contain them via the InstanceDependency of father_cd. This maps to Model.parent in Dandelion.

5. Data injector use With an operating data injector, users can now navigate models coming from an external technology. This does not impact the navigation experience, as the tool automatically applies the appropriate data injectors. Moreover, models created within the tool can interoperate normally with the injected models as long as they conform to the same (injected) meta-model.

A ROSE model on bond insurances is used as a running example. The model contains 10 287 domain concepts. The most frequent types are slots (6 028), which define form fields, and functionClasses (3 798), which describe the behavior of these forms in a ROSE-specific language. The model also contains accesspoints and roles to scope and restrict access to these forms.

Figure 8.9 shows a screenshot of the model which has been loaded, using pagination. To ease the exploration of models, dedicated visualizations have been created for each domain concept (i.e., meta-classes) through a concrete syntax based on icons.

6. Conformance analysis Designing injectors is generally a manual, error-prone task that demands a deep understanding of the technologies they bridge. Therefore, LCDP and language engineers may introduce inconsistencies in the process. In this last step, the tool's flexible modeling capabilities (cf. Section 4.2.5) are leveraged to ensure the soundness of the injectors and detect inconsistencies in the original data.

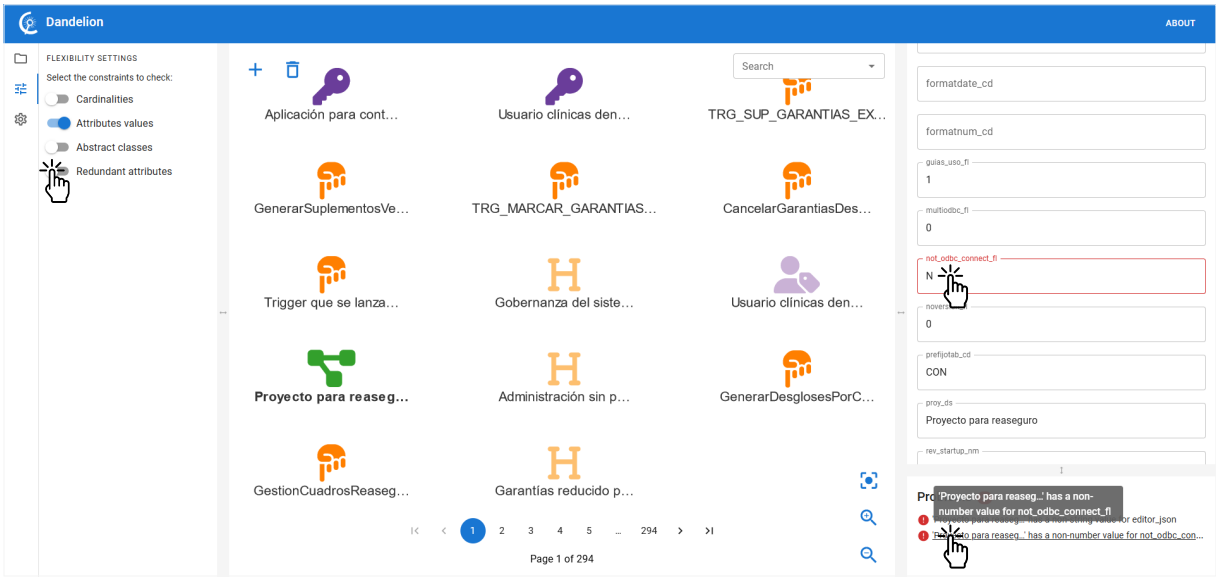


Figure 8.9 RQ3: A ROSE model in Dandelion with raised conformance issues.

An appropriate selection of the constraints to be checked by the flexible conformance functionality permits better adaptation to the constraints imposed over data in the foreign technology. Practically, this means going back and forth and iteratively fixing the injectors and fine-tuning the conformance checks. Note that Dandelion can work with all types of model conformances: from strict, full conformance to the meta-model, to disabling all the checks. The language engineer has the flexibility to choose the level of conformance that best fits the DSL needs. This feature can also help mitigate the following issues:

Injection construction errors Although the schema and the data injectors are designed separately in the process (and therefore can introduce errors that cancel out), initially all the consistency checks are enabled. That means that errors introduced in one of the injectors but not in the other will be detected and raised. Unlike other modeling tools, Dandelion does not impede loading non-conformant models. It instead loads them and reports the found issues. This leads to an iterative process where the language engineer can incrementally fix the injectors until achieving the desired level of conformance.

Original data errors Even if the foreign technology imposes constraints, some may be already violated in the existing data. Likewise, flexible modeling can be exploited to detect errors in the original data and act appropriately. For instance, the language engineer can ignore them, and Dandelion will load the models while reporting them; solve them

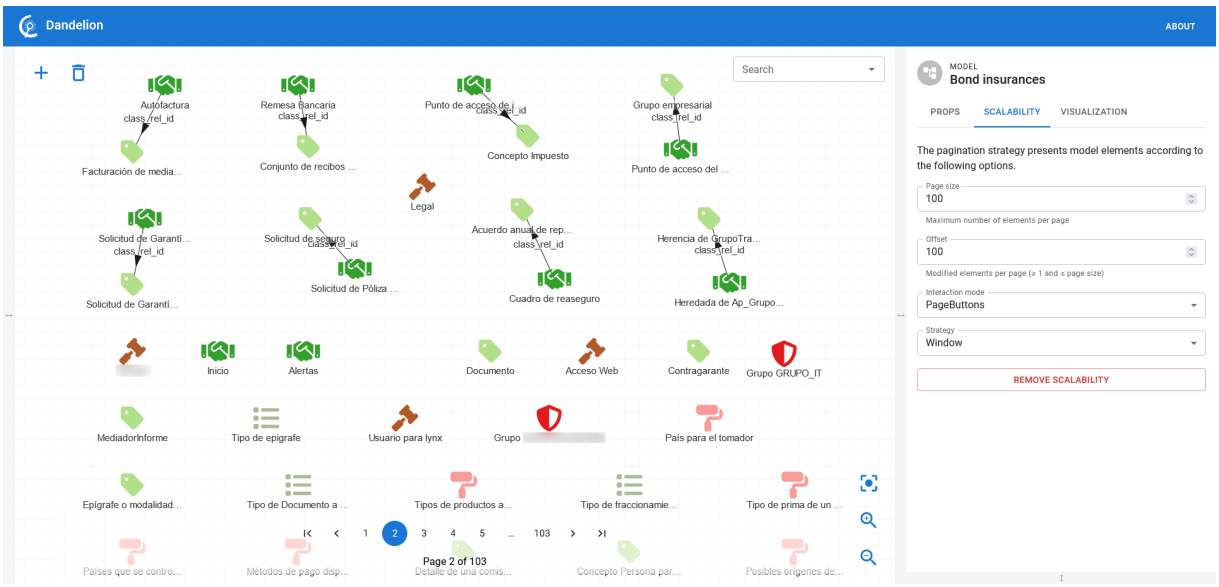


Figure 8.10 ROSE model visualized in Dandelion using pagination.

at the data level; or relax the constraints to omit the errors.

In ROSE, all the checks have been activated in order to identify errors in the existing data and ensure that the injectors are well scoped. Figure 8.9 shows how the tool has detected non-conformities in the original data. It reports that the selected element has the wrong type in one of its properties.[3] When clicked on the raised problems, the editor automatically selects the faulty element and highlights the wrong properties for the user to fix them.

To answer RQ3, a process has been devised to integrate a foreign technology into Dandelion, and it has been illustrated with UGROUND's ROSE. The experiment illustrates a worst-case scenario, where the data —over which the integrator has limited control— do not expose an explicit meta-model. Data is coupled with business logic that must be uncovered and understood to be preserved throughout the injectors. Integrating Dandelion with strict, well-defined formats such as EMF is much simpler to design. In fact, Dandelion offers an out-of-the-box EMF injector.

Regarding the visualization of large industrial models, UGROUND models have been visualized using pages of size 100, as seen in Figure 8.10, with loading times of $3\,130 \pm 1\,100$ ms per page, achieving a reactive experience. In view of these results, RQ3 can be answered by stating that Dandelion makes it possible to inspect large industrial models of UGROUND.

[3] Only the constraint check that is raised in the elements on the explorer has been enabled for clarity of the screenshot. In practice, all the checks are enabled.

8.1.4 Threats to validity

Next, the main threats to validity of the experiments are discussed.

Construct validity Construct validity refers to the degree to which the measures used in a study accurately reflect the concepts they are intended to capture. Regarding the first set of experiments (RQ1, RQ2), the main threat to validity is that they have been conducted with the backend deployed on the same local machine as the frontend. Therefore, measured times only account for the communication between the database and the backend, thus ignoring the frontend–backend communication. The results of the experiments are still considered sound, as each time measurement only omits two frontend–backend calls (i.e., a `getNFirstChildren` command and its response), which would typically contribute a constant overhead (~ 30 ms for an average Internet connection).

While experiments have been performed to evaluate pagination, these rely on measuring loading times of both synthetic and realistic models. However, to assess the scalability mechanism from a user perspective, a user study of Dandelion would be needed, which is left for future work.

Internal validity Internal validity is the degree of confidence that a causal relationship exists between the conducted experiment and the extracted conclusions. To avoid spurious results in the experiments, each measure was repeated 10 times and the median taken. Moreover, to avoid any bias that the manual creation of the synthetic models may have introduced, the models were generated randomly according to a sensible probability distribution.

External validity External validity concerns the extent to which the findings of a study can be generalized beyond the specific experimental setting. The first two experiments used synthetic models, instances of a particular meta-model. Pagination is expected to behave similarly for instances of other meta-models, and, to mitigate this threat, realistic models of the UGROUND company were used in RQ3. However, a higher ratio of references per node would probably lead to a higher proportion of proxy nodes per page, which may affect the user experience of the pagination mechanism. Future work includes incorporating element sorting criteria for pages.

Moreover, the distribution of objects in the synthetic log models was determined experimentally. However, models in other domains may take other shapes. It is up to future work to evaluate the pagination mechanism with models that have other object distributions.

Also concerning RQ3, UGROUND's modeling technology was injected into Dandelion. This experience can be extrapolated to other existing technologies as well. The fact that no explicit meta-model was defined in

this case is possibly one of the hardest scenarios. Moreover, only industrial models of the UGROUND company were tested. Future work should experiment with artifacts of other technologies.

8.2 Sensemaking strategies evaluation within Dandelion

This section evaluates the SMS approach presented in Chapter 5 by demonstrating that SMSs can be used to understand large industrial low-code ecosystems. In particular, UGROUND's modeling ecosystem is examined, consisting of the ROSE meta-model [40] and large model instances, some of which contain more than 175 000 elements. The following research questions (RQs) are addressed:

RQ4 *Do SMSs help to understand the ROSE meta-model?*

RQ5 *Can SMSs be used to understand how ROSE is used in practice?*

RQ6 *Can SMSs be used to understand ROSE models?*

Each RQ examines the utility of SMSs for gaining insight into the modeling ecosystem from a different angle. Specifically, RQ4 (Section 8.2.1) assesses the efficacy of SMSs for understanding a complex meta-model, a useful task for novice language users and language engineers. RQ5 (Section 8.2.2) considers the meta-model together with all its instances and evaluates whether SMSs can be useful for language designers to analyze how a DSL is used in practice and guide its possible evolution. Finally, RQ6 (Section 8.2.3) focuses on concrete models, evaluating whether SMSs help create visualizations tailored to their semantics, a task that is helpful for regular language users who may need support to understand a large model (possibly built by a third person).

To structure the evaluation, a set of sensemaking tasks (STs) is introduced and associated with multiple SMSs, as summarized in Table 8.1. These were identified in collaboration with UGROUND engineers and represent recurrent tasks to enhance the understanding of the entire modeling ecosystem.

8.2.1 RQ4: Understanding complex meta-models

Comprehending meta-models is a frequent activity in the modeling process and typically entails answering sensemaking tasks ST1–ST3:

Table 8.1 RQ4–RQ6: SMSs used in the evaluation of the UGROUND ecosystem.

	ST	#SMS	Sensemaking question	SMS type
RQ4	ST1	1	*How many concepts are there?*	Free metric
		2	*Which types of concepts are there?*	Categorical
	ST2	3	*Which attributes are there?*	Literal metric
		4	*Are concepts data or connection-centric?*	Numerical
	ST3	5	*Which concepts are coupled?*	Connectivity
RQ5	ST4	6	*What is the distribution of concepts?*	Categorical
	ST5	7	*What is the instantiation range of fields?*	Categorical
RQ6	ST6	8	*Error codes of objects?*	Literal metric
	ST7	9	*States of Scoop rules (reglaScoop)?*	Categorical
	ST8	5	*Which concepts are coupled?*	Connectivity

ST1. *What are the concepts defined in the meta-model?*

ST2. *Which attributes do concepts define?*

ST3. *What are the relationships between concepts?*

Meta-models are typically built using editors that are not explicitly designed for sensemaking. For example, graphical editors for meta-models usually rely on a graph-based visualization metaphor, which is effective for small meta-models but does not scale well for large ones (cf. Figure 5.1). SMSs can help alleviate this problem.

To address STs 1–3, SMSs #1–5 are proposed and detailed in Table 8.1. One or more SMSs are used to answer each ST, as sensemaking theory suggests that sensemaking tasks benefit from a variety of visualizations [144]. Figure 8.11 shows SMSs #1–4 applied to UGROUND's ROSE meta-model (left in Figure 5.1), and Figure 5.10 displays SMS #5.

These SMSs allow answering STs 1–3. For ST1, SMS #1 counts the number of concepts, and SMS #2 categorizes them into concrete concepts, abstract concepts, or enums. For ST2, SMS #3 summarizes 500+ attributes in a word cloud, highlighting four common attribute names. This suggests that many concepts support error control. Additionally, SMS #4 displays the number of attributes and references of each concept in a scatterplot. Complex concepts control and slot stand out, having a high, yet balanced number of attributes and references. Other concepts, like model, are data-centric, with a high number of attributes (> 30). This SMS provides a bidirectional interaction: selecting a concept in the plot highlights the corresponding concept in the modeling canvas, and vice versa. The analysis can be complemented, for instance, with SMS #5, which presents a *dependency structure matrix* to grasp the relationships between concepts. As it is sorted

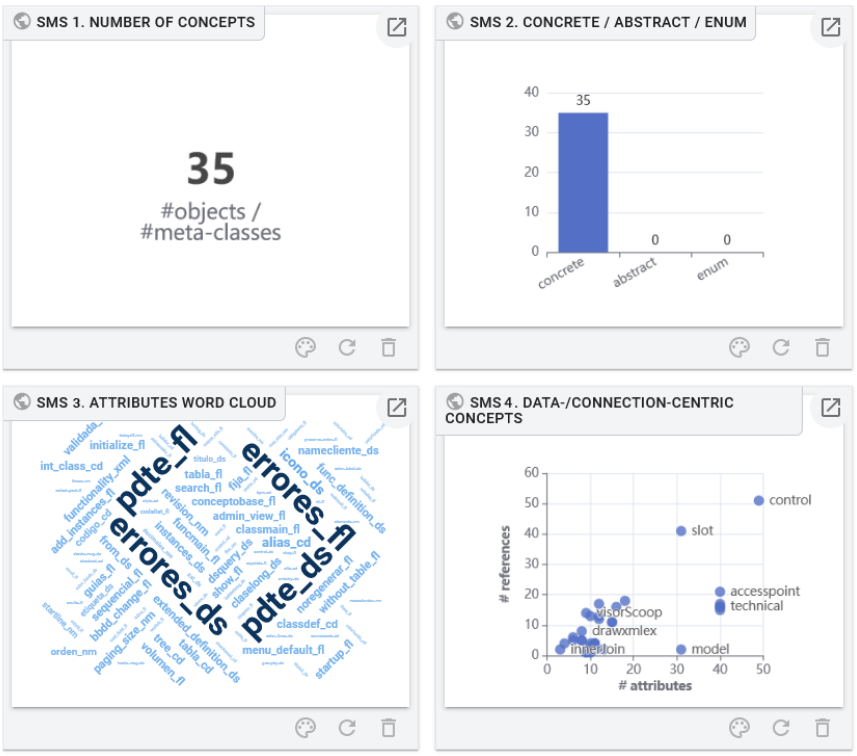


Figure 8.11
RQ4: SMSs #1–4 targeting ROSE meta-model.

by degree of incidence, it is suited to answer ST3. For example, it reveals that concepts like ontology, tipology, or accesspoint (the first ones in the axes) are highly coupled to other concepts.

These SMSs are domain-agnostic (i.e., they target the linguistic meta-model) and can therefore be reused for other meta-models, not necessarily within the UGROUND ecosystem. Some SMSs employ derived bindings, which are specified using the expression language. For example, SMS #2 categorizes concepts into ‘concrete’, ‘abstract’, or ‘enum’ by inspecting the isAbstract and isEnum properties of SemanticNode in Dandelion’s linguistic meta-model (Figure 5.9). In the case of ROSE, SMS #1 and SMS #2 reveal that most concepts in the meta-model are concrete. Combined with the prevalence of a small set of attribute names highlighted by SMS #3 (in particular, error-related fields), this suggests that a refactoring of the meta-model could be beneficial, e.g., by factoring out shared concerns into dedicated abstractions.

Answering RQ4, SMSs prove effective in understanding complex meta-models such as ROSE. They work synergistically when presented in a dashboard, providing a holistic and reactive exploration of the meta-model. Furthermore, the support for agnostic bindings and derived values enables the implementation of reusable SMSs for any meta-model, making them

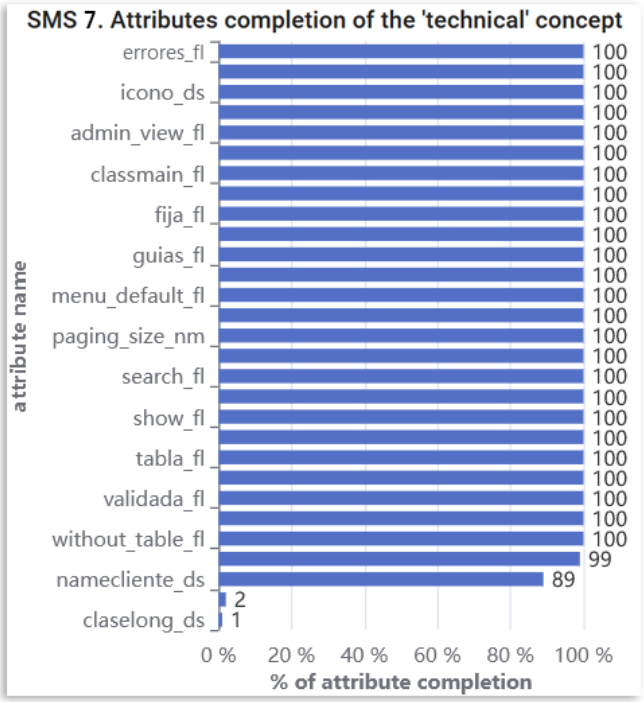


Figure 8.12
RQ5: SMS #7.

suitable for onboarding new team members into the language ecosystem.

8.2.2 RQ5: Understanding a modeling ecosystem

This RQ evaluates whether SMSs are useful for understanding how a meta-model is used in practice, including the contents and structure of the emerging modeling ecosystem, comprising meta-models and their instances (domain models). Two sensemaking tasks are considered to this end:

> **ST4.** *What is the distribution of concepts in the models?*
>
> **ST5.** *What is the instantiation range of fields?*

SMSs #6–7 aim to answer these questions by applying them to UGROUND's modeling ecosystem. SMS #6 (Figure 5.1, right) displays a prominent metric of the usage of a meta-model: the number of instances per meta-class, answering ST4. SMS #7 (Figure 8.12) focuses on specific meta-model concepts (e.g., technical in the figure) to understand the usage of its optional attributes. For instance, errores_fl with value 100 % means that every instance of technical populates this attribute.

This analysis can be leveraged for language evolution: highly used attributes can be designated mandatory, and scarcely used ones can be

deemed redundant. A similar analysis can be used for the instantiation range of references.

In the ROSE ecosystem, this kind of analysis highlighted scarcely used primitives (such as innerJoin) and typical instantiation ranges for individual fields, providing actionable input for the evolution of the DSL definition.

Overall, RQ5 can be answered positively. SMSs #6 and #7 are domain-agnostic, collecting data from model instances and not merely from the meta-model. Hence, they enable the creation of effective, reusable SMSs for any modeling ecosystem.

8.2.3 RQ6: Understanding specific models

Once the meta-model of a domain-specific language is understood, it is possible to devise tailored SMSs for it. SMSs #5, #8, and #9 have been applied to UGROUND models to resolve:

ST6. *Which error codes can be triggered by the objects in this model?*

ST7. *Which are the states of Scoop rules?*

ST8. *How are elements connected?*

ST6 targets the understanding of the error codes produced by the defined services within the model. A literal metric (a word cloud) is used to visualize them. Stop words are discarded to avoid noise in the visualization. This way, the user can quickly grasp the most common errors.

Scoop rules are used to describe the processes of the information system and are enacted by the Scoop rule engine. By understanding the state of these rules (Active, Draft, Obsolete, Pending, and others), the condition of the services within the model can be assessed. SMS #9 (cf. Figure 8.13) categorizes the states of Scoop rules (reglaScoop in the meta-model) by a estado reference. As multiple states can be grouped under the same term, the SMS aggregates them with a derived binding.

Finally, SMS #5, a dependency structure matrix, can be reused to solve ST8. This is possible because this SMS is domain-agnostic (i.e., like SMSs #1–5). The resulting visualization is identical to Figure 5.10, but the concepts in the matrix are the objects of the currently analyzed model.

Hence, RQ6 can be answered positively: a combination of domain-specific and agnostic SMSs can be used to understand specific ROSE models. In the current industrial setting, UGROUND plans to replace the Scoop engine and migrate its rules. SMSs such as #9 are particularly useful in this context, as they support the progressive analysis and evolution of rule sets during the migration.

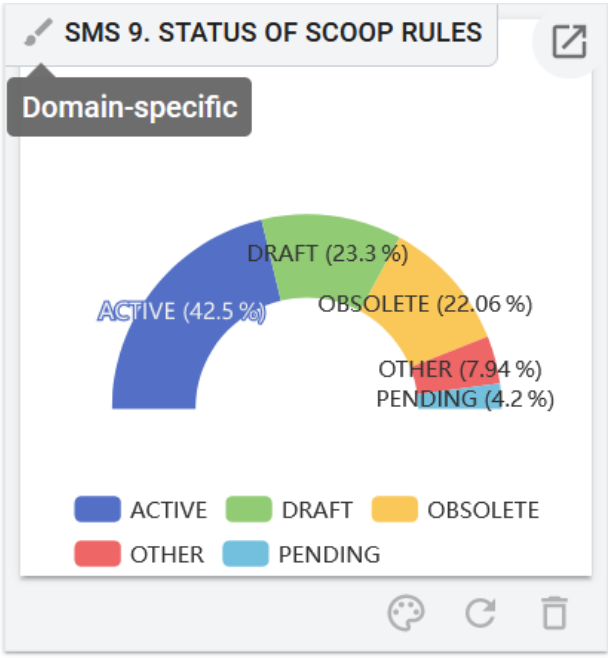


Figure 8.13
RQ6: SMS #9.

8.2.4 Threats to validity

Regarding threats to validity, SMSs have been used to analyze UGROUND's ecosystem. Since this is a complex industrial ecosystem, it is reasonable to expect that SMSs can also be used to understand other modeling ecosystems. The evaluation has been driven by the needs of UGROUND's employees when working with (meta-)models. Future work includes complementing this evaluation with user experiments measuring the efficacy of SMSs in resolving concrete sensemaking tasks.

8.3 Low-code development evaluation in Dandelion+

This section evaluates the competitiveness and suitability of Dandelion+ (Chapter 6) through these research questions:

RQ7 *How does Dandelion+ compare to other low-code platforms to build applications that require code generation?*

RQ8 *How can Dandelion+ improve UGROUND's industrial process to create low-code solutions?*

In the following, Section 8.3.1 explores RQ7, and Section 8.3.2 explores RQ8. Then, Section 8.3.3 discusses the threats to validity of the evaluation.

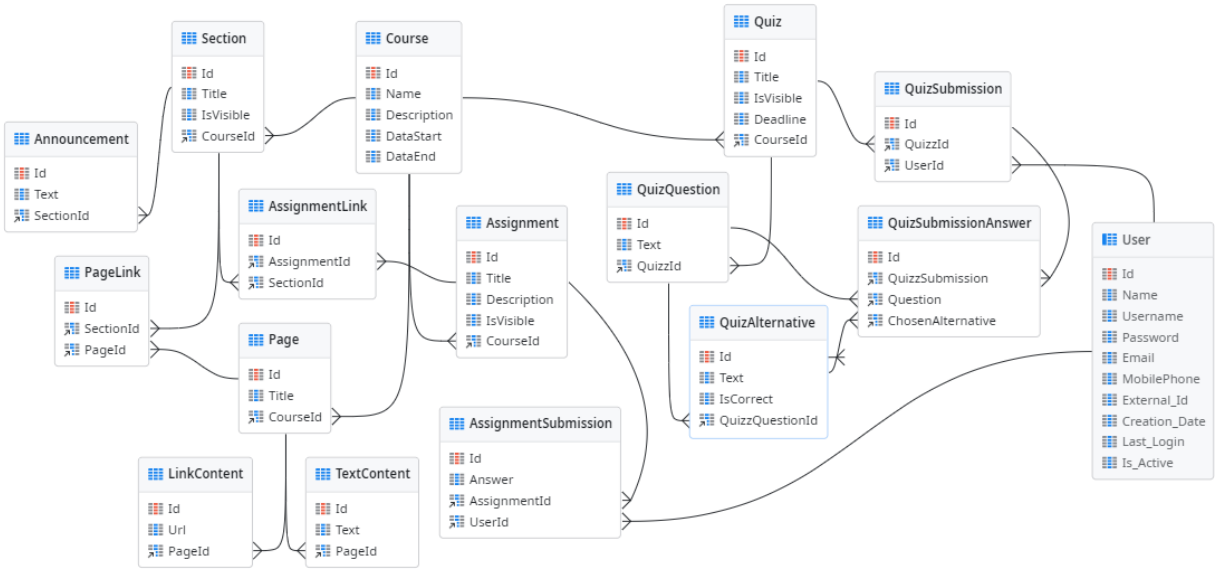


Figure 8.14 RQ7: Database schema of Mudel in OutSystems.

8.3.1 RQ7: Comparison with other low-code platforms

This section evaluates the use of Dandelion+ to build applications that require code generation. For this purpose, Dandelion+ is compared with state-of-the-art low-code platforms, as reviewed in Section 3.3.

Specifically, the Mudel running example from Section 6.2 is modeled using Dandelion+, and the resulting solution is compared with similar implementations in OutSystems [61], Mendix [169], and BESSER [179]. OutSystems and Mendix have been selected because they are fully-fledged low-code platforms widely used in industry. As for BESSER, it is included as it is one of the few academic low-code platforms, sharing a model-driven foundation with Dandelion+. Overall, the goal is to cover a broad spectrum of low-code platform features in the evaluation.

OutSystems

The Mudel domain has been implemented using Service Studio, OutSystems' desktop IDE.

Regarding the structure of the platform, its domain is modeled in the database schema shown in Figure 8.14. The schema mostly covers the Mudel domain model, with some limitations. First, as OutSystems relies on a relational persistence schema, inheritance relationships cannot be expressed natively. To circumvent this limitation, object-relational mapping strategies, such as single, class, or concrete table inheritance patterns [246] can be applied. Still, the semantics of the original model is not preserved. In

this case, the single table inheritance pattern is applied to SectionLink and MediaContent and their respective subclasses. Second, concepts FileContent and AssignmentAttachment (cf. Figure 6.7) cannot be represented with the available OutSystems components in the setup, as file uploads are not supported in the employed application configuration. Finally, object ownership can be modeled by importing User (an entity defined at a system level) and associating it with the ownable entities.

Regarding roles, all the proposed ones in Table 6.1 (i.e., Professor, TA, Student, and Personnel) have been defined. Concerning pages, a homepage, a course list, a course detail, as well as creation and delete screens for all entities have been implemented. Some pages had to be duplicated and adapted to cater to the roles in the platform. Figure 8.15 shows some screenshots of the application.

Regarding behavior, it was not possible to implement any of the workflows, as OutSystems server actions (the ones that manage data) can only execute SQL queries to perform complex operations. While SQL is a powerful language (and potentially Turing-complete), this approach is impractical for implementing the behavior of the platform, including scoring quizzes.

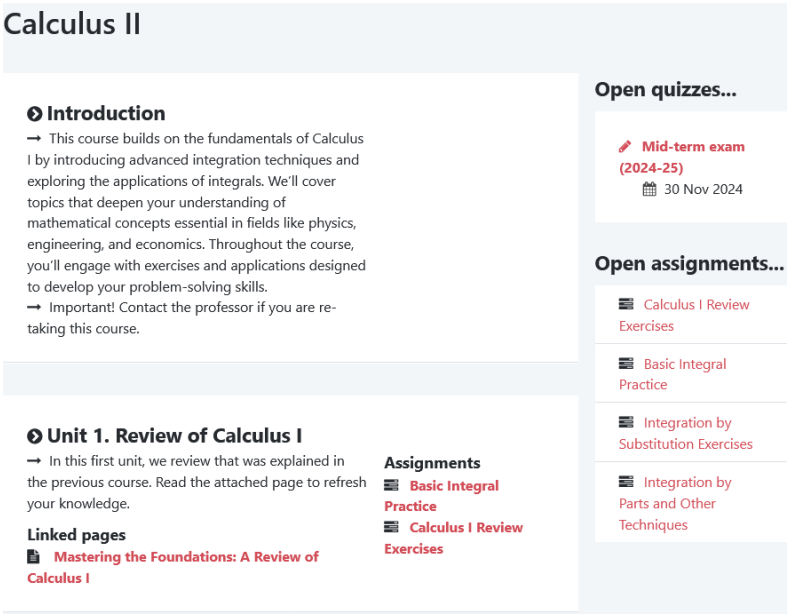
Mendix

For Mendix, Mendix Studio has been used. Regarding the structure of the platform, the entities are captured in the domain model shown in Figure 8.16. The schema covers the entirety of the Mudel domain. First, inheritance is supported for meta-classes like SectionLink and MediaContent through generalization. Second, FileSubmissions can be represented using the mechanism of generalization over the Mendix's system-defined concept of FileDocument. Third, some additional entities are required as temporary types, such as QuizGrading and UserCountHelper.

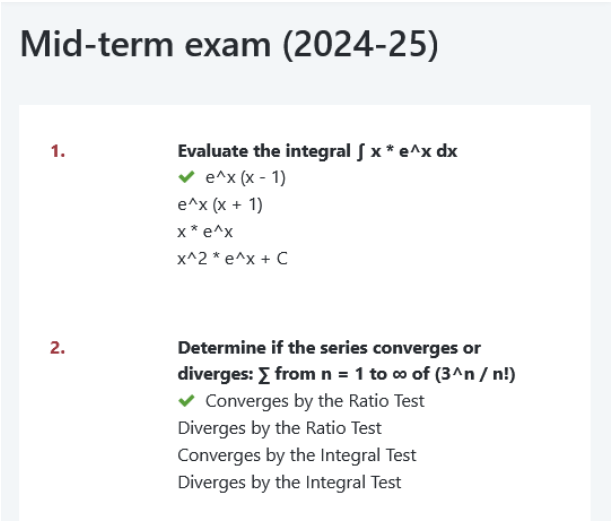
Regarding roles, Mendix has a similar approach to Dandelion+: all can be defined (from Professor to Personnel), each having a different menu and scope of executable workflows, as well as restricted pages.

Regarding pages, a homepage, a course detail view, a dashboard, and detail pages have been implemented for all the entities of the platform. Mendix automatically generates them with a "Generate overview pages..." option. In particular, in Figure 8.17a, conditional rendering for empty related assignments or related pages could not be replicated. In Figure 8.17b, it was possible to replicate the user metric and the course enumeration and detailing, but not the course timeline, as it is not an available graph.

Regarding behavior, it was possible to implement PDF generation for Workflow #3, although it was not possible to populate it with the required data. While Mendix has micro-workflows to process data, its facilities to implement business logic are very limited.



(a) Course detail (students, personnel).



(b) Quiz detail (professors).

Figure 8.15
RQ7: Pages defined in OutSystems.

BESSER

BESSER is a low-code tool that aims to "*model, generate, personalize and deploy smart and complex software systems*" [179]. Its approach starts with the user providing a meta-model expressed in B-UML (BESSER's Universal Modeling Language), from which code generators are executed to produce code targeting the desired technologies. Currently, meta-models can be imported through PlantUML, custom Python scripts, or images

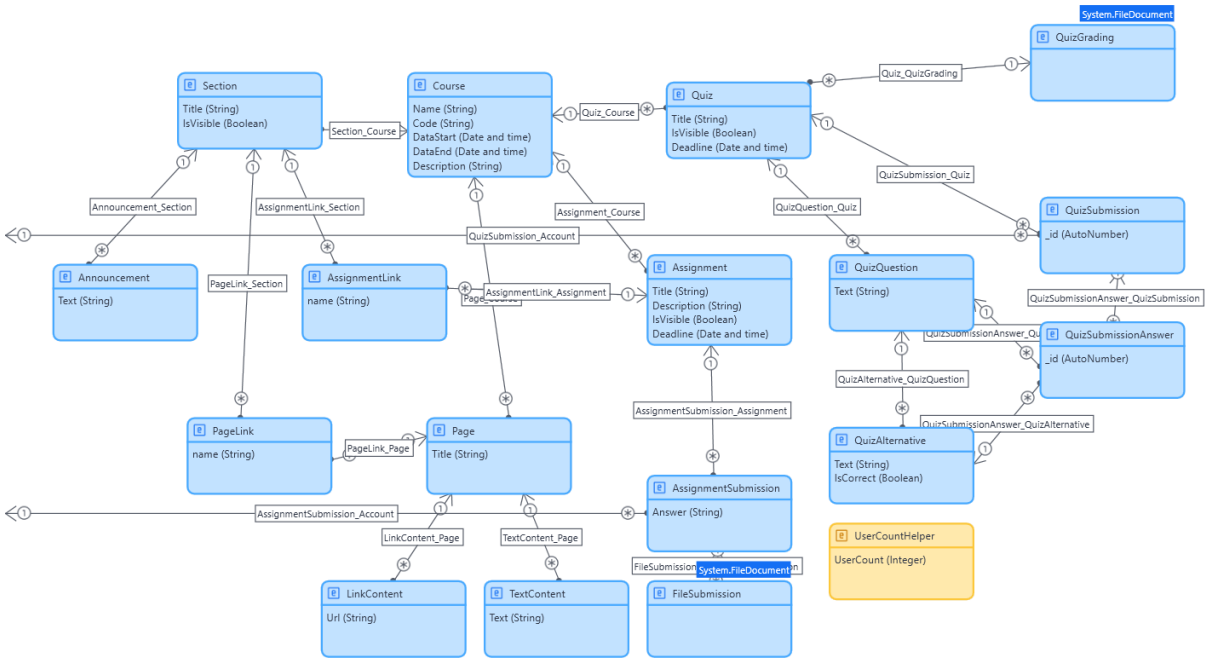


Figure 8.16 RQ7: Domain model of Mudel in Mendix.

of class diagrams. Regarding code generators, the tool supports multiple generators targeting, among others, Django, REST APIs with Python's FastAPI, database schemas, or Flutter applications.

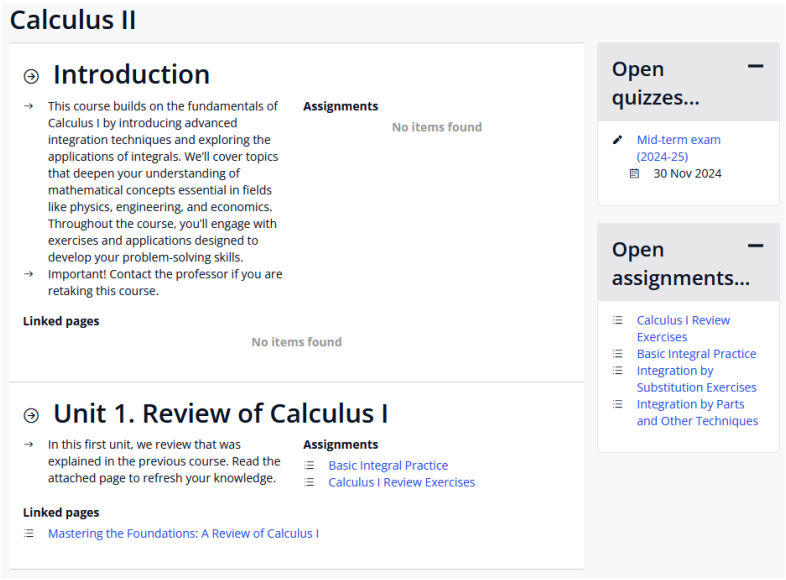
In this evaluation, the meta-model is expressed using PlantUML, as its semantics coincides with that of the class diagram employed in the running example. Regarding the generated artifacts, Django is used, as it produces web applications and automatically exposes an administration section (i.e., Django's admin site) that allows users to manage the application's data at runtime in a low-code fashion.

Regarding the meta-model, the one shown in Figure 6.2 has been replicated in PlantUML syntax with some modifications. In particular, the User class and the Role enumeration have been omitted, as these concepts are handled natively by Django. Additionally, the classes that manage files (FileContent and AssignmentSubmission) have been removed, as BESSER currently does not provide file fields,[4] although Django supports them natively.[5]

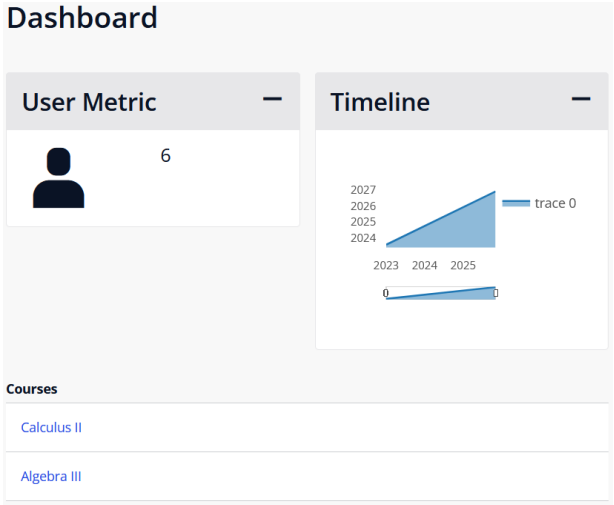
Regarding roles, Django manages them through groups. Groups Student, Professor, TA, and Personnel have been defined (cf. Figure 8.18).

[4] https://github.com/BESSER-PEARL/BESSER/blob/master/besser/generators/django/templates/django_fields.py.j2

[5] https://docs.djangoproject.com/en/5.1/ref/models/fields/#filefield



(a) Course detail (students, personnel).



(b) Dashboard (professors).

Figure 8.17 RQ7: Pages defined in Mendix.

Regarding pages, these are defined using views,[6] which typically load models from the database and render them using templates employing the Django template language,[7] which is similar to Jinja and placeholder-based as well. Figure 8.19 shows one of the generated pages out of a BESSER model.

Regarding behavior, to the best of our current knowledge, BESSER can only generate static information out of the input meta-model. Any

[6] https://docs.djangoproject.com/en/5.1/topics/http/views

[7] https://docs.djangoproject.com/en/5.1/ref/templates/language

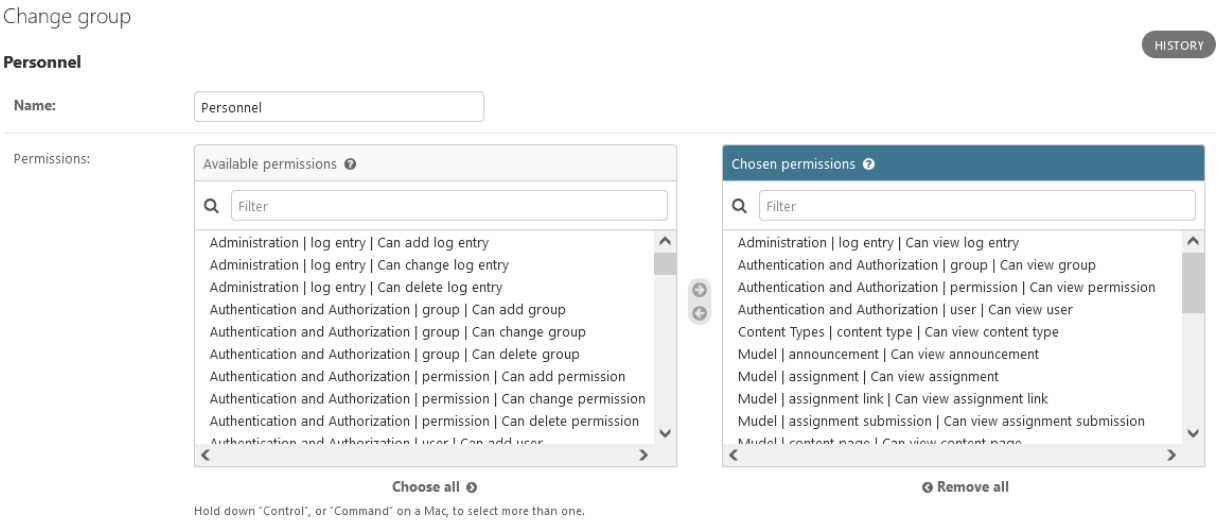


Figure 8.18 RQ7: Group (role) 'Personnel' defined in Django.

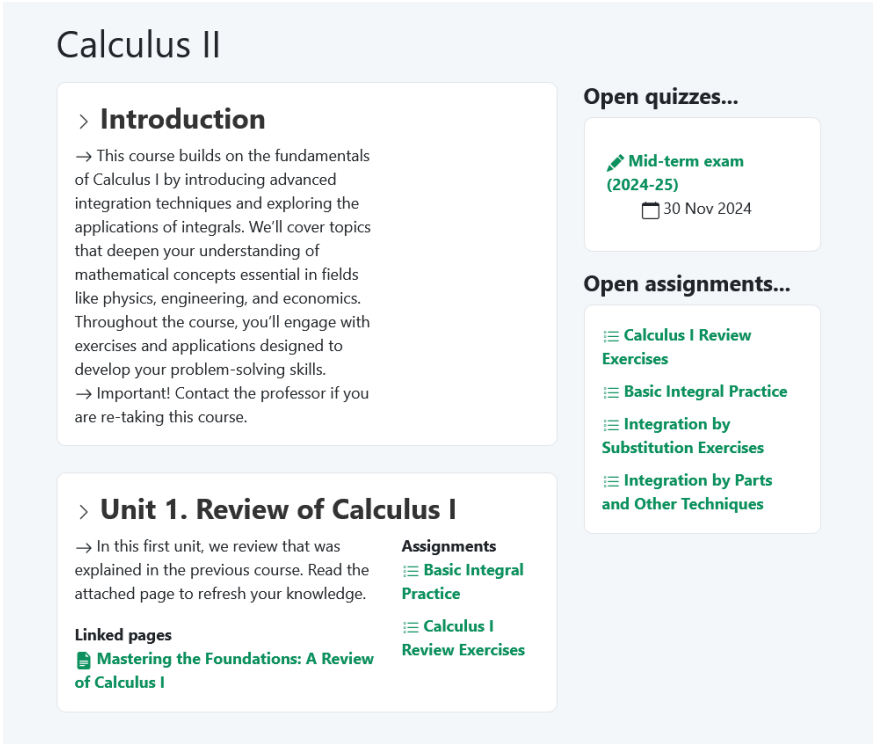


Figure 8.19 RQ7: Page defined in Django out of a BESSER model: course detail (students, personnel).

interactivity must currently be programmed using Django ORM in Python, which limits the extent to which BESSER supports a purely low-code approach.

Feature	OutSystems	Mendix	BESSER	Dandelion+
Domain model construction	DB schema	Object schema	Model-based	Model-based
Inheritance	◦	•	•	•
Object ownership	• (reuse entity)	• (generalization)	◦	• (annotation)
File persistence support	◦	•	◦	•
Graphical editor	•	•	•	•
Behavior specification	Proprietary workflows	Proprietary workflows	—	Open workflows
Model transformation	Insufficient	Limited	*(Before runtime)*	Full (MDE-based)
Custom code generation	Insufficient	Limited (PDF, HTML)	*(Before runtime)*	Full (agnostic)

Table 8.2 RQ7: Comparison of different low-code platforms for replicating the Mudel domain.

Discussion

Table 8.2 summarizes the main features of the studied low-code platforms for replicating the Mudel domain. Two main feature categories can be distinguished in this evaluation: domain model construction and behavior specification.

Regarding domain model construction, commercial LCDPs predominantly use a database schema or an equivalent approach, whereas BESSER and Dandelion+ adopt a model-driven methodology. The level of support for modeling features varies across platforms. Most LCDPs support inheritance and object ownership through different mechanisms. OutSystems achieves object ownership by reusing (i.e., importing) a predefined User table/entity at the system level. In contrast, Mendix and Dandelion+ implement ownership via "generalization" or annotations. File persistence support is more restricted, with only Mendix and Dandelion+ providing full support, while BESSER offers partial support. Finally, all platforms feature a graphical editor, facilitating the visual construction of the domain model.

Regarding behavior specification, all examined LCDPs with runtime behavior (OutSystems, Mendix, and Dandelion+) follow a workflow-based approach. However, the expressiveness of these workflows is generally very limited. Specifically, OutSystems and Mendix fail to fully reproduce Workflows #1–#3, as their workflow constructs and SQL query capabilities are insufficient to capture the required behavior. BESSER, which generates greenfield applications from a modeled specification, applies model transformation and custom code generation at design time rather than runtime. However, in a strict sense, this does not fully align with the definition of behavior specification, as behavior should be enacted at runtime. In contrast, Dandelion+ successfully generates all required artifacts using an MDSE approach, supporting both platform-agnostic model transformations and custom code generation.

In conclusion, RQ7 can be answered positively: Dandelion+ stands out over the other LCDPs when building applications that require code

generation. For structure, Dandelion+ provides key features for constructing complex domain models, such as inheritance, files, and object ownership. For behavior, Dandelion+ offers a more expressive workflow-based behavior specification toolkit, improving those of competing LCDPs that lack the necessary expressiveness for this use case.

8.3.2 RQ8: Improving UGROUND development process

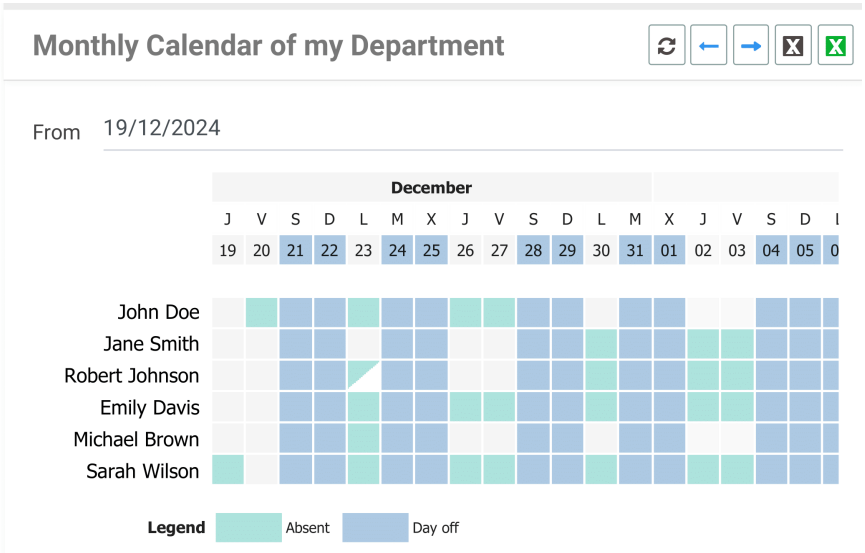
This section evaluates the advantages of Dandelion+ for building low-code solutions in an industrial setting. In particular, it compares UGROUND's organizational digital twin development environment with Dandelion+ workflows to assess the benefits of the latter.

Developing behavior in UGROUND

The development process of features in UGROUND mixes traditional programming with a model-driven approach. Namely, the company uses JCORE, a clone of JavaScript, to develop platform features, and ROSE for platform structure.

Figure 8.20 presents the monthly calendar report developed using UGROUND's platform: it tabulates the number of off and absent days per employee. The following focuses on the last action of the report, i.e., the green button with an Excel icon. When clicked, this action generates an Excel spreadsheet with the number of taken holiday days per month per employee, as seen in Figure 8.20. These buttons are called actions in UGROUND, and their creation or modification entails the following steps:

1. Pinpoint the page type and the action code In order to serve pages like Figure 8.20, UGROUND platforms do not execute JCORE code directly. Instead, an intermediate representation layer is generated automatically in a domain-specific XML format (cf. Listing 8.1). This markup language describes the visual components of the page, from which the frontend can render the appropriate HTML DOM using a templating system. In order to examine this intermediate code, the developer must open the log tracer of the platform (a desktop application). Three pieces of information are important here. First, the window is of type TSVDIR (line 2). This means that the implementation of this page/window is located at a tsvdir.cor file. Second, the page defines a control (i.e., a field) called START_DATE with a "From" label of type date (line 5). This control is used to set the start date of the holiday report, which is a raw HTML dump in line 13. Finally, the action of interest is located in line 11, and its procedure name bridges the visual component and its execution.



(a) Monthly calendar report within UGROUND's platform.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**Holiday Report 2024**												
2		**JAN**	**FEB**	**MAR**	**APR**	**MAY**	**JUN**	**JUL**	**AUG**	**SEP**	**OCT**	**NOV**	**DEC**
3	John Doe	3	4	4	1	0	0,5	0	9	0	1	0	3
4	Jane Smith	4	3	4	0	0	0	5	1	3	0,5	0	1,5
5	Robert Johnson	0,5	0,5	2	3	0	2,5	3	4	0	4	1,5	1,5
6	Emily Davis	0	0	3	0	0	0	0	16	0	0	0	4
7	Michael Brown	4	2	0	0	0	0	3	9	0	0	0	1
8	Sarah Wilson	1,5	1,5	4	1	0	0	5	5	1	0	0	5

(b) Exported Excel spreadsheet with holiday days per employee.

Figure 8.20
RQ8: Monthly calendar report in UGROUND's platform. Names have been removed to preserve privacy.

2. Modify/create the action in the class file Listing 8.2 shows the class tsvdir.cor identified in the previous step. Classes in JCORE are divided into three sections: data (line 2), page structure (lines 4–13), and functions (lines 17–26). Data are the fields of the current page. Page structure is described by constants and interfaces. In particular, constants define the placement of the visual elements (e.g., fields and controls). In the example, lines 4–7 are the code definition of Listing 8.1. Constants consume interfaces by name. Note that our action (line 13) is consumed by name in line 4, and the last parameter in line 13 points to the function to be executed.

In this example, the function (lines 17–24) creates an XLSX file out of a generated HTML report with the contents of the spreadsheet. This approach is followed as many JCORE libraries make extensive use of HTML as a source format from which to convert to other formats. Finally, the spreadsheet file is returned. If the structure of the generated Excel file (line 21) needs to be changed, then the function GetYear from class CalendarAux

Listing 8.1 Generated XML artifact as intermediate concrete syntax

```
1  <?xml version="1.0"?>
2  <Window type="TSVDIR">
3   <Title>Monthly Calendar of my Department</Title>
4   <Page>
5    <Control type="date" title="From"
         target="START_DATE">19/12/2024</Control>
6    <Control type="ToolBar" for="TSVDIR">
7     <bO procName="REFRESH" alt="Refresh" />
8     <bO procName="PREVVACATIONPERIOD" alt="30 previous days" />
9     <bO procName="NEXTVACATIONPERIOD" alt="30 next days"/>
10    <bO procName="EXPORTTOEXCEL" alt="Export to Excel"/>
11    <bO procName="EXPORTTOEXCELANNUAL" alt="Export to Excel Holiday
          Report" />
12   </Control>
13   <Control type="html"> ... </Control>
14  </Page>
15 </Window>
```

(located in calendaraux.cor) must be modified. As a reference, this file has 98 lines of imperative code. When a new action is added, the constant in line 4 has to be altered, and both an interface and a function have to be created and linked appropriately by name.

3. Push the changes and synchronize them in ROSE Once the changes are persisted in the version control, the developer accesses the ROSE platform in the browser and synchronizes the changes. By clicking on the 'synchronize' button, ROSE diffs the changes and updates its model about the interfaces and functions of the platform. If there are conflicts, they will be notified. At this stage, the developer may want to document the created or modified page structure elements.

4. Regenerate the code In order to enforce conformity between the model and the JCORE code, the code must be regenerated after committing changes through ROSE. In this step, static analysis is performed to raise issues, and a linter is applied.

5. Deploy the changes Once the code is regenerated, another developer from the platform-as-a-service (PAAS) department accesses the (pre)production servers and stops the running service. Next, they get the latest version of the code and restart the service. This completes the feature development cycle.

Dandelion+ approach

Dandelion+ uses workflows to replicate the same action.

Listing 8.2 File tsvdir.cor in JCORE

```
class TSVDir extends Window {
 data START_DATE

 constant PAGE_SHOW_DEPT_MONTH_CAL =
  "#date,From,START_DATE;" +
  "ToolBar,REFRESH,PREVVACATIONPERIOD, NEXTVACATIONPERIOD,
      EXPORTOEXCEL, EXPORTTOEXCELANNUAL;" +
  "#html,HTML_STR;"

 interface REFRESH("Refresh","m_repeat", SetUsrDeptMonthCalHTMLBtn)
 interface PREVVACATIONPERIOD("30 previous days", SetPrevVacationPeriod)
 interface NEXTVACATIONPERIOD("30 next days", SetNextVacationPeriod)
 interface EXPORTOEXCEL("Export to Excel", DoExportToExcel)
 interface EXPORTTOEXCELANNUAL("Export to Excel Holiday Report",
     DoExportToExcelAnnual)

 // ...

 function DoExportToExcelAnnual() {
  var objIoFile = new IO_FILE();
  objIoFile.SetFileName(Path_Make(Uuid(), "xls");
  objIoFile.AppendLn("<h3>Holiday Report
      %s</h3>".Format(Year(START_DATE)));
  var reportHtml = CalendarAux.GetYear(START_DATE);
  objIoFile.AppendLn(reportHtml);
  return ViewFile(objIoFile.FileName);
 }

 // ...
}
```

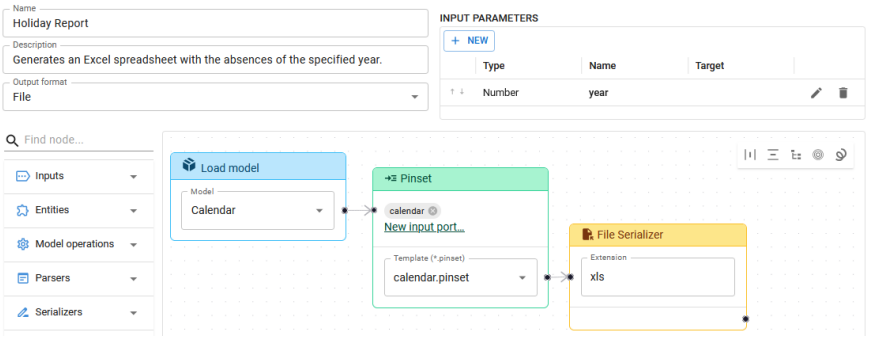


Figure 8.21
RQ8: Holiday report workflow in Dandelion+.

Figure 8.21 presents the holiday report workflow. The workflow takes a year as an input parameter, loads a calendar model, applies a Pinset node, and returns a file with the generated tabular content as an XLSX file.

Listing 8.3 shows the script of the Pinset node. This node consumes the provided model and generates a tabular structure with months as columns

and employees as rows. Each cell counts the number of absences (counting only half a point per half-day absence).

Listing 8.3 Pinset script for holiday report

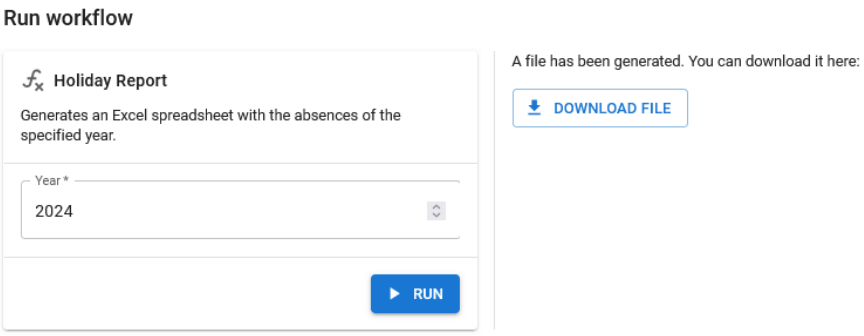
```
pre {
 var months = "JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC"
              .split(" ");
}

dataset HolidayReport over e : Employee {
 properties [name]
 grid {
  keys: months;
  header: key;
  body {
   var month = (months.indexOf(key) + 1).asString().pad(2, "0", false);
   var dateSuffix = month + "/" + year;
   var absences = e.absences.select(ab | ab.day.endsWith(dateSuffix));
   var halfAbsences = absences.select(ab | ab.isHalf).size();
   var fullAbsences = absences.size() - halfAbsences;
   var fullDays = fullAbsences + 0.5*halfAbsences;
   return fullDays == fullDays.round() ? fullDays.round() : fullDays;
  }
 }
}
```

Figure 8.22 shows the Excel file that is downloaded when the workflow is executed for the year 2024.

Discussion

Several differences and similarities can be observed between UGROUND's and Dandelion+'s approaches.

The following differences can be identified. First, in UGROUND, deploying changes implies regenerating code manually, stopping the platform, and restarting the system. In contrast, in Dandelion+, both the structure and behavior of workflows can be changed at runtime, allowing a rapid and dynamic development. Second, the approaches differ in platform containment. In Dandelion+, a web browser is the only necessary development tool, while in UGROUND, an IDE and a dedicated log tracer are required in a desktop environment. Third, Dandelion+ fully relies on model-driven principles, offering a set of model management languages of the Epsilon family. This allows using the most appropriate language for the task at hand. In the example, Pinset's declarative rules are used to map a model into a tabular structure, instead of relying on a lower-level programming language. Finally, the approaches manage format conversions differently. Like UGROUND, Dandelion+ can generate Excel files from models. However,



(a) Execution of the holiday report workflow in Dandelion+.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	name	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
2	John Doe	3	4	4	1	0	0.5	0	9	0	1	0	3
3	Jane Smith	4	3	4	0	0	0	5	1	3	0.5	0	1.5
4	Robert Johnson	0.5	0.5	2	3	0	2.5	3	4	0	4	1.5	1.5
5	Emily Davis	0	0	3	0	0	0	0	16	0	0	0	4
6	Michael Brown	4	2	0	0	0	0	3	9	0	0	0	1
7	Sarah Wilson	1.5	1.5	4	1	0	0	5	5	1	0	0	5

(b) Excel file generated by the holiday report workflow in Dandelion+.

Figure 8.22
RQ8: Holiday report in Dandelion+.

when processing existing files to produce other artifacts (i.e., the reversed data processing), Dandelion+ uses dedicated drivers (e.g., for JSON files) to yield operable models. This allows for targeted management through model-driven workflow nodes that use tailored DSLs.

Regarding their similarities, both approaches share a foundation in model-driven principles: the structure of their applications and actions (workflows) is represented and persisted using models. However, Dandelion+ goes a step further: while UGROUND duplicates structure in code and ROSE, Dandelion+ only keeps it in its linguistic model.

Overall, to conclude RQ8, Dandelion+ can improve UGROUND's industrial process in terms of rapid development, focused model management languages, and format flexibility, thereby streamlining the creation of low-code solutions that currently require manual coding and multi-step deployment.

8.3.3 Threats to validity

Internal validity In this evaluation, the conclusions are based on a comparison of solutions implemented by the authors across various low-code platforms, including the proposed one. Thus, especially for third-party tools, there is a risk of having overlooked alternative solutions, potentially more optimal in certain aspects. To prevent this from affecting the conclusions, all the tools used (OutSystems, Mendix, and BESSER) have been studied in

depth, and for the features under examination, the analysis supports that any features identified as missing in these tools are indeed absent, ensuring the validity of the conclusions at the time of this study.

Additionally, for RQ8, the description of UGROUND's process used in this work was validated by three employees who routinely follow this process.

External validity For RQ7, the comparison was conducted on the basis of a single case study (Mudel), which may not fully represent the diverse range of low-code-based applications requiring code generation. To mitigate this limitation, the case study was carefully designed to include instances of all key features discussed in the evaluation, such as access control, ownership, users, roles, and behavior. In any case, to further validate our results, additional case studies could be conducted, including industrial ones. Likewise, our comparison covered only a subset of the existing low-code platforms. To minimize this threat, two widely used professional low-code platforms were selected for the comparison, as well as a recent academic tool built atop model-driven principles. These three tools are considered highly representative of the broader landscape of low-code development platforms.

For RQ8, Dandelion+'s process for building low-code solutions was compared with that of UGROUND, based on detailed knowledge of this company's development process. However, other software companies may follow slightly different processes. Therefore, the conclusions for RQ8 may not be universally generalizable to all companies.

Finally, the extensive capabilities of Dandelion+ pose a trade-off between expressiveness and ease of use. While the platform enables developers to construct complex architectures, this flexibility may come at the cost of a steeper learning curve for those unfamiliar with model-driven principles. Therefore, some training or documentation may be necessary to use the tool effectively.

8.4 Summary and conclusion

This chapter has evaluated the three main prototype tools presented in this thesis —Dandelion, model sensemaking strategies (SMSs), and Dandelion+— mainly using industrial data from UGROUND.

First, regarding scalability, Dandelion's pagination mechanism was evaluated on synthetic models, showing that it can handle models with up to one million elements while providing reactive page load times. Dandelion was also applied to UGROUND's ROSE ecosystem, which contains more than 175 000 model elements. The results indicate that Dandelion is able to support a technology-injection process, showing that it can be applied in industrial scenarios.

Second, regarding SMSs, they were evaluated to understand the entire UGROUND modeling ecosystem. The evaluation reveals that, thanks to being domain-agnostic, SMSs prove reusable: they help make sense of complex meta-models and reveal how a language is used in practice. As a result, both language engineers and citizen developers can rely on these strategies to support the understanding and evolution of large modeling ecosystems, as well as onboarding new team members.

Third, regarding Dandelion+, it was evaluated against representative commercial and academic low-code platforms, using a learning management system (LMS) as a running example. The results indicate that Dandelion+ covers certain domain requirements, such as supporting inheritance, file management, and object ownership, which may not be covered by other platforms. Moreover, thanks to the workflow-based behavior specification of the tool, Dandelion+ is able to express complex processes, including code and artifact generation. Overall, Dandelion+ proved useful in an industrial use case at UGROUND, reducing development complexity.

Overall, the evaluations reveal that Dandelion, SMSs, and Dandelion+ effectively address several issues and particularities of low-code platforms through a model-driven approach: Dandelion focuses on scalability; SMSs on understanding large modeling ecosystems; and Dandelion+ on covering both the structural and behavioral aspects of low-code applications.

The next chapter closes the thesis by summarizing these results and presenting directions for future work.

Part IV

Conclusions

Chapter 9

Conclusions and Future Work

This chapter summarizes the main contributions and findings of this thesis (9.1) and outlines avenues for future work (9.2).

9.1 Conclusions

The previous chapters have introduced several tools and approaches that leverage Model-Driven Software Engineering (MDSE) to improve low-code platforms, both in terms of their design and execution. In the following, we detail our contributions, together with the conclusions drawn from them.

Firstly, this thesis has presented Dandelion (Chapter 4), a novel framework for creating cloud-based editors for graphical DSLs. The tool provides a scalable, cloud-based graphical language workbench for industrial low-code development. The main contributions of Dandelion are a harmonizing meta-model, a concrete syntax meta-model, scalability configurations, and support for flexible modeling. In particular, the harmonizing meta-model permits the incorporation of models coming from heterogeneous sources in a level-agnostic manner. The realization of this meta-model is backed by a persistence layer based on Elasticsearch, allowing for highly scalable persistence and querying of models. Additionally, the concrete syntax of the DSLs can be augmented with scalability configurations, which allow the visualization of large models through pagination. This mechanism allows the partitioning and visualization of large models in a few seconds for appropriate page capacities. Finally, the support for flexible modeling allows a finer control over the degree of conformance between models and their meta-models, thus facilitating their development by citizen developers. We have evaluated the approach against both synthetic and real industrial data in Section 8.1, showing that Dandelion can load models with up to one million elements using pagination and can be used to create frontends for industrial low-code platforms, where large models are imported and

visualized reactively using pagination. Overall, Dandelion demonstrates that grounding the linguistic and scalability aspects of low-code platforms on MDSE foundations allows scalable management of models in a low-code platform, both in terms of their persistence and visualization.

Secondly, this thesis has introduced the notion of model sensemaking strategy (SMS; Chapter 5), a reusable, generic component for creating flexible, insightful visualizations for large models. SMSs address the challenge of visualizing large, complex models, which frequently appear in many domains and for which the traditional graph-based visualization metaphor fails to scale effectively. Instead, SMSs exploit a model-driven approach suitable for multi-level modeling, working over models, meta-models, and entire modeling ecosystems. SMSs are instantiated via a binding to the —domain or linguistic— meta-model to which they are applied. We have proposed a catalog of 10 SMSs and 20 reusable visualizations spanning across different levels of abstraction in modeling ecosystems. SMSs have been integrated into Dandelion, together with a flexible binding language and a recommender that suggests suitable SMSs for a given meta-model. SMSs have been evaluated within UGROUND's large industrial ecosystem, showcasing their flexibility and feasibility for understanding complex meta-models, analyzing how a DSL is used in practice, and supporting the comprehension of specific models at different levels of abstraction (Section 8.2).

Thirdly, this thesis has introduced Dandelion+ (Chapter 6), a low-code platform for defining DSL-based platforms. Dandelion+ aims to cover the whole spectrum of low-code platform development, from specification to execution. The tool evolves Dandelion's model-based structure to support all elements of a low-code platform, including models, meta-models, users, and permissions. Regarding behavior, Dandelion+ supports the definition of workflows that specify how low-code platforms operate with their data. In particular, the tool introduces PlatFlow, a model-based workflow language catered to low-code platforms. We have evaluated Dandelion+ against other academic and non-academic low-code platforms and within a realistic industrial holiday-report scenario at UGROUND (Section 8.3). In these settings, Dandelion+ alleviated the flexibility and vendor lock-in limitations observed in the compared platforms, better supported applications requiring complex behavior and code generation, and improved UGROUND's industrial process for defining certain low-code components.

Finally, this thesis has presented LowcoBot (Chapter 7), a code generator for chatbots aimed at low-code platforms. LowcoBot is built following a model-driven approach, allowing it to capture the graphical, programmatic, and ontological sides of low-code platforms, from which a chatbot is generated. The tool derives automatic prompts and tools that help users get accustomed to, navigate, and use the platform. LowcoBot has been applied to Dandelion+, where it automatically generated both generic tools

(for summary, navigation, and exploration of platform capabilities) and platform-specific tools derived from endpoints and intents. This yielded a generated-artifacts-to-specification ratio of 10× in lines of code, significantly reducing the effort compared with an *ad hoc* implementation.

In conclusion, this thesis has covered a large spectrum of low-code platform features, including their definition, configuration, and usage. MDSE has been applied to the development of all of them, showing the suitability of the approaches to overcome existing limitations in the field. The proposals have been evaluated against use cases with an industrial partner, demonstrating the feasibility of the approaches for real-world scenarios.

9.2 Future work

This thesis has opened several lines of research regarding the symbiosis of model-driven engineering and low-code platforms. We now present the future work for the tools and approaches developed throughout this work.

Regarding Dandelion (Chapter 4), we aim to enhance scalability by improving the pagination mechanism through semantically aware strategies for element distribution and data mining techniques. Regarding modeling features, we plan to extend the concrete syntax with conditional styles and leverage LLMs to provide contextual quick fixes for flexible modeling. Finally, we are considering the adoption of the Graphical Language Server Protocol (GLSP) [107] to standardize communication and facilitate integration into open-source IDEs like Eclipse Theia [247].

Regarding Dandelion+ (Chapter 6), we aim to enhance both its structural and behavioral dimensions. Structure-wise, we intend to support the evolution of (meta-)models by integrating change-based persistence mechanisms [248] and model migration capabilities with tools like Epsilon Flock [249]. This would enable an incremental meta-model evolution while maintaining a trace of the changes and ensuring an efficient migration of existing models. Regarding behavior, we plan to support the definition of state machines for classes, as they are a key behavioral component in UGROUND's models. This extension would allow infusing a state into objects, enabling their lifecycle management. In addition, we plan to explore the feasibility of typing workflows, including a static analysis to detect structural errors at design time. Moreover, our template system currently computes all subpages when running an application. Instead, we plan to compute only the needed subpages on demand, as in [250]. Finally, we aim to complement this template system with a drag-and-drop visual editor, similar to those found in other low-code platforms, so that users can design their interfaces without coding.

Overall, as both Dandelion and Dandelion+ share the core of their linguistic meta-models, they can benefit from some common improvements.

For instance, we are working towards supporting the definition of integrity constraints (e.g., in OCL) to improve the expressivity of meta-models and ensure a better representation of the domains. Additionally, we plan to bridge both tools by porting Dandelion's capabilities for defining graphical concrete syntaxes into Dandelion+. This would allow manipulating artifacts using a more suitable representation —e.g., when domains have a graph-based or state-and-transition-based nature.

Regarding model sensemaking strategies (Chapter 5), we envision three lines of future work. First, while the current catalog focuses on generic and reusable visual metaphors (e.g., charts and treemaps), we intend to explore how to define highly specialized visual metaphors tailored to specific domains. This is particularly beneficial for domains with well-established visual representations, like circuit boards for electronics. Second, we plan to study how to integrate an LLM-based assistant that, given a description in natural language of a sensemaking task, automatically recommends a strategy and a binding. Finally, we aim to extend the current catalog with new graph-based SMSs, supporting more layouts and injectors from other formats, like knowledge graphs in RDF.

Finally, regarding LowcoBot (Chapter 7), we plan to support automatic endpoint extraction through Swagger/OpenAPI specifications [251], provide a concrete syntax for the tool configuration, and extend parameters for finer typing. We also aim to complement the current use case with user studies to assess the real-world usability of the generated chatbots. In particular, we intend to integrate LowcoBot directly into Dandelion+ to leverage the platform awareness of PlatFlow. This would allow modeling chatbots as first-class actors that understand platform concepts such as roles, permissions, and entity lifecycles. We also want to evaluate the proposed approach regarding LLM hallucination, and we intend to introduce defensive mechanisms whenever the chatbot is confronted with a question outside its scope. Finally, we plan to explore whether LowcoBot can be extended to generate assistants for (meta-)modeling tasks.

Capítulo 10

Conclusiones y Trabajo Futuro

Este capítulo resume las principales contribuciones y hallazgos de esta tesis (10.1) y esboza el trabajo futuro (10.2).

10.1 Conclusiones

Los capítulos anteriores han introducido varias herramientas y enfoques que aprovechan la ingeniería de *software* dirigida por modelos (ISDM, *MDSE* por sus siglas en inglés) para mejorar las plataformas *low-code*, tanto en términos de su diseño como de su ejecución. A continuación, detallamos nuestras contribuciones, junto con las conclusiones extraídas de ellas.

En primer lugar, esta tesis ha presentado Dandelion (Capítulo 4), un nuevo *framework* para la creación de editores basados en la nube para DSLs gráficos. La herramienta proporciona un entorno de trabajo para lenguajes gráficos, escalable y basado en la nube, para el desarrollo industrial *low-code*. Las principales contribuciones de Dandelion son un meta-modelo armonizador, un meta-modelo de sintaxis concreta, configuraciones de escalabilidad y soporte para modelado flexible. En particular, el meta-modelo armonizador permite la incorporación de modelos procedentes de fuentes heterogéneas de manera agnóstica al nivel. La realización de este meta-modelo está respaldada por una capa de persistencia basada en Elasticsearch, que permite una persistencia y consulta de modelos altamente escalables. Además, la sintaxis concreta de los DSLs puede ampliarse con configuraciones de escalabilidad, que permiten la visualización de modelos grandes mediante paginación. Este mecanismo permite la partición y visualización de modelos grandes en pocos segundos para capacidades de página adecuadas. Por último, el soporte para modelado flexible permite un control más fino sobre el grado de conformidad entre los modelos y sus meta-modelos, facilitando así su desarrollo por parte de desarrolladores ciudadanos. Hemos evaluado el enfoque con datos tanto sintéticos como industriales reales

en Apartado 8.1, mostrando que Dandelion puede cargar modelos de hasta un millón de elementos mediante paginación y puede utilizarse para crear *frontends* para plataformas *low-code* industriales, en las que los modelos grandes se importan y visualizan de forma reactiva mediante paginación. En conjunto, Dandelion demuestra que fundamentar los aspectos lingüísticos y de escalabilidad de las plataformas *low-code* en bases de la ISDM permite una gestión escalable de los modelos en una plataforma *low-code*, tanto en términos de su persistencia como de su visualización.

En segundo lugar, esta tesis ha introducido la noción de estrategia de *sensemaking* de modelos (SMS; Capítulo 5), un componente genérico reutilizable para crear visualizaciones flexibles y reveladoras para modelos grandes. Las SMS abordan el reto de visualizar modelos grandes y complejos, que aparecen con frecuencia en muchos dominios y para los cuales la metáfora tradicional de visualización basada en grafos no escala de forma eficaz. En su lugar, las SMS explotan un enfoque dirigido por modelos adecuado para el modelado multinivel, trabajando sobre modelos, meta-modelos, así como sobre ecosistemas de modelado completos. Las SMS se instancian mediante una asignación (*mapping*) al meta-modelo —de dominio o lingüístico— al que se aplican. Hemos propuesto un catálogo de 10 SMS y 20 visualizaciones reusables que abarcan distintos niveles de abstracción en los ecosistemas de modelado. Las SMS se han integrado en Dandelion, junto con un lenguaje de asignación flexible y un recomendador que sugiere SMS adecuadas para un meta-modelo dado. Las SMS se han evaluado dentro del amplio ecosistema industrial de UGROUND, demostrando su flexibilidad y viabilidad para comprender meta-modelos complejos, analizar cómo se utiliza un DSL en la práctica y apoyar la comprensión de modelos concretos, todo ello en distintos niveles de abstracción (Apartado 8.2).

En tercer lugar, esta tesis ha introducido Dandelion+ (Capítulo 6), una plataforma *low-code* para definir plataformas basadas en DSL. Dandelion+ tiene como objetivo cubrir todo el espectro del desarrollo de plataformas *low-code*, desde la especificación hasta la ejecución. La herramienta evoluciona la estructura basada en modelos de Dandelion para soportar todos los elementos de una plataforma *low-code*, incluyendo modelos, meta-modelos, usuarios y permisos. En cuanto al comportamiento, Dandelion+ soporta la definición de flujos de trabajo que especifican cómo las plataformas *low-code* operan con sus datos. En particular, la herramienta introduce PlatFlow, un lenguaje de flujo de trabajo basado en modelos adaptado a plataformas *low-code*. Hemos evaluado Dandelion+ frente a otras plataformas *low-code* académicas y no académicas, utilizando un sistema de gestión del aprendizaje como ejemplo de referencia, y dentro de un escenario industrial real de generación de informes de vacaciones en UGROUND (Apartado 8.3). En estos escenarios, Dandelion+ alivió las limitaciones de flexibilidad y bloqueo del proveedor observadas en las plataformas comparadas, ofreció un mejor soporte para

aplicaciones que requieren comportamiento complejo y generación de código y artefactos, y mejoró el proceso industrial de UGROUND para definir ciertos componentes *low-code*.

Además, esta tesis ha presentado LowcoBot (Capítulo 7), un generador de código para chatbots dirigido a plataformas *low-code*. LowcoBot se construye siguiendo un enfoque dirigido por modelos que permite capturar las dimensiones gráfica, programática y ontológica de las plataformas *low-code*, a partir de las cuales se genera un chatbot. La herramienta deriva *prompts* automáticos y herramientas que ayudan a los usuarios a acostumbrarse a la plataforma, navegar por ella y utilizarla. LowcoBot se ha aplicado a Dandelion+, donde ha generado automáticamente tanto herramientas genéricas (para resumen, navegación y exploración de las capacidades de la plataforma) como herramientas específicas de la plataforma derivadas de *endpoints* e *intents*. Esto ha dado lugar a una razón entre artefactos generados y especificación de 10× en líneas de código, reduciendo así significativamente el esfuerzo de desarrollo frente a un enfoque *ad hoc*.

En conclusión, esta tesis ha cubierto un amplio espectro de funcionalidades de las plataformas *low-code*, que abarca su definición, configuración y uso. La ISDM se ha aplicado al desarrollo de todas ellas, mostrando la idoneidad de los enfoques para superar las limitaciones existentes en el campo. Las propuestas se han evaluado frente a casos de uso con un colaborador industrial, demostrando la viabilidad de los enfoques en escenarios del mundo real.

10.2 Trabajo futuro

Esta tesis ha abierto varias líneas de investigación en torno a la simbiosis entre la ISDM y las plataformas *low-code*. A continuación, presentamos el trabajo futuro sobre las herramientas y enfoques desarrollados en esta tesis.

Con respecto a Dandelion (Capítulo 4), pretendemos mejorar la escalabilidad optimizando el mecanismo de paginación mediante estrategias de distribución de elementos sensibles a la semántica y técnicas de minería de datos. En cuanto a las características de modelado, planeamos extender la sintaxis concreta con estilos condicionales y aprovechar los LLM para proporcionar *quick fixes* contextuales en el modelado flexible. Finalmente, estamos considerando la adopción del Graphical Language Server Protocol (GLSP) [107] para estandarizar la comunicación y facilitar la integración en IDEs de código abierto como Eclipse Theia [247].

Con respecto a Dandelion+ (Capítulo 6), pretendemos mejorar tanto sus dimensiones estructurales como de comportamiento. En cuanto a la estructura, pretendemos dar soporte a la evolución de (meta-)modelos integrando mecanismos de persistencia basados en cambios [248] y capacidades de migración de modelos con herramientas como Epsilon Flock [249]. Esto

permitiría una evolución incremental del meta-modelo manteniendo una traza de los cambios y garantizando una migración eficiente de los modelos existentes. En lo relativo al comportamiento, planeamos dar soporte a la definición de máquinas de estados para las clases, ya que son un componente conductual clave en los modelos de UGROUND. Esta extensión permitiría dotar a los objetos de un estado, habilitando la gestión de su ciclo de vida. Además, queremos explorar la viabilidad de tipar los flujos de trabajo incorporando un análisis estático para detectar errores estructurales en tiempo de diseño. Asimismo, nuestro sistema de plantillas calcula actualmente todas las subpáginas al ejecutar una aplicación. En su lugar, queremos calcular únicamente las subpáginas necesarias bajo demanda, como en [250]. Por último, pretendemos complementar este sistema de plantillas con un editor visual de tipo *arrastrar y soltar* (*drag-and-drop*), similar a los que se encuentran en otras plataformas *low-code*, de modo que los usuarios puedan diseñar sus interfaces sin programar.

En general, dado que tanto Dandelion como Dandelion+ comparten el núcleo de sus meta-modelos lingüísticos, pueden beneficiarse de algunas mejoras comunes. Por ejemplo, estamos trabajando para dar soporte a la definición de restricciones de integridad (p. ej., en OCL) con el fin de mejorar la expresividad de los meta-modelos y garantizar una mejor representación de los dominios. Además, queremos tender un puente entre ambas herramientas portando las capacidades de Dandelion para definir sintaxis concretas gráficas a Dandelion+. Esto permitiría manipular artefactos utilizando una representación más adecuada —por ejemplo, cuando los dominios tienen una naturaleza de grafo o de estados y transiciones.

Con respecto a las estrategias de *sensemaking* de modelos (Capítulo 5), trazamos tres líneas de trabajo futuro. En primer lugar, si bien el catálogo actual se centra en metáforas visuales genéricas y reutilizables (p. ej., gráficos y mapas de árbol), pretendemos explorar cómo definir metáforas visuales altamente especializadas y adaptadas a dominios específicos. Esto es particularmente beneficioso para dominios con representaciones visuales bien establecidas, como los circuitos en electrónica. En segundo lugar, planeamos estudiar cómo integrar un asistente basado en LLM que, dada una descripción en lenguaje natural de una tarea de *sensemaking*, recomiende automáticamente una estrategia y un *binding*. Finalmente, pretendemos ampliar el catálogo actual con nuevas SMS basadas en grafos, dando soporte a más *layouts* e inyectores de otros formatos, como grafos de conocimiento en RDF.

Por último, en relación con LowcoBot (Capítulo 7), planeamos dar soporte a la extracción automática de *endpoints* a partir de especificaciones Swagger/OpenAPI [251], proporcionar una sintaxis concreta para la configuración de la herramienta y ampliar los parámetros para un tipado más fino. También pretendemos complementar el caso de uso actual con

estudios de usuarios para evaluar la usabilidad en el mundo real de los chatbots generados. Fundamentalmente, tenemos la intención de integrar LowcoBot directamente en Dandelion+ para aprovechar la consciencia de plataforma de PlatFlow. Esto permitiría modelar los chatbots como actores que comprenden conceptos de la plataforma como roles, permisos y ciclos de vida de las entidades. También queremos evaluar el enfoque propuesto en lo relativo a la alucinación de los LLM, y tenemos la intención de introducir mecanismos defensivos siempre que el chatbot se enfrente a una pregunta fuera de su ámbito. Finalmente, planeamos explorar si LowcoBot puede ampliarse para generar asistentes para tareas de (meta-)modelado.

Part V

Bibliography

References

[1] I. Sommerville, *Software Engineering, Global Edition*, 10th edition. Harlow, Essex, England: Pearson Education Limited, 2016.

[2] P. Mohagheghi and V. Dehlen, *Where Is the Proof? – A Review of Experiences from Applying MDE in Industry*, in *Model Driven Architecture – Foundations and Applications*, I. Schieferdecker and A. Hartman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 432–443.

[3] M. Knell, *The Digital Revolution and Digitalized Network Society*, *Review of Evolutionary Political Economy*, vol. 2, pp. 9–25, 2021.

[4] D. Rosenberg, B. Boehm, B. Wang, and K. Qi, *Rapid, Evolutionary, Reliable, Scalable System and Software Development: The Resilient Agile Process*, in *Proceedings of the 2017 International Conference on Software and System Process (ICSSP)*, Paris, France: ACM, 2017, pp. 60–69.

[5] L. Qian, Z. Luo, Y. Du, and L. Guo, *Cloud Computing: An Overview*, in *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 626–631.

[6] OpenAI, *GPT-4 Technical Report*, 2023. arXiv: 2303.08774.

[7] S. Noy and W. Zhang, *Experimental Evidence on the Productivity Effects of Generative Artificial Intelligence*, *Science*, vol. 381, no. 6654, pp. 187–192, 2023.

[8] J. Cabot, *Vibe Modeling: Challenges and Opportunities*, in *International Conference on Conceptual Modeling*, Springer, 2025, pp. 105–118.

[9] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, *Do Users Write More Insecure Code with AI Assistants?*, in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Copenhagen, Denmark: ACM, 2023, pp. 2785–2799.

[10] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, *Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions*, *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, Jan. 2025.

[11] J. Cámara, J. Troya, L. Burgueño, and A. Vallecillo, *On the Assessment of Generative AI in Modeling Tasks: An Experience Report with ChatGPT and UML*, *Software and Systems Modeling*, vol. 22, no. 3, pp. 781–793, Jun. 2023.

[12] E. Elshan, B. Binzer, and T. J. Winkler, *From Software Users to Software Creators: An Exploration of the Core Characteristics of the Citizen Developer Role and the Related Re- and Upskilling Programs*, *Business & Information Systems Engineering*, vol. 67, no. 1, pp. 31–53, 2025.

[13] S. Muhammad, V. Prybutok, and V. Sinha, *Unlocking Citizen Developer Potential: A Systematic Review and Model for Digital Transformation*, *Encyclopedia Journal*, vol. 5, no. 1, 2025.

[14] C. Richardson and J. R. Rymer, *New Development Platforms Emerge for Customer-Facing Applications*, *Forrester Research, Cambridge*, 2014.

[15] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, *Supporting the Understanding and Comparison of Low-Code Development Platforms*, in *Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Portorož, Slovenia: IEEE, Aug. 2020, pp. 171–178.

[16] I. Alfonso, A. Conrardy, and J. Cabot, *Towards the Interoperability of Low-Code Platforms*, in *Intelligent Information Systems*, L. Pufahl, K. Rosenthal, S. España, and S. Nurcan, Eds., Cham: Springer Nature Switzerland, 2025, pp. 3–11.

[17] A. Wąsowski and T. Berger, *Domain-Specific Languages – Effective Modeling, Automation, and Reuse*. Springer, 2023.

[18] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice, Second Edition* (Synthesis Lectures on Software Engineering). Morgan & Claypool Publishers, 2017.

[19] J. Bézivin, *On the Unification Power of Models*, *Software and Systems Modeling*, vol. 4, no. 2, pp. 171–188, May 2005.

[20] B. Selic, *The Pragmatics of Model-Driven Development*, *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003.

[21] S. Erdweg et al., *The State of the Art in Language Workbenches*, in *Software Language Engineering*, M. Erwig, R. F. Paige, and E. Van Wyk, Eds., Cham: Springer International Publishing, 2013, pp. 197–217.

[22] A. van Deursen, P. Klint, and J. Visser, *Domain-Specific Languages: An Annotated Bibliography*, *SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, Jun. 2000.

[23] S. Kelly and J. Tolvanen, *Domain-Specific Modeling – Enabling Full Code Generation*. Wiley, 2008.

[24] M. Fowler, *Domain-Specific Languages*. Pearson Education, 2010.

[25] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[26] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[27] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[28] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay, *Neo4EMF, A Scalable Persistence Layer for EMF Models*, in *Modelling Foundations and Applications*, J. Cabot and J. Rubin, Eds., Cham: Springer International Publishing, 2014, pp. 230–241.

[29] D. Kolovos et al., *A Research Roadmap towards Achieving Scalability in Model Driven Engineering*, in *BigMDE Workshop at the Software Technologies: Applications and Foundations (STAF) Conference*, Budapest, Hungary: ACM Press, 2013, pp. 1–10.

[30] M. Mernik, J. Heering, and A. M. Sloane, *When and How to Develop Domain-Specific Languages*, *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, Dec. 2005.

[31] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer, *Low-Code Development and Model-Driven Engineering: Two Sides of the Same Coin?*, *Software and Systems Modeling*, vol. 21, pp. 437–446, 2022.

[32] M. Tisi, J.-M. Mottu, D. Kolovos, J. de Lara, E. Guerra, D. Di Ruscio, A. Pierantonio, and M. Wimmer, *Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms*, in *STAF Co-Located Events*, ser. CEUR Workshop Proceedings, vol. 2405, CEUR-WS.org, 2019, pp. 73–78.

[33] F. Martínez-Lasaca, P. Díez, E. Guerra, and J. de Lara, *A Model and Workflow-Driven Approach for Engineering Domain-Specific Low-Code Platforms and Applications*, *Software and Systems Modeling*, Jul. 2025.

[34] F. Martínez-Lasaca, P. Díez, E. Guerra, and J. de Lara, *Model Sensemaking Strategies: Exploiting Meta-Model Patterns to Understand Large Models*, in *Proceedings of the 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2023, pp. 261–272.

[35] A. Garmendia, E. Guerra, D. Kolovos, and J. de Lara, *EMF Splitter: A Structured Approach to EMF Modularity*, in *Extreme Modeling (XM) Workshop at the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. CEUR Workshop Proceedings, vol. 1239, CEUR-WS.org, 2014, pp. 22–31.

[36] *UGROUND website*. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.uground.com`

[37] F. Martínez-Lasaca, P. Díez, E. Guerra, and J. de Lara, *Dandelion: A Scalable, Cloud-Based Graphical Language Workbench for Industrial Low-Code Development*, *Journal of Computer Languages*, vol. 76, 2023.

[38] F. Martínez-Lasaca, P. Díez, E. Guerra, and J. de Lara, *Engineering Low-Code Modelling Environments with Dandelion*, in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2023.

[39] F. Martínez-Lasaca, P. Díez, E. Guerra, and J. de Lara, *LowcoBot: Towards Chatting with Low-Code Platforms*, in *STAF 2024 Workshops*, ser. CEUR Workshop Proceedings, vol. 3727, CEUR-WS.org, 2024, pp. 66–76.

[40] A. Díez, *Recursive Ontology-Based Systems Engineering*, U.S. Patent 9,760,345, 2017. Accessed: Dec. 3, 2025. [Online]. Available: `https://patents.google.com/patent/US9760345B2/en`

[41] D. Kolovos, R. F. Paige, and F. Polack, *Eclipse Development Tools for Epsilon*, in *Eclipse Summit Europe*, 2006.

[42] I. Newton, *Philosophiæ Naturalis Principia Mathematica*. G. Brookman, 1833, vol. 1.

[43] S. H. Schneider and R. E. Dickinson, *Climate Modeling*, *Reviews of Geophysics*, vol. 12, no. 3, pp. 447–493, 1974.

[44] N. Bencomo, J. Cabot, M. Chechik, B. H. C. Cheng, B. Combemale, A. Wąsowski, and S. Zschaler, *Abstraction Engineering*, 2024. arXiv: `2408.14074`.

[45] J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, Eds., *Model-Driven Engineering Languages and Systems* (Lecture Notes in Computer Science). Cham: Springer International Publishing, 2014, vol. 8767.

[46] C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*. McDonnell Douglas Systems Integration Company, 1977.

[47] P. P.-S. Chen, *The Entity-Relationship Model — Toward a Unified View of Data*, *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, Mar. 1976.

[48] M. Chen, J. F. Nunamaker, and E. S. Weber, *Computer-Aided Software Engineering: Present Status and Future Directions*, *SIGMIS Database*, vol. 20, no. 1, pp. 7–13, Apr. 1989.

[49] Object Management Group, *Unified Modeling Language Specification*, Version 1.1, Object Management Group, 1997. [Online]. Available: https://www.omg.org/spec/UML/1.1

[50] D. Harel and B. Rumpe, *Meaningful Modeling: What's the Semantics of "Semantics"?*, *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[51] Object Management Group, *Model Driven Architecture*, 2003. Accessed: Dec. 3, 2025. [Online]. Available: https://www.omg.org/mda

[52] J. de Lara, *Software Development... For All? Democratisation and Automation in Software Development*, in *Extended and Selected papers of ICSOFT'25 (CCIS).* Springer, 2025, pp. 1–19.

[53] Object Management Group, *OMG Website*, 2025. Accessed: Dec. 3, 2025. [Online]. Available: https://www.omg.org

[54] Object Management Group, *Meta Object Facility (MOF) Core Specification*, 2016. Accessed: Dec. 3, 2025. [Online]. Available: https://www.omg.org/spec/MOF/2.5.1/About-MOF

[55] N. Chomsky, *Syntactic Structures*. Walter de Gruyter, 2002.

[56] I. Predoaia, D. Kolovos, and A. García-Domínguez, *Graphite: Automated Development of Hybrid Graphical-Textual DSL Editors*, in *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, York, 2025.

[57] D. L. Moody, *The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering*, *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, Nov. 2009. Accessed: Jun. 1, 2021.

[58] J. J. López-Fernández, A. G. Jorge, E. Guerra, and J. de Lara, *An Example Is Worth a Thousand Words: Creating Graphical Modelling Environments by Example*, *Software and Systems Modeling (Springer)*, vol. 18, no. 2, pp. 961–993, 2019.

[59] The Eclipse Foundation. *Xtext*, Accessed: Dec. 3, 2025. [Online]. Available: https://eclipse.dev/Xtext

[60] The Eclipse Foundation. *Sirius*, Accessed: Dec. 3, 2025. [Online]. Available: https://eclipse.dev/sirius

[61] *OutSystems*, Accessed: Dec. 3, 2025. [Online]. Available: https://www.outsystems.com

[62] J. de Lara, E. Guerra, and J. Sánchez Cuadrado, *When and How to Use Multilevel Modelling*, *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, 2014.

[63] J. de Lara and E. Guerra, *Towards the Flexible Reuse of Model Transformations: A Formal Approach Based on Graph Transformation*, *Journal of Logical and Algebraic Methods in Programming*, vol. 83, no. 5-6, pp. 427–458, 2014.

[64] C. Atkinson and T. Kühne, *The Essence of Multilevel Metamodeling*, in *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, M. Gogolla and C. Kobryn, Eds., red. by G. Goos, J. Hartmanis, and J. van Leeuwen, vol. 2185, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 19–33.

[65] C. Atkinson and T. Kühne, *Rearchitecting the UML infrastructure*, *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 12, no. 4, pp. 290–321, 2002.

[66] E. Guerra and J. de Lara, *On the Quest for Flexible Modelling*, in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ACM, 2018, pp. 23–33.

[67] A. Zolotas, R. Clarisó, N. Matragkas, D. Kolovos, and R. F. Paige, *Constraint Programming for Type Inference in Flexible Model-Driven Engineering*, *Computer Languages, Systems & Structures*, vol. 49, pp. 216–230, 2017.

[68] K. Czarnecki and S. Helsen, *Classification of Model Transformation Approaches*, in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, USA, vol. 45, 2003, pp. 1–17.

[69] J. L. Cánovas Izquierdo and J. García Molina, *Extracting Models from Source Code in Software Modernization*, *Software and Systems Modeling*, vol. 13, no. 2, pp. 713–734, 2014.

[70] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, *Model Transformation By-Example: A Survey of the First Wave*, in *Conceptual Modelling and Its Theoretical Foundations*, A. Düsterhöft, M. Klettke, and K.-D. Schewe, Eds., vol. 7260, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 197–215.

[71] T. Mens and P. Van Gorp, *A Taxonomy of Model Transformation*, *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[72] L. M. Rose, R. F. Paige, D. Kolovos, and F. Polack, *The Epsilon Generation Language*, in *Proceedings of the 4th European Conference on Model Driven Architecture (ECMDA)*, ser. Lecture Notes in Computer Science, vol. 5095, Springer, 2008, pp. 1–16.

[73] The Eclipse Foundation, *Graphiti*, 2010. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.eclipse.dev/graphiti`

[74] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, *ATL: A Model Transformation Tool*, *Science of computer programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[75] The Eclipse Foundation, *Acceleo: an open source code generator implementing the MOFM2T standard*, 2025. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.eclipse.dev/acceleo`

[76] The Eclipse Foundation, *Xtend: A Flexible and Expressive Dialect of Java*, 2025. Accessed: Dec. 3, 2025. [Online]. Available: `https://eclipse.dev/Xtext/xtend/`

[77] D. Kolovos, R. F. Paige, and F. Polack, *The Epsilon Object Language (EOL)*, in *Proceedings of the 2nd European Conference on Model Driven Architecture (ECMDA)*, ser. Lecture Notes in Computer Science, vol. 4066, Springer, 2006, pp. 128–142.

[78] D. Kolovos, R. F. Paige, and F. A. C. Polack, *The Epsilon Transformation Language*, in *Theory and Practice of Model Transformations*, A. Vallecillo, J. Gray, and A. Pierantonio, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 46–60.

[79] The Eclipse Foundation, *EVL: Epsilon Validation Language*, 2025. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.eclipse.dev/epsilon/doc/evl`

[80] Gartner, Inc. *Enterprise Low-Code Application Platforms*, Accessed: Dec. 3, 2025. [Online]. Available: `https://www.gartner.com/reviews/market/enterprise-low-code-application-platform`

[81] A. Bucaioni, C. Di Francescomarino, S. Gnesi, B. Johansson, G. Spanoudakis, and J. M. van der Werf, *Modelling in Low-Code Development: A Multi-Vocal Systematic Review*, *Software and Systems Modeling*, vol. 21, pp. 461–487, 2022.

[82] K. Rokis and M. Kirikova, *Exploring Low-Code Development: A Comprehensive Literature Review*, *Complex Systems Informatics and Modeling Quarterly*, no. 36, pp. 68–86, Oct. 2023.

[83] M. D. L. Tosi, J. L. C. Izquierdo, and J. Cabot, *A Metascience Study of the Low-Code Scientific Field*, *Journal of Object Technology*, vol. 24, no. 2, May 2025, The 21st European Conference on Modelling Foundations and Applications (ECMFA 2025).

[84] B. R. Barricelli, F. Cassano, D. Fogli, and A. Piccinno, *End-User Development, End-User Programming and End-User Software Engineering: A Systematic Mapping Study*, *Journal of Systems and Software*, vol. 149, pp. 101–137, 2019.

[85] A. C. Bock and U. Frank, *Low-Code Platform*, *Business & Information Systems Engineering*, vol. 63, no. 6, pp. 733–740, 2021.

[86] M. O. Ajimati, N. Carroll, and M. Maher, *Adoption of Low-Code and No-Code Development: A Systematic Literature Review and Future Research Agenda*, *Journal of Systems and Software*, vol. 222, 2025.

[87] B. Binzer and T. J. Winkler, *Democratizing Software Development: A Systematic Multivocal Literature Review and Research Agenda on Citizen Development*, in *Software Business*, N. Carroll, A. Nguyen-Duc, X. Wang, and V. Stray, Eds., Cham: Springer International Publishing, 2022, pp. 244–259.

[88] G. Fischer and E. Giaccardi, *Meta-design: A Framework for the Future of End-User Development*, in *End User Development*, H. Lieberman, F. Paternò, and V. Wulf, Eds. Dordrecht: Springer Netherlands, 2006, pp. 427–457.

[89] S. F. Abdul Razak, Y. Phey Ernn, F. I. Yussoff, U. Ali Bukar, and S. Yogarayan, *Enhancing Business Efficiency through Low-Code/No-Code Technology Adoption: Insights from an Extended UTAUT Model*, *Journal of Human, Earth, and Future*, vol. 5, no. 1, pp. 85–99, Mar. 2024.

[90] K. Rokis and M. Kirikova, *Challenges of Low-Code/No-Code Software Development: A Literature Review*, in *Perspectives in Business Informatics Research*, ser. Lecture Notes in Business Information Processing, vol. 462, Springer, 2022, pp. 3–17.

[91] Object Management Group, *Business Process Model and Notation*, 2008. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.bpmn.org`

[92] G. S. Guaki and G. P. Genove, *A Literature Review on Low Code and No Code (LCNC) Platforms in Reshaping Web and Application Development*, *AIP Conference Proceedings*, vol. 3287, no. 1, p. 030 021, Apr. 2025.

[93] K. Smolander, K. Lyytinen, V. Tahvanainen, and P. Marttiin, *MetaEdit – A flexible graphical environment for methodology modelling*, in *CAiSE*, ser. LNCS, vol. 498, Springer, 1991, pp. 168–193.

[94] J. Sztipanovits, G. Karsai, C. Biegl, T. Bapty, A. Ledeczi, and A. Misra, *MULTIGRAPH: An Architecture for Model-Integrated Computing*, in *Proceedings of 1st IEEE International Conference on Engineering of Complex Computer Systems*, IEEE, 1995, pp. 361–368.

[95] E. Engstrom and J. Krueger, *Building and Rapidly Evolving Domain-Specific Tools with DOME*, in *CACSD*, IEEE, 2000, pp. 83–88.

[96] J. de Lara and H. Vangheluwe, *AToM^3: A Tool for Multi-Formalism and Meta-Modelling*, in *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE)*, ser. Lecture Notes in Computer Science, vol. 2306, Springer, 2002, pp. 174–188.

[97] C. Ermel, K. Ehrig, G. Taentzer, and E. Weiss, *Object Oriented and Rule-Based Design of Visual Languages Using Tiger*, *Electronic Communications of the EASST*, vol. 1, 2006.

[98] The Eclipse Foundation, *Graphical Modeling Framework (GMF)*, 2014. Accessed: Dec. 3, 2025. [Online]. Available: `https://projects.eclipse.org/projects/modeling.gmf-tooling`

[99] D. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige, *Eugenia: Towards Disciplined and Automated Development of GMF-Based Graphical Model Editors*, *Software and Systems Modeling*, vol. 16, no. 1, pp. 229–255, 2017.

[100] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, and Á. Lédeczi, *Next Generation (Meta)Modeling: Web- and Cloud-Based Collaborative Tool Infrastructure*, in *Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. CEUR Workshop Proceedings, vol. 1237, CEUR-WS.org, 2014, pp. 41–60.

[101] J. Corley, E. Syriani, and H. Ergin, *Evaluating the Cloud Architecture of AToMPM*, in *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, SciTePress, 2016, pp. 339–346.

[102] P. Zweihoff, S. Naujokat, and B. Steffen, *Pyro: Generating domain-specific collaborative online modeling environments*, in *Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE)*, ser. Lecture Notes in Computer Science, vol. 11424, Springer, 2019, pp. 101–115.

[103] The Eclipse Foundation, *Sirius Web*. Accessed: Dec. 3, 2025. [Online]. Available: `https://eclipse.dev/sirius/sirius-web.html`

[104] The Eclipse Foundation. *EMF.cloud*, Accessed: Dec. 3, 2025. [Online]. Available: `https://www.eclipse.dev/emfcloud`

[105] A. Bucchiarone, J. Di Rocco, D. Di Vincenzo, and A. Pierantonio, *Modeling in Jjodel: Towards Bridging Complexity and Usability in Model-Driven Engineering*, *Software and Systems Modeling*, Oct. 6, 2025.

[106] Microsoft, *Language Server Protocol*, 2016. Accessed: Dec. 3, 2025. [Online]. Available: `https://microsoft.github.io/language-server-protocol`

[107] R. Rodríguez-Echeverría, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, *Towards a Language Server Protocol Infrastructure for Graphical Modeling*, in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Copenhagen, Denmark: ACM, 2018, pp. 370–380.

[108] The Eclipse Foundation, *Sprotty*, 2019. Accessed: Dec. 3, 2025. [Online]. Available: `https://sprotty.org`

[109] A. Garmendia, E. Guerra, J. de Lara, A. García-Domínguez, and D. Kolovos, *Scaling-Up Domain-Specific Modelling Languages through Modularity Services*, *Information and Software Technology*, vol. 115, pp. 97–118, 2019.

[110] K. Jahed, M. Bagherzadeh, and J. Dingel, *On the Benefits of File-Level Modularity for EMF Models*, *Software and Systems Modeling*, vol. 20, no. 1, pp. 267–286, 2021.

[111] R. Wei, D. Kolovos, A. García-Domínguez, K. Barmpis, and R. F. Paige, *Partial Loading of XMI Models*, in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ACM, 2016, pp. 329–339.

[112] Q. Ma, P. Kelsen, and C. Glodt, *A Generic Model Decomposition Technique and Its Application to the Eclipse Modeling Framework*, *Software and Systems Modeling*, vol. 14, no. 2, pp. 921–952, 2015.

[113] K. Barmpis and D. Kolovos, *Hawk: Towards a Scalable Model Indexing Architecture*, in *BigMDE Workshop at the Software Technologies: Applications and Foundations (STAF) Conference*, ACM, 2013, p. 6.

[114] G. Daniel, *Efficient Persistence and Query Techniques for Very Large Models*, in *ACM Student Research Competition at the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. CEUR Workshop Proceedings, vol. 1775, CEUR-WS.org, 2016.

[115] G. Daniel, G. Sunyé, and J. Cabot, *Advanced Prefetching and Caching of Models with PrefetchML*, *Software and Systems Modeling*, vol. 18, no. 3, pp. 1773–1794, 2019.

[116] The Eclipse Foundation, *CDO. The model repository*, 2009. Accessed: Dec. 3, 2025. [Online]. Available: `https://eclipse.dev/cdo`

[117] The Eclipse Foundation, *Teneo*, 2010. Accessed: Dec. 3, 2025. [Online]. Available: `https://wiki.eclipse.org/Teneo`

[118] J. Espinazo-Pagán, J. Sánchez Cuadrado, and J. G. Molina, *Morsa: A Scalable Approach for Persisting and Accessing Large Models*, in *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. LNCS, vol. 6981, Springer, 2011, pp. 77–92.

[119] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot, *NeoEMF: A multi-database model persistence framework for very large models*, *Science of Computer Programming*, vol. 149, pp. 9–14, 2017.

[120] M. Franzago, D. Di Ruscio, I. Malavolta, and H. Muccini, *Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map*, *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1146–1175, 2018.

[121] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, *MDEForge: An Extensible Web-Based Modeling Platform*, in *CloudMDE Workshop at the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. CEUR Workshop Proceedings, vol. 1242, 2014, pp. 66–75.

[122] F. Basciani, J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio, *Exploring Model Repositories by Means of Megamodel-Aware Search Operators*, in *MODELS Workshops*, ser. CEUR Workshop Proceedings, vol. 2245, CEUR-WS.org, 2018, pp. 793–798.

[123] Elastic NV. *Elasticsearch*, Accessed: Dec. 3, 2025. [Online]. Available: `https://www.elastic.co/elasticsearch`

[124] Apache Software Foundation, *Lucene*, 1999. Accessed: Dec. 3, 2025. [Online]. Available: `https://lucene.apache.org`

[125] B. Nuseibeh, S. M. Easterbrook, and A. Russo, *Making Inconsistency Respectable in Software Development*, *Journal of Systems and Software*, vol. 58, no. 2, pp. 171–180, 2001.

[126] N. Hili, *A Metamodeling Framework for Promoting Flexibility and Creativity over Strict Model Conformance*, in *FlexMDE Workshop at the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, vol. 1694, CEUR, 2016, pp. 2–11.

[127] J. Sottet and N. Biri, *JSMF: A JavaScript Flexible Modelling Framework*, in *FlexMDE Workshop at the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, vol. 1694, CEUR, 2016, pp. 42–51.

[128] F. R. Golra, A. Beugnard, F. Dagnat, S. Guérin, and C. Guychard, *Using Free Modeling as an Agile Method for Developing Domain Specific Modeling Languages*, in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ACM, 2016, pp. 24–34.

[129] L. Nachreiner, A. Raschke, M. Stegmaier, and M. Tichy, *CouchEdit: A Relaxed Conformance Editing Approach*, in *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, ACM, 2020.

[130] R. Jongeling, F. Ciccozzi, A. Cicchetti, and J. Carlson, *From Informal Architecture Diagrams to Flexible Blended Models*, in *ECSA*, ser. LNCS, vol. 13444, Springer, 2022, pp. 143–158.

[131] D. Wüest, N. Seyff, and M. Glinz, *FlexiSketch Team: Collaborative Sketching and Notation Creation on the Fly*, in *International Conference on Software Engineering*, vol. 2, IEEE, 2015, pp. 685–688.

[132] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara, *Example-Driven Meta-Model Development*, *Software and Systems Modeling*, vol. 14, no. 4, pp. 1323–1347, 2015.

[133] A. Zolotas, N. Matragkas, S. Devlin, D. Kolovos, and R. F. Paige, *Type Inference in Flexible Model-Driven Engineering Using Classification Algorithms*, *Software and Systems Modeling*, vol. 18, no. 1, pp. 345–366, 2019.

[134] M. Gogolla, B. Selic, A. Kästner, L. Degrandow, and C. Namegni, *From Object to Class Models: More Steps towards Flexible Modeling*, in *FPVM Workshop at the Software Technologies: Applications and Foundations (STAF) Conference*, ser. CEUR Workshop Proceedings, vol. 3250, CEUR-WS.org, 2022.

[135] T. Franz, C. Seidl, P. M. Fischer, and A. Gerndt, *Utilizing Multi-Level Concepts for Multi-Phase Modeling*, *Software and Systems Modeling*, vol. 21, no. 4, pp. 1665–1683, 2022.

[136] A. Jiménez-Pastor, A. Garmendia, and J. de Lara, *Scalable Model Exploration for Model-Driven Engineering*, *Journal of Systems and Software*, vol. 132, pp. 204–225, 2017.

[137] Y. Hu and L. Shi, *Visualizing Large Graphs*, *WIREs Computational Statistics*, vol. 7, no. 2, pp. 115–136, 2015.

[138] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J. Fekete, and D. W. Fellner, *Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges*, *Computer Graphics Forum*, vol. 30, no. 6, pp. 1719–1749, 2011.

[139] J. J. Thomas and K. A. Cook, *A Visual Analytics Agenda*, *IEEE Computer Graphics and Applications*, vol. 26, no. 1, pp. 10–13, 2006.

[140] N. Tovanich, A. Pister, G. Richer, P. Valdivia, C. Prieur, J. Fekete, and P. Isenberg, *VAST 2020 Contest Challenge: GraphMatchMaker – Visual Analytics for Graph Comparison and Matching*, *IEEE Computer Graphics and Applications*, vol. 42, no. 4, pp. 89–102, 2022.

[141] C. Vehlow, F. Beck, and D. Weiskopf, *Visualizing Group Structures in Graphs: A Survey*, *Computer Graphics Forum*, vol. 36, no. 6, pp. 201–225, 2017.

[142] F. Beck, M. Burch, S. Diehl, and D. Weiskopf, *A Taxonomy and Survey of Dynamic Graph Visualization*, *Computer Graphics Forum*, vol. 36, no. 1, pp. 133–159, 2017.

[143] R. S. Pienta, M. Kahng, Z. Lin, J. Vreeken, P. P. Talukdar, J. Abello, G. Parameswaran, and D. H. Chau, *FACETS: Adaptive Local Exploration of Large Graphs*, in *SIAM International Conference on Data Mining*. SIAM, 2017, pp. 597–605.

[144] R. Pienta, J. Abello, M. Kahng, and D. H. Chau, *Scalable Graph Exploration and Visualization: Sensemaking Challenges and Opportunities*, in *BIGCOMP*, 2015, pp. 271–278.

[145] D. M. Russell, M. J. Stefik, P. Pirolli, and S. K. Card, *The Cost Structure of Sensemaking*, in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ACM, 1993, pp. 269–276.

[146] S. Dueñas, V. Cosentino, J. M. González-Barahona, A. del Castillo San Felix, D. Izquierdo-Cortazar, L. Cañas-Díaz, and A. P. García-Plaza, *GrimoireLab: A toolset for software development analytics*, *PeerJ Computer Science*, vol. 7, e601, 2021.

[147] F. D. Luca, M. I. Hossain, S. G. Kobourov, and K. Börner, *Multi-Level Tree Based Approach for Interactive Graph Visualization with Semantic Zoom*, *EUROGRAPHICS*, vol. 39, no. 3, M. Gleicher, T. L. von Antburg, and I. Viola, Eds., 2020.

[148] B. Musial and T. Jacobs, *Application of Focus + Context to UML*, in *Australasian Symposium on Information Visualisation, InVis.au, Adelaide, Australia, 2003*, ser. CRPIT, vol. 24, Australian Computer Society, 2003, pp. 75–80.

[149] M. Frisch, R. Dachselt, and T. Brückmann, *Towards Seamless Semantic Zooming Techniques for UML Diagrams*, in *ACM Symposium on Software Visualization*, ACM, 2008, pp. 207–208.

[150] J. F. G. Navarro, R. Therón, and F. J. García-Fernández, *Semantic Zoom: A Details on Demand Visualisation Technique for Modelling OWL Ontologies*, in *Highlights in Practical Applications of Agents and Multiagent Systems – 9th International Conference on Practical Applications of Agents and Multiagent Systems, PAAMS*, ser. Advances in Intelligent and Soft Computing, vol. 89, Springer, 2011, pp. 85–92.

[151] Y. E. Ahmar, S. Gerard, C. Dumoulin, and X. L. Pallec, *Enhancing the Communication Value of UML Models with Graphical Layers*, in *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE Computer Society, 2015, pp. 64–69.

[152] C. D. Schulze, G. Hoops, and R. von Hanxleden, *Automatic Layout and Label Management for Compact UML Sequence Diagrams*, in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE Computer Society, 2018, pp. 187–191.

[153] C. D. Schulze, Y. Lasch, and R. von Hanxleden, *Label Management: Keeping Complex Diagrams Usable*, in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2016*, IEEE Computer Society, 2016, pp. 3–11.

[154] J. de Lara, E. Guerra, and J. S. Cuadrado, *Reusable abstractions for modeling languages*, *Information Systems*, vol. 38, no. 8, pp. 1128–1149, 2013.

[155] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer, *A Feature-Based Survey of Model View Approaches*, *Software and Systems Modeling*, vol. 18, no. 3, pp. 1931–1952, 2019.

[156] R. Pienta, F. Hohman, A. Endert, A. Tamersoy, K. Roundy, C. Gates, S. Navathe, and D. H. Chau, *VIGOR: Interactive Visual Exploration of Graph Query Results*, *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 215–225, Jan. 2018.

[157] R. Morgan, G. Grossmann, M. Schrefl, M. Stumptner, and T. Payne, *VizDSL: A Visual DSL for Interactive Information Visualization*, in *Advanced Information Systems Engineering*, J. Krogstie and H. A. Reijers, Eds., Cham: Springer International Publishing, 2018, pp. 440–455.

[158] G. D. Carlo, P. Langer, and D. Bork, *Rethinking Model Representation – A Taxonomy of Advanced Information Visualization in Conceptual Modeling*, in *International Conference on Conceptual Modeling (ER)*, ser. Lecture Notes in Computer Science, vol. 13607, Springer, 2022, pp. 35–51.

[159] N. Moha, V. Mahé, O. Barais, and J. Jézéquel, *Generic Model Refactorings*, in *12th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. Lecture Notes in Computer Science, vol. 5795, Springer, 2009, pp. 628–643.

[160] J. Sánchez Cuadrado, E. Guerra, and J. de Lara, *A Component Model for Model Transformations*, *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1042–1060, 2014.

[161] J. Bruel, B. Combemale, E. Guerra, J. Jézéquel, J. Kienzle, J. de Lara, G. Mussbacher, E. Syriani, and H. Vangheluwe, *Comparing and Classifying Model Transformation Reuse Approaches Across Metamodels*, *Software and Systems Modeling*, vol. 19, no. 2, pp. 441–465, 2020.

[162] C. Guy, B. Combemale, S. Derrien, J. Steel, and J. Jézéquel, *On Model Subtyping*, in *Modelling Foundations and Applications – 8th European Conference, ECMFA*, ser. Lecture Notes in Computer Science, vol. 7349, Springer, 2012, pp. 400–415.

[163] J. de Lara and E. Guerra, *From Types to Type Requirements: Genericity for Model-Driven Engineering*, *Software and Systems Modeling*, vol. 12, no. 3, pp. 453–474, 2013.

[164] H. Ergin, E. Syriani, and J. Gray, *Design Pattern Oriented Development of Model Transformations*, *Computer Languages, Systems & Structures*, vol. 46, pp. 106–139, 2016.

[165] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, and J. de Lara, *Pattern-Based Development of Domain-Specific Modelling Languages*, in *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE Computer Society, 2015, pp. 166–175.

[166] B. Hamid, S. Gürgens, and A. Fuchs, *Security Patterns Modeling and Formalization for Pattern-Based Development of Secure Software Systems*, *Innovations in Systems and Software Engineering*, vol. 12, no. 2, pp. 109–140, 2016.

[167] B. Hamid and J. Pérez, *Supporting Pattern-Based Dependability Engineering via Model-Driven Development: Approach, Tool-Support and Empirical Validation*, *Journal of Systems and Software*, vol. 122, pp. 239–273, 2016.

[168] The GEMOC Initiative. *GEMOC Studio*, Accessed: Dec. 3, 2025. [Online]. Available: `https://gemoc.org/studio.html`

[169] Siemens Industry Software. *Mendix*, Accessed: Dec. 3, 2025. [Online]. Available: `https://www.mendix.com`

[170] Zoho Corporation Pvt. Ltd. *Zoho Creator*, Accessed: Dec. 3, 2025. [Online]. Available: `https://www.zoho.com/creator`

[171] Microsoft, *Microsoft PowerApps*. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.microsoft.com/power-platform/products/power-apps`

[172] Google, *Google AppSheet*. Accessed: Dec. 3, 2025. [Online]. Available: `https://about.appsheet.com/home`

[173] *Kissflow*, Accessed: Dec. 3, 2025. [Online]. Available: `https://kissflow.com`

[174] Salesforce, Inc. *Salesforce Platform*, Accessed: Dec. 3, 2025. [Online]. Available: `https://www.salesforce.com/platform`

[175] *Appian*, Accessed: Dec. 3, 2025. [Online]. Available: `https://appian.com`

[176] Caspio, Inc., *Caspio*. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.caspio.com`

[177] M. Francis, D. Kolovos, N. Matragkas, and R. F. Paige, *Adding Spreadsheets to the MDE Toolkit*, in *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS)*, ser. Lecture Notes in Computer Science, vol. 8107, Springer, 2013, pp. 35–51.

[178] A. Díez and J. de Lara, *Semantic Digital Twins for Organizational Development*, in *Second International Workshop on Semantic Digital Twins*, vol. 2887, CEUR, 2021, pp. 1–12.

[179] I. Alfonso, A. Conrardy, A. Sulejmani, A. Nirumand, F. Ul Haq, M. Gomez-Vazquez, J.-S. Sottet, and J. Cabot, *Building BESSER: An Open-Source Low-Code Platform*, in *Proceedings of the 25th International Conference on Enterprise, Business-Process and Information Systems Modeling (EMMSAD)*, ser. Lecture Notes in Business Information Processing, vol. 511, Springer, 2024, pp. 203–212.

[180] E. Chavarriaga, F. Jurado, and F. D. Rodríguez, *An Approach to Build JSON-Based Domain Specific Languages Solutions for Web Applications*, *Journal of Computer Languages*, vol. 75, 2023.

[181] E. Chavarriaga, L. Rojas, F. D. Rodríguez, K. Sorbello, and F. Jurado, *RestRho: A JSON-Based Domain-Specific Language for Designing and Developing RESTful APIs to Validate RhoArchitecture*, *Future Internet*, vol. 17, no. 8, 2025.

[182] W. M. P. van der Aalst and K. M. van Hee, *Workflow Management: Models, Methods, and Systems*. Cambridge, MA, USA: MIT Press, 2004.

[183] Object Management Group, *Software & Systems Process Engineering Metamodel (SPEM)*, 2008. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.omg.org/spec/SPEM/2.0/About-SPEM`

[184] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede, *Workflow Patterns: The Definitive Guide*. MIT Press, 2016.

[185] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edition. Addison-Wesley Professional, 1994.

[186] J. Rivera, D. Ruiz-González, F. López-Romero, J. Bautista, and A. Vallecillo, *Orchestrating ATL Model Transformations*, in *Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL)*, 2009.

[187] B. Sánchez, D. Kolovos, and R. Paige, *ModelFlow: Towards Reactive Model Management Workflows*, in *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, Athens, Greece: ACM, 2019, pp. 30–39.

[188] B. Sánchez, D. Kolovos, and R. Paige, *To build, or not to build: ModelFlow, a build solution for MDE projects*, in *Proceedings of the 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Virtual Event, Canada: ACM, 2020, pp. 1–11.

[189] M. Gamboa and E. Syriani, *Improving User Productivity in Modeling Tools by Explicitly Modeling Workflows*, *Software and Systems Modeling*, vol. 18, no. 4, pp. 2441–2463, Aug. 2019.

[190] The Eclipse Foundation, *Modeling Workflow Engine*. Accessed: Dec. 3, 2025. [Online]. Available: `https://projects.eclipse.org/projects/modeling.mwe`

[191] S. Kokaly, *Towards a Structured Workflow Language for Model Management*, in *Proceedings of Doctoral Symposium co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. CEUR Workshop Proceedings, vol. 1321, CEUR-WS.org, 2014.

[192] A. Indamutsa, J. Di Rocco, D. Di Ruscio, and A. Pierantonio, *MDEForgeWL: Towards Cloud-Based Discovery and Composition of Model Management Services*, in *Proceedings of the 24th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, IEEE, 2021, pp. 118–127.

[193] A. Indamutsa, J. D. Rocco, L. Almonte, D. D. Ruscio, and A. Pierantonio, *Advanced Discovery Mechanisms in Model Repositories*, *Software: Practice and Experience*, vol. 54, no. 11, pp. 2214–2248, 2024.

[194] S. Pérez-Soler, E. Guerra, J. de Lara, and F. Jurado, *The Rise of the (Modelling) Bots: Towards Assisted Modelling via Social Networks*, in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 723–728.

[195] M. B. Chaaben, L. Burgueño, and H. A. Sahraoui, *Towards using Few-Shot Prompt Learning for Automating Model Completion*, in *Proceedings of the New Ideas and Emerging Results (NIER) Track at the International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 7–12.

[196] K. Chen, Y. Yang, B. Chen, J. A. H. López, G. Mussbacher, and D. Varró, *Automated Domain Modeling with Large Language Models: A Comparative Study*, in *Proceedings of the 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2023, pp. 162–172.

[197] L. Almonte, E. Guerra, I. Cantador, and J. de Lara, *Building Recommenders for Modelling Languages with Droid*, in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2022.

[198] S. Pérez-Soler, S. Juarez-Puerta, E. Guerra, and J. de Lara, *Choosing a Chatbot Development Tool*, *IEEE Software*, vol. 38, no. 4, pp. 94–103, 2021.

[199] S. Pérez-Soler, E. Guerra, and J. de Lara, *Model-Driven Chatbot Development*, in *International Conference on Conceptual Modeling*, ser. LNCS, vol. 12400, Springer, 2020, pp. 207–222.

[200] B. Estrada-Torres, A. del-Río-Ortega, and M. Resinas, *DemaBot: a Tool to Automatically Generate Decision-Support Chatbots*, in *Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track at BPM 2021 co-located with the 19th International Conference on Business Process Management (BPM)*, ser. CEUR Workshop Proceedings, vol. 2973, Rome, Italy: CEUR-WS.org, 2021, pp. 141–145.

[201] N. Rao, J. Tsay, K. Kate, V. Hellendoorn, and M. Hirzel, *AI for Low-Code for AI*, in *Proceedings of the 29th International Conference on Intelligent User Interfaces (IUI)*, New York, NY, USA: ACM, 2024, pp. 837–852.

[202] P. Chittò, M. Baez, F. Daniel, and B. Benatallah, *Automatic Generation of Chatbots for Conversational Web Browsing*, in *Proceedings of the 29th International Conference on Conceptual Modeling (ER)*, Springer, 2020, pp. 239–249.

[203] I. Weber, *Low-Code from Frontend to Backend: Connecting Conversational User Interfaces to Backend Services via a Low-Code IoT Platform*, in *Proceedings of the 3rd Conference on Conversational User Interfaces (CUI)*, Bilbao, Spain: ACM, 2021.

[204] A. Díez, N. Nguyen, F. Díez, and E. Chavarriaga, *MDE for Enterprise Application Systems*, in *1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, SciTePress, 2013, pp. 253–256.

[205] The World Wide Web Consortium, *Web Ontology Language (OWL)*, 2004. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.w3.org/OWL`

[206] The World Wide Web Consortium, *Resource Description Framework (RDF)*, 2014. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.w3.org/RDF`

[207] M. S. Ángel, J. de Lara, P. Neubauer, and M. Wimmer, *Automated Modelling Assistance by Integrating Heterogeneous Information Sources*, *Computer Languages, Systems & Structures*, vol. 53, pp. 90–120, 2018.

[208] A. Lange and C. Atkinson, *Multi-Level Modeling with LML: A Contribution to the Multi-level Process Challenge*, *Enterprise Modelling and Information Systems Architectures*, vol. 17, pp. 1–36, 2022.

[209] OCL, 2014. Accessed: Dec. 3, 2025. [Online]. Available: `https://www.omg.org/spec/OCL`

[210] The vis.js community, *Vis.js*, 2019. Accessed: Dec. 3, 2025. [Online]. Available: `https://visjs.org`

[211] Facebook Open Source, *React*, 2013. Accessed: Dec. 3, 2025. [Online]. Available: `https://react.dev`

[212] Node.js Foundation, *Node.js*, 2009. Accessed: Dec. 3, 2025. [Online]. Available: `https://nodejs.org`

[213] Microsoft, *TypeScript*, 2012. Accessed: Dec. 3, 2025. [Online]. Available: https://www.typescriptlang.org

[214] The Eclipse Foundation, *Eclipse Layout Kernel (ELK)*, 2018. Accessed: Dec. 3, 2025. [Online]. Available: https://www.eclipse.dev/elk

[215] *Socket.IO*, 2012. Accessed: Dec. 3, 2025. [Online]. Available: https://socket.io

[216] IEEE, *IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams*, *IEEE Std 1849-2016*, pp. 1–50, 2016.

[217] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, *Automating Co-Evolution in Model-Driven Engineering*, in *EDOC*, IEEE, 2008, pp. 222–231.

[218] J. Bertin, *Semiology of Graphics*. University of Wisconsin Press, 1983.

[219] D. P. Gregor, J. Järvi, J. G. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, *Concepts: Linguistic Support for Generic Programming in C++*, in *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2006, pp. 291–310.

[220] J. Sánchez Cuadrado, E. Guerra, and J. de Lara, *Flexible Model-to-Model Transformation Templates: An Application to ATL*, *Journal of Object Technology*, vol. 11, no. 2, 2012.

[221] S. MacLane, *Categories for the Working Mathematician*. New York: Springer-Verlag, 1971.

[222] C. Atkinson and T. Kühne, *Model-Driven Development: a Metamodeling Foundation*, *IEEE Software*, vol. 20, no. 5, pp. 36–41, 2003.

[223] J. R. Ullmann, *An Algorithm for Subgraph Isomorphism*, *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[224] *Data Driven Documents (D3.js)*, 2011. Accessed: Dec. 3, 2025. [Online]. Available: https://d3js.org

[225] The Apache Software Foundation, *Apache ECharts*, 2017. Accessed: Dec. 3, 2025. [Online]. Available: https://echarts.apache.org/en/index.html

[226] R. Salado-Cid, A. Vallecillo, K. Munir, and J. R. Romero, *SWEL: A Domain-Specific Language for Modeling Data-Intensive Workflows*, *Business & Information Systems Engineering*, vol. 66, no. 2, pp. 137–160, 2024.

[227] Moodle Community, *Moodle*, 2002. Accessed: Dec. 3, 2025. [Online]. Available: https://moodle.org

[228] Instructure Inc., *Canvas LMS*, 2011. Accessed: Dec. 3, 2025. [Online]. Available: https://www.instructure.com/es/canvas

[229] N. N. Mohd Kasim and F. Khalid, *Choosing the Right Learning Management System (LMS) for the Higher Education Institution Context: A Systematic Review*, *International Journal of Emerging Technologies in Learning*, vol. 11, no. 6, pp. 55–61, Jun. 2016.

[230] Ç. Cirit and F. Buzluca, *A UML Profile for Role-Based Access Control*, in *Proceedings of the 2nd International Conference on Security of Information and Networks (SIN)*, ACM, 2009, pp. 83–92.

[231] F. Bancilhon and N. Spyratos, *Update Semantics of Relational Views*, *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 557–575, 1981.

[232] R. F. Paige, D. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack, *The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering*, in *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE Computer Society, 2009, pp. 162–171.

[233] A. de la Vega, P. Sánchez, and D. Kolovos, *Pinset: A DSL for Extracting Datasets from Models for Data Mining-Based Quality Analysis*, in *Proceedings of the 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, IEEE, 2018, pp. 83–91.

[234] D. Kolovos and A. de la Vega, *Flexmi: A Generic and Modular Textual Syntax for Domain-Specific Modelling*, *Software and Systems Modeling*, vol. 22, no. 4, pp. 1197–1215, 2023.

[235] A. B. Kahn, *Topological Sorting of Large Networks*, *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[236] F. Brooks, *No Silver Bullet – Essence and Accident in Software Engineering*, *IEEE Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[237] S. Pérez-Soler, E. Guerra, and J. de Lara, *Flexible Modelling Using Conversational Agents*, in *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2019, pp. 478–482.

[238] R. Ren, J. W. Castro, S. T. Acuña, and J. de Lara, *Usability of Chatbots: A Systematic Mapping Study*, in *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019*, A. Perkusich, Ed., KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019, pp. 479–484.

[239] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention Is All You Need*, *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[240] W. X. Zhao et al., *A Survey of Large Language Models*, 2023. arXiv: 2303.18223.

[241] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, *ReAct: Synergizing Reasoning and Acting in Language Models*, 2023. arXiv: 2210.03629. Accessed: Dec. 3, 2025.

[242] Google LLC, *Angular*, 2016. Accessed: Dec. 3, 2025. [Online]. Available: https://angular.dev

[243] LangChain Inc., *LangChain*, 2022. Accessed: Dec. 3, 2025. [Online]. Available: https://www.langchain.com

[244] W. M. P. van der Aalst, *Process Mining: A 360 Degree Overview*, in *Process Mining Handbook*, ser. Lecture Notes in Business Information Processing (LNBIP), vol. 448, Springer, 2022, pp. 3–34.

[245] J. Bézivin, *Model Driven Engineering: An Emerging Technical Space*, in *GTTSE*, ser. LNCS, vol. 4143, Springer, 2005, pp. 36–64.

[246] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[247] The Eclipse Foundation, *Eclipse Theia*, 2017. Accessed: Dec. 3, 2025. [Online]. Available: https://theia-ide.org

[248] A. Yohannis, R. Hoyos-Rodríguez, F. Polack, and D. Kolovos, *Towards Efficient Comparison of Change-Based Models*, *Journal of Object Technology*, vol. 18, no. 2, Jul. 2019.

[249] L. M. Rose, D. Kolovos, R. F. Paige, F. A. Polack, and S. Poulding, *Epsilon Flock: A Model Migration Language*, *Software and Systems Modeling*, vol. 13, no. 2, pp. 735–755, May 2014.

[250] A. Yohannis, D. Kolovos, A. García-Domínguez, and C. J. Fernández Candel, *Picto Web: A Tool for Complex Model Exploration*, in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Montreal, Quebec, Canada: ACM, 2022, pp. 56–60.

[251] H. Ed-douibi, J. L. Cánovas Izquierdo, F. Bordeleau, and J. Cabot, *WAPIml: Towards a Modeling Infrastructure for Web APIs*, in *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2019, pp. 748–752.

Part VI

Appendices

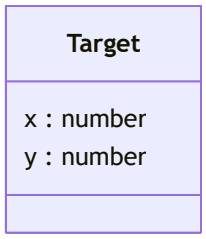
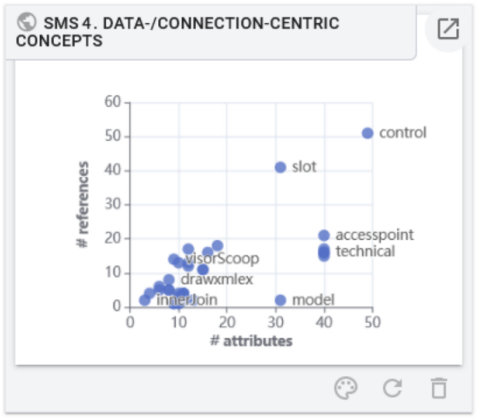
Appendix A

Catalog of Model Sensemaking Strategies

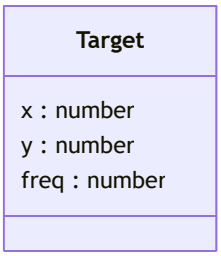
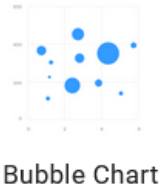
This appendix presents the complete catalog of model sensemaking strategies (SMSs) introduced in Chapter 5. Each strategy is described following a common template, which includes its intent, presentation metaphor, context meta-model, visualization variants, properties, and motivating examples.

The catalog presents the following SMSs: Numerical (A.1), Numerical with frequency (A.2), Categorical (A.3), Metric distribution (A.4), Free metric (A.5), Bounded metric (A.6), Literal metric (A.7), Time-based (A.8), Connectivity (A.9), and Weighted hierarchy (A.10).

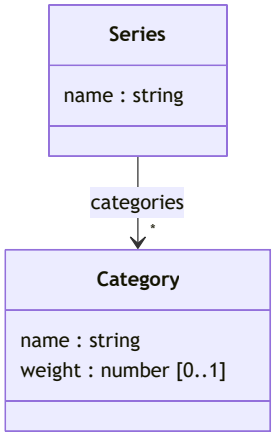
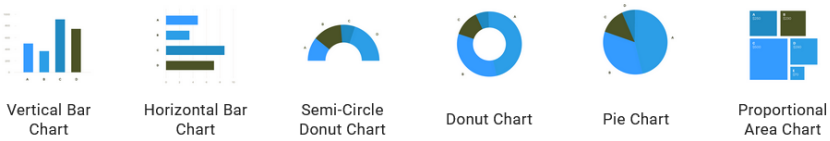
A.1 Numerical

Intent	• *What is the relationship between two variables?* • *How does a variation in one variable affect the other?*
Presentation metaphor	Data, together with *Numerical with frequency*.
Context meta-model	The context meta-model of *Numerical* contains a single class, Target, with two numeric attributes x and y. 
Visualization variants	• *Line Graph* and *Area Chart*, ideal for data ordered by the X axis; • *Scatter Plot*, for independent X and Y values. 
Properties	• title : string [1..1] – title of the visualization. • X_label : string [0..1] – label for the X axis. • Y_label : string [0..1] – label for the Y axis.
Motivating example	The strategy can be used to understand the "size" of meta-model concepts in a single chart. In particular, it associates the X and Y coordinates to the number of attributes and relationships of a concept, respectively. The mapping is domain-agnostic, so it can be reused for any meta-model: • Target ↦ SemanticNode • x ↦ / this.getDataProperties().length • y ↦ / this.getObjectProperties().length 

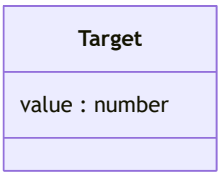
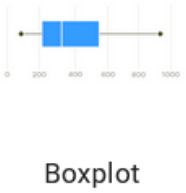
A.2 Numerical with frequency

Intent	The *Numerical with frequency* strategy allows visualizing triplets as coordinates (x, y) with an associated numeric frequency, aiming to answer: • *What is the relationship between three numeric variables?*
Presentation metaphor	Data, together with *Numerical*.
Context meta-model	The context meta-model of *Numerical with frequency* contains a single class, Target, with three numeric attributes: x, y, and freq. 
Visualization variants	• *Bubble Chart* displays coordinates as bubbles with a size proportional to the frequency freq. 
Properties	• title : string [1..1] – title of the visualization. • X_label : string [0..1] – label for the X axis. • Y_label : string [0..1] – label for the Y axis.
Motivating examples	The strategy can be applied to augment scatter plots with a third numeric dimension. For instance, a scatter plot that compares countries by GDP (X axis) and life expectancy (Y axis) can scale the size of the points according to the population (frequency) of each country. Another example comes from the world of finances: a bubble chart can visualize stocks by plotting volatility on the X axis and expected return on the Y axis, with the size of each bubble representing the investment amount in that stock.

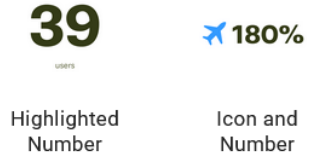
A.3 Categorical

Intent	The *Categorical* SMS exploits aggregation to partition objects into categories and understand their incidence. It aims to answer: • *What categories emerge from the data?* • *What is the incidence of each category?* (including discovering the most and least frequent categories).
Presentation metaphor	This is a grouping-based SMS. Other examples under the same metaphor include cluster analysis and pattern matching.
Context meta-model	The context meta-model of *Categorical* contains two classes, Series and Category, both identified by name. Each series is associated with a set of categories, refining the scope of the analysis. Categories expose an optional numeric weight, which determines the contribution of the bound object to its category tally; this weight is assumed to be 1 if left unspecified. Objects sharing a Category.name are aggregated into the same category, and the visualization is replicated for each series. 
Visualization variants	• *Vertical and horizontal bar charts*; • *Semi-circle donut chart*; • *Donut chart*; • *Pie chart*; • *Proportional area chart* 
Properties	• title : string [1..1] – title of the visualization. • X_label : string [0..1] – label for the X axis. • Y_label : string [0..1] – label for the Y axis. • category_names : string [0..*] – forces the categories to be displayed, and their order. Categories with a 0 value are still displayed. If empty, only categories with a non-zero value are displayed. • percent : boolean [0..1] – if true, the values are displayed as percentages of the total.
Motivating example	The *Categorical* SMS can be applied, for instance, to a meta-model of conferences and papers.

A.4 Metric distribution

Intent	The *Metric distribution* strategy aims to answer the following sensemaking task: • *What is the distribution of a metric?*
Presentation metaphor	Metric.
Context meta-model	The context meta-model of *Metric distribution* contains a single class, Target, with a numeric attribute value. 
Visualization variants	• *Boxplot*. It displays five summary statistics: the minimum (min), the first quartile (Q1), the median, the third quartile (Q3), and the maximum (max). The visualization additionally reports the mean of the data. 
Properties	• title : string [1..1] – title of the visualization.

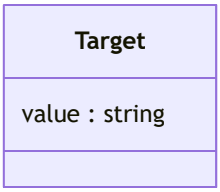
A.5 Free metric

Intent	The *Free metric* strategy is intended to display any relevant metric as plain text.
Presentation metaphor	Metric.
Context meta-model	This SMS does not have a context meta-model and, therefore, can be applied to any (meta-)model.
Visualization variants	• *Simple highlighted number* • *Icon and number* 
Properties	• title : string [1..1] – title of the visualization. • value : any [1..1] – the value to be displayed. • subtitle : string [0..1].

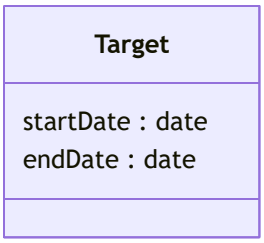
A.6 Bounded metric

Intent	The *Bounded metric* strategy displays a numeric value within a range. It is populated by the value, minValue, and maxValue properties (which can be fixed or derived).
Presentation metaphor	Metric.
Context meta-model	This SMS does not have a context meta-model and, therefore, can be applied to any (meta-)model.
Visualization variants	• *Angular gauge* • *Gauge chart*  Angular Gauge  Gauge Chart
Properties	• title : string [1..1] – title of the visualization. • value : number [1..1] – the value to be displayed. • minValue : number [1..1] – the minimum value of the range. • maxValue : number [1..1] – the maximum value of the range.

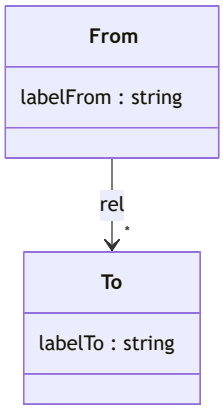
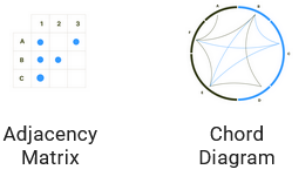
A.7 Literal metric

Intent	The *Literal metric* strategy depicts textual fields and intends to answer: • *What are the most frequent terms?*
Presentation metaphor	Metric.
Context meta-model	The context meta-model contains a single class, Target, with a textual attribute value. 
Visualization variants	• *Word cloud*  Word cloud
Properties	• title : string [1..1] – title of the visualization.
Motivating examples	Detecting the most common tags in a set of articles, or visualizing the most frequent categories of logs in a system.

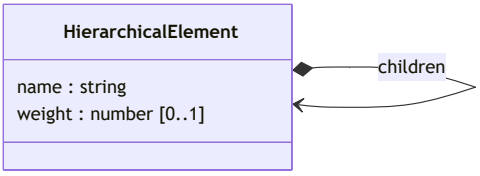
A.8 Time-based

Intent	The *Time-based* strategy aims to answer the following sensemaking task: • *What are the time intervals of a series of tasks?*
Presentation metaphor	Time-based.
Context meta-model	The context meta-model contains a single class, Target, with two date attributes startDate and endDate. 
Visualization variants	• *Gantt chart* 
Properties	• title : string [1..1] – title of the visualization.
Motivating examples	The *Time-based* SMS can be applied to numerous scenarios. The most common scenario is visualizing the timeline of a project. It can also be exploited to avoid overloading resources when scheduling tasks. Finally, it can be applied to event planning to prevent bottlenecks and scheduling conflicts.

A.9 Connectivity

Intent	The *Connectivity* strategy displays a relation over two types of objects and aims to answer: • *What are the relationships between two types of objects?*
Presentation metaphor	Structural.
Context meta-model	The context meta-model of *Connectivity* contains two classes, From and To, with associated labels labelFrom and labelTo, respectively. They are connected via a relation from From to To. 
Visualization variants	References can be visualized: • in a table through an *Adjacency Matrix* • as a *Chord Diagram* 
Properties	• title : string [1..1] – title of the visualization.
Motivating example	An example application is the analysis of dependencies in an architecture meta-model. From elements may represent components, To elements represent services or data stores, and the relation captures invocations or access links. The adjacency matrix or chord diagram reveals highly coupled components, isolated elements, and asymmetric dependencies, helping architects to identify refactoring opportunities and potential single points of failure.

A.10 Weighted hierarchy

Intent	The *Weighted Hierarchy* SMS aims to understand (multi-level) hierarchies emerging from containment relationships, and the relevance of their elements. It addresses: • *What is the structure of the examined component?* • *What is the relevance of the components?*
Presentation metaphor	This is a structural-based SMS. The other SMS that belongs to this group is *Connectivity*, which relaxes the composition to an association.
Context meta-model	The strategy fuses the concepts of containers and containees into a single class, HierarchicalElement. These have a name and an optional numeric weight which, by default, is the size of the children collection. Hierarchical elements are related to themselves via a children composition, supporting recursive containment relationships. 
Visualization variants	• The SMS can be visualized with *Treemap*s spanning multiple levels.
Properties	• title : string [1..1] – title of the visualization.
Motivating example	A common use case is the inspection of a modular software system, where HierarchicalElement represents packages, modules, and classes organized by containment. The weight attribute can be bound to a metric such as lines of code, cyclomatic complexity, or defect count. The resulting treemap emphasizes which subtrees concentrate most of the size or problems, supporting the prioritization of testing, maintenance, and refactoring activities.

Appendix B

Addenda to PLATFLOW

This appendix complements Chapter 6 with the OCL constraints of PLATFLOW's workflow package from Figure 6.11.

First, input and output port names within a node must be unique:

```
context Node
inv: inputPorts->isUnique(name)

context Node
inv: outputPorts->isUnique(name)
```

Moreover, input parameters of a workflow must also be unique:

```
context Workflow
inv: parameters->isUnique(name)
```

Second, workflow nodes admit an unlimited number of input ports by default (Node.inputPorts). However, some node types constrain their cardinality:

```
context InputNode
inv: inputPorts->isEmpty()

context EntityRetriever
inv: inputPorts->isEmpty()

context Text
inv: inputPorts->isEmpty()

context Evl
inv: inputPorts->size() = 1

context Pinset
inv: inputPorts->size() = 1
```

```
context Parser
inv: inputPorts->size() = 1

context Conditional
inv: inputPorts->size() = 1

context SensemakingStrategy
inv: inputPorts->size() = 1
```

Third, all node types have exactly one output port except Conditional, which has two of them – one per branch:

```
context Node
inv: not self.oclIsTypeOf(Conditional)
     implies outputPorts->size() = 1

context Conditional
inv: outputPorts->size() = 2
```

Specifically, both output ports of the Conditional node must be called "ok" and "ko" to correctly identify the execution branches:

```
context Conditional
inv: outputPorts->collect(name)->asSet()
     = Set{'ok', 'ko'}
```

Fourth, nodes that call other workflows cannot call the workflow they are part of (recursion is not allowed):

```
context Workflow
inv: nodes->forAll(n |
    n.oclIsKindOf(WfConsumer) implies
    n.references <> self
)
```

Finally, in WorkflowCalls, their input ports must match with the input parameters of the called workflow by name:

```
context WorkflowCall
inv:
let inputNames = references.parameters
    ->collect(name) in
let portNames = inputPorts
    ->collect(name) in
inputNames->asSet() = portNames->asSet()
```