Software and Systems Modeling
https://doi.org/10.1007/s10270-025-01324-y

TOOL PAPER

Modeling in Jjodel: towards bridging complexity and usability in model-driven engineering

Antonio Bucchiarone[1] · Juri Di Rocco[1] · Damiano Di Vincenzo[1] · Alfonso Pierantonio[1]

Received: 13 January 2025 / Revised: 19 August 2025 / Accepted: 9 September 2025

Abstract
Jjodel is a cloud-based reflective platform designed to address the challenges of Model-Driven Engineering (MDE), particularly the cognitive complexity and usability barriers often encountered in existing model-driven tools. This article presents the motivation and requirements behind the design of Jjodel and demonstrates how it satisfies these through its key features. By offering a low-code environment with modular viewpoints for syntax, validation, and semantics, Jjodel empowers language designers to define and refine domain-specific languages (DSLs) with ease. Its innovative capabilities, such as live co-evolution support, and syntax customization, ensure adaptability for academic and industrial contexts. A practical case study of an algebraic expression language highlights the ability of Jjodel to manage positional semantics and event-driven workflows, illustrating its effectiveness in simplifying complex modeling scenarios. Built on modern front-end technologies, Jjodel aims to operationalize concepts from MDE research into a usable platform that supports a range of modeling tasks.

Keywords Model-driven engineering (MDE) · Reflective platforms · Low-code development · Jjodel · Co-evolution · Language workbench · Graphical model editing

1 Introduction

The growing complexity of software systems demands tools that allow for efficient and intuitive modeling [1]. MDE has emerged as a powerful paradigm to facilitate the transition from abstract models to concrete implementations [2, 5]. However, many existing modeling tools inadvertently add complexity [6], especially in educational settings where the primary objective is to simplify learning and make modeling more accessible to students and newcomers [26]. In MDE, managing change and uncertainty is essential for supporting real-world development processes. *Co-evolution* refers to automated or semi-automated synchronization between evolving metamodels and their corresponding models, ensuring consistency across abstraction levels while minimizing manual intervention [19, 36]. Complementing this, *flexibility* in modeling empowers users to explore solutions incrementally, allowing them to define structures without committing to rigid typing early, adapt metamodels on-the-fly, and refine constructs as understanding evolves [4, 10, 20]. Together, these capabilities support a more adaptive and resilient modeling process, enabling tools to accommodate evolving requirements and design decisions without compromising model integrity or user intent. Modeling tools play a critical role in software development, offering designers and engineers a means to explore design spaces and communicate complex systems effectively to stakeholders [24]. Despite their importance, academic tools often fail to meet expectations, lacking maturity, robustness, and usability. Students are frequently confronted with issues such as cumbersome installation, configuration, and maintenance processes, which shift their focus from understanding modeling concepts to overcoming technical obstacles. In contrast, while industrial tools tend to be more polished and feature rich, they are often prohibitively expensive, difficult to learn, and misaligned with the specific needs of educational settings [24].

Communicated by Abel Gómez and Jordi Cabot.

✉ Antonio Bucchiarone
antonio.bucchiarone@univaq.it

Juri Di Rocco
juri.dirocco@univaq.it

Damiano Di Vincenzo
damiano.divincenzo@student.univaq.it

Alfonso Pierantonio
alfonso.pierantonio@univaq.it

[1] SWEN, Università degli Studi dell'Aquila, 67100 L'Aquila, Italy

Published online: 06 October 2025

These challenges highlight the need for tools that simplify the modeling process without compromising capability. An ideal tool for education should be intuitive, easy to use and platform independent, while supporting various paradigms and DSLs [24]. Furthermore, modern tools must showcase the practical benefits of MDE by allowing students to create models that drive downstream development activities, rather than serving as static diagrams [34].

This article introduces Jjodel [13, 14][1], a cloud-based reflective platform designed to address these challenges. The name blends technical and cultural cues: the suffix *-odel* alludes to "models," consistent with the focus of the platform on model-driven engineering; one "J" references JavaScript, one of the main implementation language; the other is a playful nod to *jasant*, a colloquial Eastern Abruzzese interjection meaning "damn!" to recall the perils of designing a complex tool like Jjodel.

Through a practical example, we demonstrate how Jjodel streamlines the modeling process by offering a low-code environment that balances simplicity and advanced functionality, thus enabling diverse modeling tasks while prioritizing accessibility and usability. Building on the principle of transparency in tools, understood as the ability of a tool to recede from the user's attention and allow focus to remain on the task at hand, Jjodel aims to eliminate technical barriers that often interrupt the modeling process. This notion, originally articulated in the philosophy of technology by Heidegger under the concept of ready-to-hand [18], has since been adopted and expanded within the fields of human–computer interaction and cognitive psychology [30]. When tools are transparent in this sense, they become cognitively assimilated by users, enabling fluid and unimpeded interaction. Jjodel embodies this idea by prioritizing responsiveness and ease of use, integrating features such as syntax viewpoints and real-time validation to support seamless modeling experiences. A more comprehensive discussion of these principles as applied to model-driven engineering is presented in [31].

Unlike many academic tools, Jjodel removes the challenges of installation and configuration, allowing modelers to focus on modeling concepts and their practical applications. Its features, automated feedback, and support for multiple modeling paradigms bridge the gap between the simplicity required for modeling and the robustness demanded by software development. Jjodel aims to improve learning by offering an intuitive modeling environment aligned with the practical goals of MDE.

[1] The Jjodel tool available online at https://www.jjodel.io. The source code can be accessed on GitHub at https://github.com/MDEGroup/jjodel. Furthermore, a collection of video tutorials demonstrating various features and use cases of Jjodel is available at https://www.jjodel.io/video-tutorials/.

In this article, we present how Jjodel transforms the language engineering process. Its innovative approach simplifies the learning curve, offering students an opportunity to focus on mastering MDE principles and applying them effectively in diverse scenarios. Through its integration of accessibility, functionality, and educational support, Jjodel emerges as a compelling solution to the challenges of teaching modeling in modern educational environments.

1.1 Objectives and structure of the article

This article introduces Jjodel, a cloud-based platform that tackles the challenges of MDE by combining accessibility, advanced features, and educational support. By simplifying complexity and enabling automated co-evolution and modular syntax viewpoints, Jjodel connects more theoretical research with practical MDE applications.

To demonstrate the flexibility and expressiveness of Jjodel, we exploit two complementary, yet contrasting, example modeling languages. The first, Entity-Relationship Diagram (ERD), is a widely familiar and structurally rich visual notation. Its popularity and simplicity make it ideal for introducing the core capabilities of Jjodel's functions, such as visual syntax specification, metamodel conformance, and diagrammatic interaction, without introducing domain-specific complexity that might distract the reader. Despite its apparent simplicity, the ERD still benefits from features that are often essential in DSL design, including custom metamodeling, entity typing, and association constraints. This makes it a minimal but meaningful scenario that serves as an effective entry point to explore Jjodel. Our intention is to use a notation that would allow a smooth learning curve, gradually transitioning to more sophisticated and DSL-relevant examples. The second example, an expression-based language, illustrates Jjodel's ability to handle interactive and layout-dependent semantics. Unlike ERD, it emphasizes spatial arrangement and model execution, offering a dynamic and executable modeling experience. Together, these two scenarios show the breadth of modeling styles supported by Jjodel from classical diagrammatic modeling to advanced spatial and executable DSLs.

The article opens with an in-depth overview of Jjodel's architecture and key features (Sect. 2), detailing its front and back-end components, the object model and the dynamic viewpoints that collectively improve the modeling process. To demonstrate its practical utility, a case study of an algebraic expression language is presented (Sect. 4), illustrating Jjodel's capabilities in managing abstract and concrete syntax, validation, and event-driven workflows. The article reflects on the lessons learned during the development of Jjodel, emphasizing its broader implications for the MDE community (Sect. 5). Sect. 6 reviews related tools and highlights how Jjodel addresses specific challenges in MDE.

Conclusions outline future directions (Sect. 7), with a focus on strategies to improve scalability, integration, and community participation. Furthermore, it highlights the opportunity to showcase the potential of a demonstrator, illustrating Jjodel as a flexible tool to address a range of MDE challenges.

2 Overview of Jjodel

This section provides a concise overview of the architecture and core concepts of Jjodel that are used in the case studies. Readers interested only in concepts can skip Sects. 2.1 and 2.2 and proceed to Sect. 2.3. We then outline the architecture and core components, its overall structure (Sect. 2.1), front- and back-end services, the Jjodel Object Model (JOM), the syntax and validation viewpoints, and the editing mechanisms, showing how Jjodel enhances MDE workflows with flexibility, unobtrusive co-evolution, and robust and efficient functionality.

We illustrate these features using an example of an entity relationship diagram (ERD). In particular:

> Figure 1a shows the ERD metamodel (classes and enumerations). An `Entity` has `ownedAttributes`; each `Attribute` has a `type` and an `isPK` flag (primary key). A `Relation` links two entities via `left`/`right` references and is constrained by a `Multiplicity`. The `1` beside `Int`, `String` and `Boolean` marks singleton classes: auto-created once, not duplicated or deletable.
> Figure 1b shows a conforming ERD instance rendered as a concrete diagrammatic syntax: `User` (`id`, `surname`, `firstname`) and `Role` (`id`, `name`, `description`) connected by `has`, illustrating the adhesion to the metamodel.

This example shows how Jjodel handles metamodels and conforming models in a single environment, supporting flexible, precise MDE. Next, we outline the architecture, its core components and how they enable efficient MDE workflows.

2.1 Jjodel architecture

Jjodel is a cloud-based reflective modeling platform that aims to simplify MDE by minimizing accidental complexity. Supports metamodel/model definition, concrete syntax specification, and user-defined validation. Two main roles: (i) DSL engineers, who define abstract and concrete syntax and rapidly prototype web-based modeling tools; (ii) model editors, who use domain-specific tools through customizable web interfaces (Fig. 2).

Jjodel adopts a modern front-/back-end stack that provides robust services and a responsive user interface. It prioritizes expressiveness and extensibility, allowing even end users to add functionality. Components are containerized with Docker[2] for a consistent, scalable deployment across infrastructures.

The *front-end* is the primary workspace for defining and managing projects. Built with TypeScript[3] and React[4] for scalability and a dynamic user interface, it allows users to work seamlessly with metamodels, models, and concrete syntax in an integrated environment. The core is the *Jjodel Object Store*, a Redux[5]-based repository that manages all artifacts and synchronizes the application state, ensuring consistent updates via a unidirectional data flow. More details on JOM APIs for interacting with the Jjodel Object Store are provided in Sect. 2.2, highlighting how developers can query, manipulate, and extend stored artifacts seamlessly.

The Jjodel Workbench UI is the primary front-end interface. Provide editors for metamodels, models, and viewpoints to define structures, ensure conformance, and explore role-specific perspectives. Its key capability is *live reflective co-evolution*: the platform seamlessly adapts models and editors on the fly to metamodel changes by leveraging the reflective metamodeling architecture (models, metamodels, and their interdependencies), aligning with automated co-evolution [11]. This enables live synchronization across artifacts, though some structural changes still require user confirmation (Sect. 3.3). The resilience of the workbench to changes in the metamodel has been evaluated by Tolvanen et a. [35]. This adaptability eases maintenance and supports test-driven metamodel development and evolution [13]. The workbench also integrates DSL tooling–validation, interactive queries, and a console–enhancing usability and productivity. In addition, the workbench integrates tools for managing DSL aspects, including model validation, interactive queries, and console functionalities, further improving the modeling experience and improving user productivity.

The *back-end* manages persistence for the Jjodel repository, including projects, metamodels, models, concrete syntaxes, and user settings. It is built with ASP.NET Core MVC (targeting .NET 9.0)[6] and Entity Framework Core over PostgreSQL[7], with JWT[8] authentication, Serilog[9] log-

[2] https://docker.com/.

[3] https://www.typescriptlang.org/.

[4] https://react.dev/.

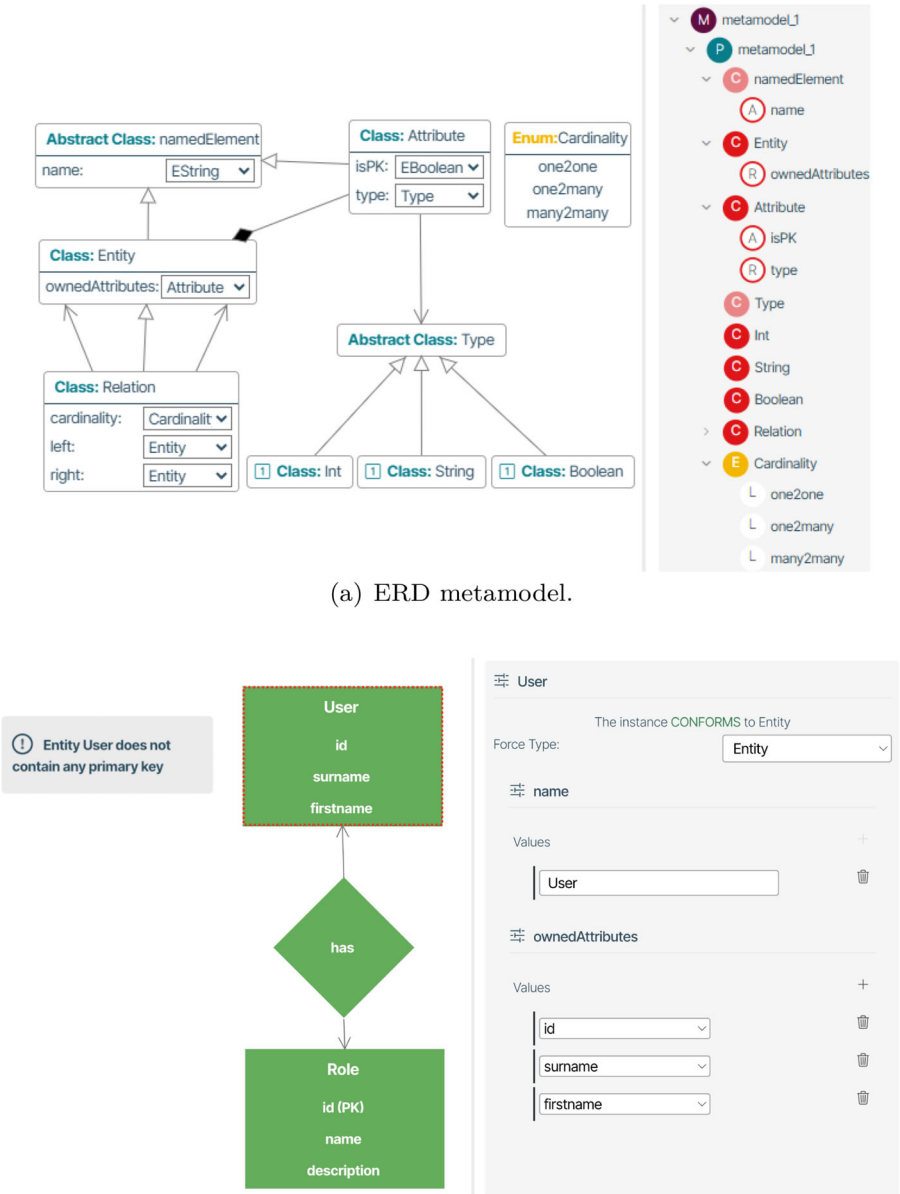
[5] https://redux.js.org/.

[6] https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0.

[7] https://www.postgresql.org.

[8] https://www.jwt.io.

[9] https://serilog.net.



(a) ERD metamodel.

(b) ERD instance.

Fig. 1 Entity relationship diagram in Jjodel.

ging, and REST APIs for external integration. It exposes three API groups: (i) *Account* (registration, authentication, profile, and password management); (ii) *Project* (CRUD on modeling projects; projects are sent as a single serialized payload, client-side compressed with async-lz-string[10]; and (iii) *ClientLog* (collection and retrieval of client-side usage data). User endpoints are documented via the Swagger UI (Fig. 3).

2.2 Jjodel object model (JOM)

JOM defines how modeling artifacts are represented and accessed in the Jjodel Object Store and exposes an API to query, evaluate, and manipulate model elements. It also supports concrete-syntax definition and provides a console for testing and debugging, offering a compact framework for model development and refinement.

It organizes modeling artifacts through three interconnected submodels, each serving a specific purpose in managing and presenting modeling artifacts:

– *Data submodel* (`data`): core artifacts, including classes, attributes, references, constraints, and their instances. The meta-metamodel includes `DModel`, `DClass`, `DAttribute`, `DReference`, `DObject`, `DValue` (Table 1); it underpins metamodels and conforming models.

[10] https://github.com/jackbister/async-lz-string/.

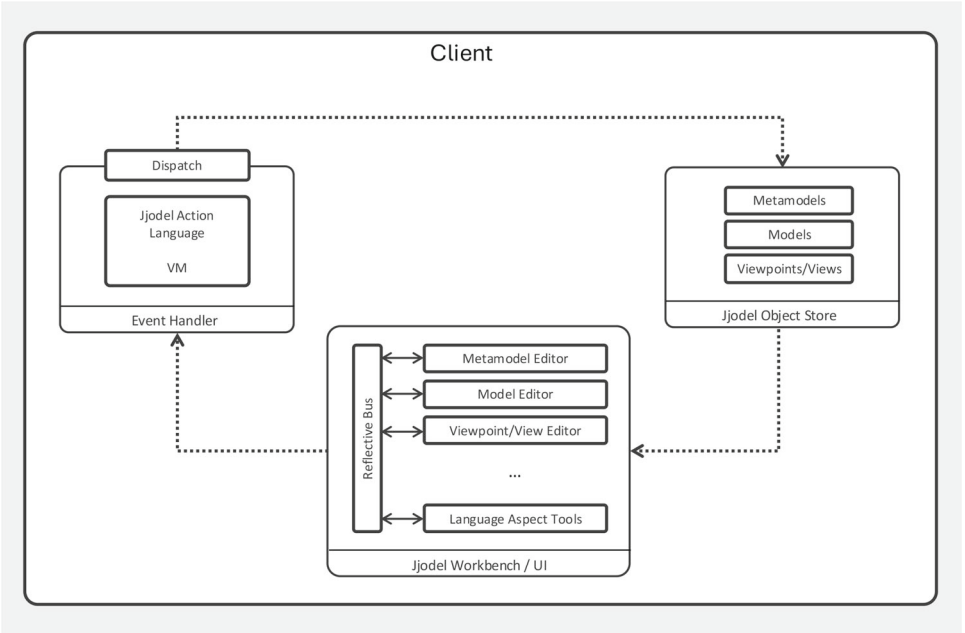


Fig. 2 Front-end architecture of Jjodel.

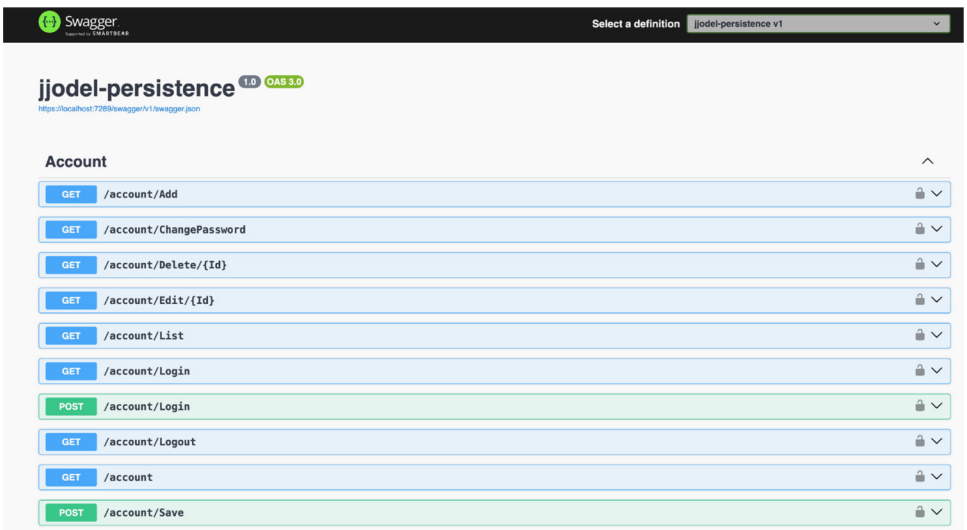


Fig. 3 A screenshot of available API accessible by dedicated Swagger UI.

– *Node submodel* (`node`): manages layout and connectivity and serves as a flexible state store (e.g., validation messages). Its links to the other submodels enable viewpoints and semantics that depend on layout (see Fig. 12).
– *View submodel* (`view`): defines concrete syntax and visual details, synchronizing `data` and `node` to keep the abstract structure consistent with its rendering.

Together, these submodels allow JOM to interact with, render, and manage modeling artifacts with precision, expressiveness, and flexibility. In particular, `data`, `node`, and `view` encapsulate the information specified respectively by the model, layout, and view definitions and expose programmatic access through the JOM API (Table 2). This interface supports querying, manipulation, and validation–for example, listing all attributes of a `DClass`, adding/removing relationships between `DObjects`, or checking metamodel constraints.

In essence, the API offers methods for retrieving and manipulating model elements, their layout configurations, and their syntactic representations. The API plays a central role in enabling JSX[11] expressions, which are particularly useful for defining navigation paths and constraints within the Jjodel environment. By allowing users to define customized constraints and navigational logic, the API facilitates flexible model querying to be used within JSX templates. This versatility not only enhances the usability of the platform, but also

[11] https://legacy.reactjs.org/docs/introducing-jsx.html.

Table 1 Core modeling constructs

Construct	Description
`DModel`	Serves as the top-level container holding Packages or Classes. DModel acts as the root of model specification.
`DPackage`	Is a logical grouping or namespace for related classes.
`DClass`	Represents a class in the metamodel. It can extend from another class and declares, attributes, and references.
`DAttribute`	Represents a field or characteristic that holds an intrinsic value that can be typed as a primitive type, i.e., integer, string, boolean, etc., or custom-defined enumeration),
`DReference`	Specifies a relationship between classes and may include multiplicity constraints (e.g., one-to-one, one-to-many). DReferences can be containment or non-containment relationships.
`DObject`	Is an instance of a DClass. It stores runtime values for each DAttribute and links pointing to other objects.
`DValue`	Is the concrete value assigned to an attribute or data field. Each DObject's DAttribute has one or more DValues, that can be a scalar (e.g., string, integer, boolean) or an enumeration literal.

Table 2 Some attributes and functions of the JOM API

Name	Scope	Description
Attributes		
`allInstances: DObject[]`	`DModel`	Array containing all instances of type `DObject`
`attributes: DAttribute[]`	`DClass`	Array of `DAttributes` associated with a `DClass`
`className: String`	`DObject`	The name of the class represented as a `String`
`extendedBy: DClass[]`	`DClass`	Array of `DClass` that extend this class
`extends: DClass[]`	`DClass`	Array of `DClass` objects that this class extends
`id: Pointer`	`Any`	Unique identifier represented as a `Pointer`
`instanceOf: DClass`	`DObject`	Reference to the `DClass` this object is an instance of
`instances: DObject[]`	`DClass`	Array containing all instances of this `DClass`
`isAbstract: Boolean`	`DClass`	Boolean indicating whether the class is abstract
`isFinal: Boolean`	`DClass`	Boolean indicating whether the class is final
`isInterface: Boolean`	`DClass`	Boolean indicating whether the class is an interface
`isMetamodel: Boolean`	`DModel`	Boolean specifying if the model belongs to a metamodel
`isPrimitive: Boolean`	`DClass`	Boolean specifying whether the type is primitive
`isRootable: Boolean`	`DClass`	Boolean specifying whether the class can be a root element
`isSingleton: Boolean`	`DClass`	Boolean indicating whether the class is a singleton
`name: String`	Global	The name of the element represented as a `String`
`objects: DObject[]`	Global	Array containing all `DObject` instances in the model
`operations: DOperation[]`	`DClass`	Array of operations (`DOperation`) defined for the class
`packages: DPackage[]`	`DModel`	Array of `DPackage` elements associated with the model
`parent: DObject`	`DObject`	Reference to the parent object, which can be of any type
`references: DReference[]`	`DClass`	Array of references (`DReference`) defined for the class
`value: DValue`	`DValue`	Single-valued data for attributes or references with an upper bound of 1
`values: DValue[]`	`DValue`	Array of multi-valued data for attributes or references
`$<name>: DValue`	`DClass`, `DModel`	Dynamically accesses child elements or instances by exact name

Table 2 continued

Name	Scope	Description
Functions		
`addAttribute( attribute: DAttribute ): DClass`	`DClass`	Adds an attribute (`DAttribute`) to the specified `DClass`
`addClass( name: String ): DPackage|DModel`	`DModel, DPackage`	Adds a `DClass` with the specified name to a `DPackage` or `DModel`
`addObject( param: JSON, className: String ): DModel`	`DModel`	Creates an instance of the specified `DClass` and initializes it with values from the JSON `param`
`addRelationship( type:DRelationType ): DClass`	`DClass`	Creates a relationship of the specified type between two `DClass` entities
`delete (): void`	`Any`	Deletes the current element and cleans up associated references or links to maintain model consistency
`executeQuery ( query: JSX Expression ): Any[]`	Global	executes a custom query and returns the matching elements
`getAttributes (): DAttribute[]`	`DClass`	Retrieves the attributes associated with the specified `DClass`.
`getElementById(id:Pointer): Any`	`DObject`	Retrieves an element by its unique identifier
`getRelationships(): DReference[]`	`DClass`	Retrieves all relationships connected to the specified `DClass`
`removeAttribute( attr: DAttribute ): DClass`	`DClass`	Removes the specified attribute from the given `DClass`
`removeRelationship ( ref: DRereference ): DClass`	`DClass`	Deletes the specified relationship from the model
`validateModel(): Error`	`DModel`	Validates the model against constraints and returns any validation errors

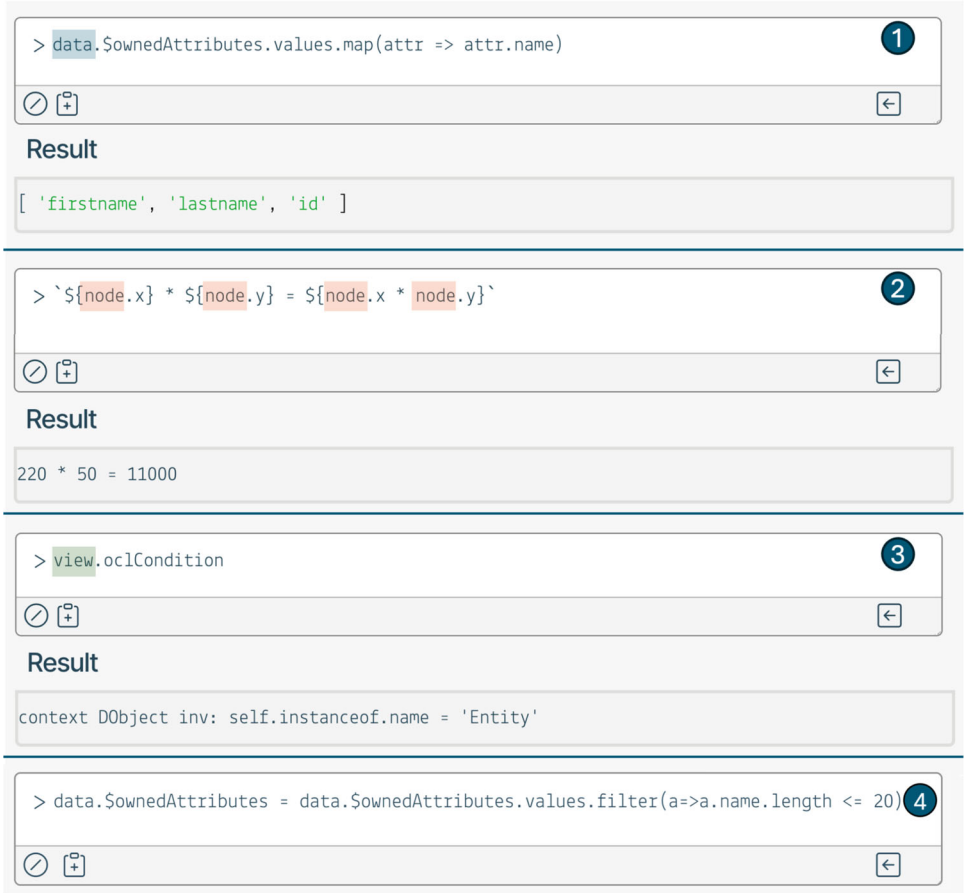
allows users to tailor their interactions to specific modeling needs and scenarios.

Figure 4 shows four expressions of the JSX exemplar that interact with the submodels `data`, `node` and `view` (highlighted blue, red, and green, respectively), demonstrated within the context of the ERD scenario. The figure illustrates the interactive console of Jjodel, which facilitates dynamic exploration and manipulation of these submodels.

Each interaction with the console is applied to the selected model element in the editor. In the following, we describe simple examples that interact with each submodel on the `User` entity:

The expression in console ❶ retrieves the names of objects pointed by a reference called `ownedAttributes`, the expression is evaluated as follows: `data` is the currently active element, equivalent to `self` in the OCL language [8], serving as a contextual reference point for querying and navigating the model elements. For example, suppose that an item of type `User` containing a reference called `ownedAttributes` pointing to N objects of type `Attribute` is selected. The $ selector followed by a name navigates to a child element that matches the said name, in this case it looks for `ownedAttributes`. if a feature with matching name is defined in the metamodel, it is retrieved. Said feature

Fig. 4 Interacting with JOM.



holds a set of properties: `type`, `upperbound`... in this case we are interested in the `values` of the collection. the `map()` function is well-known in programming and maps each element of an array to a new set of values through an argument function. In this case it retrieves the name of the pointed `Attributes`

The expression ❷ interacts with `node`, which manages spatial layout and positioning. The properties `node.x` and `node.y` represent the coordinates (or other numeric attributes) of a node, corresponding to the visual representation of a model element. The expression multiplies these values, demonstrating the dynamic retrieval and manipulation of layout-related data at run-time.

The expression ❸ references `view`, which govern the concrete syntax and visual constraints of the model elements. The `oclCondition` enforces that any `DObject` in this view must be an instance of `Entity`. This shows how OCL-like constraints can be applied to validate or control visual representations on the model front-end. These constraints integrate seamlessly into JavaScript or JSX workflows, enabling UI-level validations and custom conditional rendering.

The expression ❹ instead is modifying the model, showcasing the bidirectionality of the JOM. Assuming data is the same `User` object queried in ❶, the values are being filtered to remove the `ownedAttributes` whose name is longer than 20 characters. It is possible to insert newly created elements or replace them with pre-existing ones obtained through similar queries. This and similar instructions, altering the model or the layout can be bound to localized GUI events (onInput, onClick, onDragEnd...) or through specialized controls later defined in table3 such as `<Input/>` allowing the View designer to provide a set of tools and rules about how the model is allowed to evolve through interactions with the concrete syntax.

These examples show the expressiveness of the JOM API[12] for integrating model data with interactive UIs. JOM provides reflective, runtime access to model-typed information, enabling queries and edits per the metamodel–without generated code. To improve usability, we offer predefined facilities–high-level helpers for common tasks (element queries, layout management, visual/semantic constraints) across `data`, `node`, and `view`. Table 2 lists a subset. We adopt React JSX, familiar to many developers, which lowers the

[12] Reference guide: https://www.jjodel.io/jjodel-object-model-jjom/.

Table 3 JjDL JSX reusable components

JjDL component	Description
`<Control/>`	Adds a control panel to parameterize the syntax rendering.
`<DefaultNode/>`	Corresponds to the view definition of an instance specified as an argument.
`<Edge/>`	Corresponds to an edge among two nodes.
`<Input/>`	Permits the projectional editing of an attribute.
`<Selector/>`	Permits the projectional editing among a set of elements.
`<Slider/>`	Adds a range slider to define the value of an integer parameter.
`<Toggle/>`	Adds a toggle for a boolean parameter.
`<View/>`	It is the opening/closing tag for views.

learning curve. Together, these choices improve efficiency and accessibility for novices and experts alike.

2.3 Syntax viewpoints

Syntax viewpoints in Jjodel define concrete syntaxes of the models, specifying how instances of abstract metamodel concepts are visually represented and interacted with. Each viewpoint consists of a collection of *views* that define the mapping between abstract elements and their corresponding concrete representations. This bidirectional mapping ensures not only the visualization of abstract elements, but also, in the case of projectional editing, enables users to directly interact with the concrete representation to modify the underlying abstract model. These views are specified as graphical rules, each tailored to handle a specific class or element type. Each view consists of multiple aspects, enabling precise and intuitive model manipulation by addressing different facets of the visual representation and interaction. In particular,

- *ApplyTo*, specifies the predicate used to select the instances to which the view applies; the predicate can be defined using OCL or JavaScript.
- *Template*, defines a JSX template used to specify a component that visually represents each instance that satisfies the predicate defined in the *ApplyTo* configuration. Templates access and manipulate model data using the JOM API, i.e., declarative and functional JSX expressions, enabling dynamic and flexible visualizations. Additionally, a template can leverage predefined components provided by the Jjodel Syntax Definition (JjSD) library, which are designed to facilitate the creation of graphical syntaxes. An excerpt of these JSX components, along with their specific purposes, is listed in Table 3, offering modelers a robust foundation to define intuitive and expressive syntax representations.
- *Style*, defines customizable parameters to tailor the visual appearance of graphical components, leveraging CSS/-LESS stylings. Language engineers can specify styling variables, such as colors, border properties, and thickness, to ensure consistent and adaptable designs. As shown in Fig. 5, these parameters are configured through an intuitive interface that provides options to select color palettes, adjust numerical values (e.g. thickness), and define paths for shapes. This flexibility allows modelers to create visually cohesive and customizable syntaxes tailored to their specific requirements.
- *Events*, each view allows the modeler to define event handlers as part of Event-Condition-Action (ECA) rules, enabling the capture of user interactions such as dragging, resizing, and other actions. These events enhance interaction, allowing modelers to customize and extend the editor's behavior while dynamically manipulating the model. Furthermore, events can be used to define the semantics of the modeling notation, linking user actions to meaningful model transformations or updates. A complete list of available event handlers is shown in Table 4. The triggers do not have any parameters. It is important to note that triggers are defined within a view. The code can navigate the three submodules of Jjodel, specifically `data`, `node`, and `view`. For example, within the `data` submodule, we can access the modified model element. Figure 8 illustrates an example of how to implement a handler for the `OnDataUpdate` trigger.
- *Options*, enables fine-tuning and advanced customizations, allowing modelers to define how elements are structured and aggregated. For example, elements can be configured to appear as a list within containers or as vertices in a graph. These options provide flexibility in organizing and presenting model components, ensuring that the visual representation aligns with the desired semantics and structure of the model.

Table 4 Rule triggers

Trigger	Description
`onDataUpdate`	Triggered when model data changes.
`onDragStart`	Triggered when a drag action starts.
`whileDragging`	Triggered during a drag action.
`onDragEnd`	Triggered when a drag action ends.
`onResizeStart`	Triggered when a resize action starts.
`whileResizing`	Triggered during a resize action.
`onResizeEnd`	Triggered when a resize action ends.

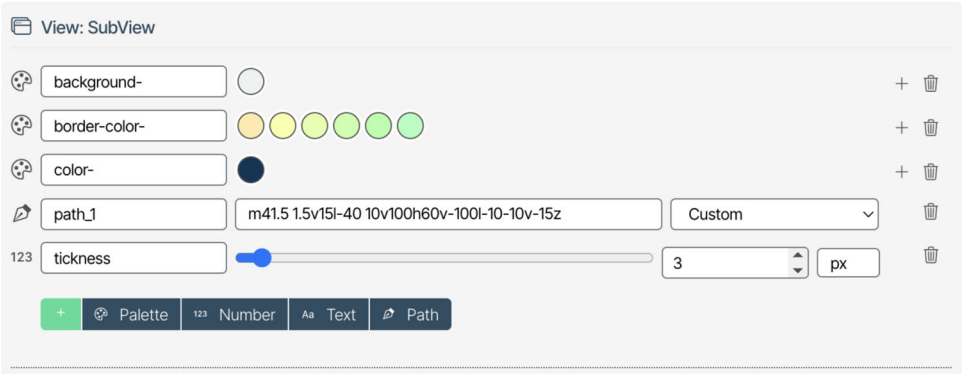


Fig. 5 Example user-defined parameters like color, thickness, and SVG paths for customizing the styling.

A default viewpoint is provided that enables a predefined reflective editor, which dynamically configures itself based on the underlying metamodel structure, allowing users to interact with models without requiring any code generation or manual editor implementation.

Using the ERD scenario, the listings 1, 2, and 3 illustrate how the templates for entities, attributes, and relations are specified within the example ERD context. Each component plays a distinct role in rendering and editing model elements in a 'projectional' style, ensuring a dynamic and customizable user experience. The component `View` serves as the root or container for a particular visualization, encapsulating the layout of all templates, such as entity, attribute, and relation templates. Within this structure, the `View` component, a key element of the Jjodel Syntax Definition (JjSD) library, plays a pivotal role in rendering sub-elements (often instances). It is a generic and polymorphic component that ensures that subelements are consistently represented, even if no custom view is explicitly defined for them.

```
<View className={'entity'}>
    <div className={'entity-header'}>
        <div className={'input-container mx-2'}>
            <Input
                data={data.$name}
                field={'value'}
                hidden={true}
                autosize={true}
                style={{backgroundColor: 'transparent', border: '
    none'}}
            />
        </div>
    </div>
    <div className={'entity-body'}>
        {data.$ownedAttributes &&
            data.$ownedAttributes.values &&
            data.$ownedAttributes.values.map((attribute) => (
                <DefaultNode data={attribute} key={attribute.id}
    />
            ))}
    </div>
    {decorators}
</View>
```

Listing 1 ERD Entity template.

In the entity template (Listing 1), attributes are iterated using the `ownedAttributes` property of the entity, and each attribute is rendered using the related view of `Attribute`. This component inductively identifies the appropriate view to apply by performing a run-time lookup based on the value of the instance that satisfies the view predicates. Since the correct view cannot always be determined statically, this dynamic mechanism allows to flexibly adapt to different scenarios and contexts. It acts as a bridge, linking to a corresponding model template or providing a minimal default rendering when no custom view is specified.

For the ERD scenario, the component for attributes is explicitly bonded to the `Attribute` template. This ensures a seamless and consistent rendering of attributes within the entity visualization while leveraging the generic and polymorphic capabilities of the component to dynamically identify and apply the appropriate view.

All three templates utilize the `Input` component to enable projectional (in place) editing of a single attribute value, such as a name or a type. In the `Entity` template, `Input` is bound to `data.$name`, representing the entity's name. The `Input` component allows bidirectional bindings between graphical elements and model elements, so when a user modifies the text in the field, the changes will automatically reflect in the semantic model. The property `field='value'` specifies that it edits the value of the feature. The component `Input` also supports additional properties; for example, the property `autosize` allows the text box to dynamically resize to fit its content, improving usability and visual clarity; or getters and setters functions to manipulate the value before displaying or modifying it (such as uppercasing, trimming...).

```
<View className="attribute">
    <div className="attribute-header">
        <div className={'input-container mx-2'}>
            <Input
                data={data}
                field={'name'}
                hidden={true}
                autosize={true}
                style={{ backgroundColor: 'transparent', border: '
    none', textAlign: 'center' }} />
            {data.$isPK.value  && '(PK)'}
            <Selector data={data} field={'type'}/>
        </div>
    </div>
    {decorators}
</View>
```

Listing 2 ERD Attribute template.

The `Attribute` template (Listing 2) extends this functionality with the inclusion of the `Selector` component, which provides a dropdown list to select an enumeration literal, such as the `type` of an attribute. This enables intuitive interaction and simplifies the selection process for users.

The `Edge` component plays a critical role in representing connections between nodes in a graphical notation, such as relationships between entities in a diagram.

```
<View className="relation">
    <div className="relation-inner">
        <div className="relation-header">
            <div className={'input-container'}>
                <Input
                    data={data.$name}
                    field={'value'}
                    hidden={true}
                    autosize={true}
                    style={{ backgroundColor: 'transparent',
  border: 'none', textAlign: 'center' }}
                />
            </div>
        </div>
    </div>
    <Edge
        view={'EdgeAssociation'}
        key={`${data.id}-${data.$left.value.id}`}
        start={node}
        end={data.$left.value.node}
    />
    <Edge
        view={'EdgeAssociation'}
        key={`${data.id}-${data.$right.value.id}`}
        start={node}
        end={data.$right.value.node}
    />
    {decorators}
</View>
```

Listing 3 ERD Relation template.

In the ERD example, the `Relation` template (Listing 3) defines two edge elements to visually represent how the `Relation` node connects to its associated left and right entities. Specifically, the first edge of the `Relation` node acts as the 'start', while the reference entity node serves as the 'end'. This means that the edge is drawn from the visual element of the relation to the entity to which it refers. By specifying which nodes to connect, the modeler can draw edges between multiple nodes in the model (e.g., from a `User` entity to a `Role` entity), enabling clear and expressive visualizations of relationships.

2.4 Validation viewpoint

The validation viewpoint enforces the rules and constraints defined in the metamodel, ensuring the general consistency of the model and structural correctness. By enabling proactive error detection and providing immediate feedback during the modeling process, it helps reduce errors and reinforces best practices. As a specialized viewpoint, the validation viewpoint focuses on identifying invalid elements within a modeling artifact and alerting users when these elements violate established constraints. This direct feedback mechanism simplifies the identification and resolution of issues, ensuring that the model remains robust and accurate. In Jjodel, validation rules are defined using views to ensure model consistency and identify errors. The `onDataUpdate` configuration is crucial to evaluating the validity of model instances, while the `state` property of a node object serves as a repository for storing validation errors associated with the model. These errors are seamlessly tied to the relevant model elements, enabling precise and context-sensitive feedback to guide modelers in solving issues.

In the ERD scenario, Fig. 1b illustrates a validation marker triggered when an entity lacks a primary key attribute. To implement this validation rule, the `onDataUpdate` handler is used to evaluate the model and identify any violations. Validation errors are stored in the `state` property of the corresponding node, ensuring that errors are persistently tied to the affected elements. Figure 6 demonstrates how to enforce the constraint of requiring at least one primary key for each entity. The implementation initializes an `err` variable to `undefined`. Using the JOM framework, model elements are navigated to check if at least one of the `ownedAttributes` is marked as a primary key. If no primary key is found, the `err` variable is updated with an error message. Finally, the error message is stored in the `state` property of the corresponding node. If a node contains an error message, a validation marker (e.g., an alert dialog) is displayed near the affected entity (highlight with a red dashed , as shown in Fig. 1b. This approach ensures that modelers receive immediate visual feedback, enabling them to address validation errors efficiently and maintain the integrity of the modeling artifact.

3 Flexible modeling in Jjodel

A core objective of Jjodel is to offer flexibility, enabling end users and language engineers to navigate the complexity and richness of modern multifaceted domains. Flexibility is not just a feature, but a foundational principle embedded within its design and technology stack. This approach allows language designers to tailor the modeling environment to their specific needs without encountering the challenges typically associated with custom coding or inflexible extension mechanisms. With Jjodel, users can create untyped objects and enrich them incrementally by adding attributes and references without requiring an initial definition in the metamodel. This allows for exploratory modeling where the structure of the system can emerge organically. At any stage, users can assign a type to an object, either by referencing an existing class or defining a new one. Jjodel handles this transition gracefully by hiding features that are not part of the selected class and adding those that are missing. Crucially, the system retains all original information, allowing users to revert typing decisions without losing data. This approach supports flexible modeling workflows that accommodate both informal prototyping and the gradual emergence of formal structure [27] through untyped objects.

Flexible modeling: from untyped instances to emerging structure. Untyped artifacts are first-class `DObjects` with a property bag. Users may add ad-hoc attributes and references without a prior class. When a type is later assigned (e.g., `Person`), the Object Store (i) materializes the class features, (ii) binds matching ad-hoc properties to those features,

Fig. 6 Validation rule.

View: Entity PK

Default Events

onDataUpdate

```
let err = undefined;

if (data && data.$ownedAttributes &&
        data.$ownedAttributes.values.filter(a => a.$isPK.value).length === 0)
    err = "Entity " + data.$name.value + ' does not contain any primary key';

node.state = {error_naming:err};
```

and (iii) retains non-matching properties as *unbound* data (lossless). The reflective layer adapts values on access (e.g., `"23"`→`23`, upper/lower-bound enforcement), flags unresolved cases (e.g., missing references) for user confirmation, and allows reverting the typing decision without data loss. *Limitations and future work.* Beyond basic binding and casting, schema learning and type-suggestion strategies (e.g., inferring features from recurring unbound properties) are out of scope here and left for future work; the architecture admits their integration as optional services.

Transition to UI examples. Having outlined how structure emerges from untyped instances and how values are adapted reflectively, we now illustrate how these mechanisms surface at the UI level. The following examples–*grid snapping* and *semantic zooming* [15]–show how view definitions exploit the Object Store and reflective APIs to provide immediate, model-aware feedback without custom code generation.

3.1 Grid-snapping

Grid snapping improves layout precision and usability. In Jjodel, it is enabled via JSX templates, CSS styles, and Event-Condition-Action (ECA) rules [38], offering a lightweight and modular setup. This design, elaborated in the following paragraphs, ensures extensibility with minimal effort.

Step 1: Defining the Grid Style. The dot-based grid style is implemented through a simple CSS class definition, as shown below:

```
.grid {
    background-image: radial-gradient(silver 1
    px, transparent 0);
    background-size: 15px 15px; /* 15px x 15px
    grid */
}
```

This class renders the grid as a pattern of evenly spaced dots. In the next step, we show how to dynamically apply or remove the `.grid` class in the model view.

Step 2: Adding a Toggle Command for Grid Control. Jjodel enables the extension of the editor functionalities using the `<Control/>` component within the model view[13]. In this example, a `<Toggle/>` component is added to dynamically enable or disable the grid. The toggle operates on a Boolean parameter, `grid`, which is defined on the template page, as illustrated in Fig. 7.

The parameter `grid` serves as a shortcut for the expression:

```
node.state.grid ?? false
```

which initializes the parameter in the `node` submodel to `false`. The updated template for the model view is shown below:

```
<View className={model ${grid && 'grid'}}>
    ...
    <Control title={'Workbench'} payoff={'
    Controls'}>
        <Toggle name={'grid'} title={'Grid'}
    node={node} />
    </Control>
</View>
```

Here, the conditional expression `${grid && 'grid'}` dynamically applies the `.grid` class whenever the value of the parameter `grid` is `true`, allowing the feature.

Step 3: Enabling Snap-to-Grid Behavior. To implement the snap-to-grid functionality, an ECA rule is defined, as shown in Fig. 8. This rule takes advantage of the `grid` parameter to ensure that the elements align with the nearest grid vertex when moved.

The event `onDataUpdate` ❶ is triggered when an element is moved. Condition ❷ verifies the state of the parameter `grid`, and if enabled, the action ❸ adjusts the coordinates `x` and `y` of the model element to align with the nearest grid vertex.

3.2 Semantic zooming

Semantic zooming [15] is an advanced interaction paradigm that dynamically adjusts the level of detail presented in a model based on the zoom level. Unlike traditional zooming, which merely magnifies or shrinks visual elements, semantic zooming alters the content itself to better align with the user's context and focus. In modeling, this approach offers several benefits, including enhanced usability, reduced cognitive load, and more efficient navigation by presenting only the most relevant details at each zoom level [32]. Recent work

[13] The model view is included in the default viewpoint but can be cloned and customized as needed.

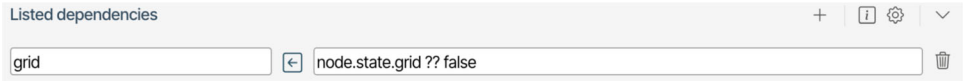


Fig. 7 The user-defined `grid` parameter.

Fig. 8 ECA rule for the *snap-to-grid* feature.

by De Carlo et al. [12] demonstrated the implementation of semantic zooming in the Eclipse GLSP platform, highlighting both its potential and its practical challenges. Their experience revealed that realizing semantic zooming in web-based modeling environments often requires considerable custom development and careful handling of performance trade-offs, particularly around zoom-triggered server-client interactions and dynamic layout adjustments. These insights underscore the importance of designing flexible, yet efficient mechanisms for managing zoom-level semantics in modern modeling tools.

In Jjodel, semantic zooming is achieved by linking different visual representations to predefined zoom thresholds. These representations are dynamically updated as users adjust the zoom level, ensuring that the displayed content remains contextually relevant and appropriately detailed. Although semantic zooming might seem more conceptually complex than features such as grid snapping, it is seamlessly implemented in Jjodel thanks to its flexible templating framework. This framework allows for the integration of controls, such as parameters similar to `grid` discussed above, that dynamically influence visualization without affecting the underlying models.

Semantic zooming in the default syntax is implemented through the following steps:

1. Define an integer parameter, such as `level`, to represent the current zoom level.
2. Add a slider control to the user interface that allows users to interactively adjust the zoom level.
3. Divide the template into sections corresponding to different zoom levels, ensuring that the level of detail displayed dynamically adjusts based on the parameter's value.

Each of these steps is elaborated in the following paragraphs.

Step 1: Defining a new integer parameter

```
node.state.level ?? 3
```

(see Fig. 9); this stores `level` in the `node` submodel and defaults it to 3 if unset.

Step 2: Adding a Slider Control for Zoom Level Adjustment.

```
<View className={'model'}>
  ...
  <Control title={'Workbench'} payoff={'Zoom Controls'}>
    <Slider name={'level'} title={'Detail level'} node={node} min={0} max={3} />
  </Control>
</View>
```

The `node` and `name` props bind the slider to `level`; `min`/`max` define the detail range.

Step 3: Slicing the Template for Semantic Zooming. Level 0 shows package summaries; level 1 shows classes, enumerators, and objects without features; level 2 adds summaries (counts of properties; Fig. 10); level 3 (default) shows feature names and types.

A corresponding JSX slice enables `level == 2 &&` to compute summaries (attrs/refs/ops) and, for level 3, lists features as sub-elements, enabling component-based composition. Similar slices apply to Packages, Enumerators, and Objects.

This approach keeps semantic zooming straightforward and effective: using `level` and Jjodel's templating, it requires no custom code generation. Both semantic zooming and grid snapping are implemented entirely via view definitions and could be authored by users, illustrating the platform's extensibility.

3.3 Co-evolution

Jjodel supports transparent and lossless co-evolution heuristics. Using reflective APIs (e.g., proxies), whenever a modeling element is accessed, the runtime determines whether the returned value must be adapted to remain consistent with the current metamodel. The persisted data are not destructively rewritten; instead, a consistent projection is produced for the editor and JOM queries.

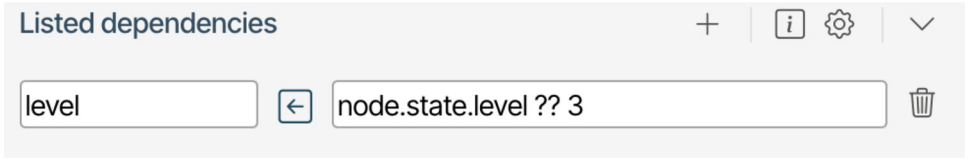


Fig. 9 User-defined `level` parameter in the Model view.

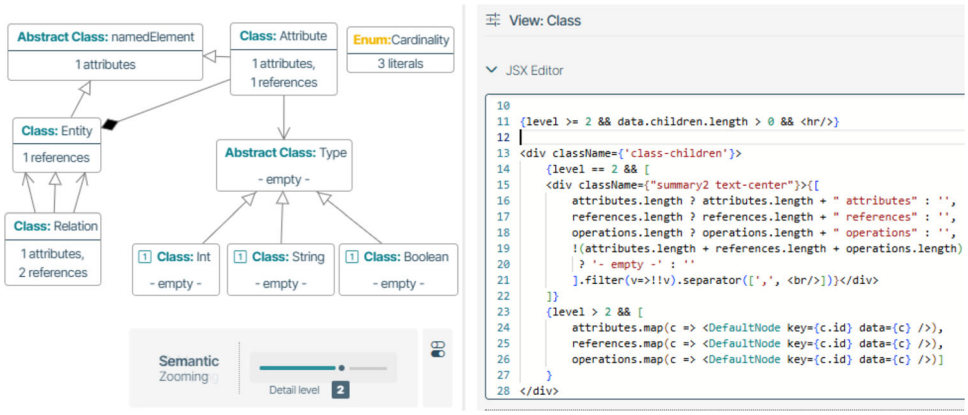


Fig. 10 The metamodel of Fig. 1a at semantic zoom level 2.

Mechanism and example. Consider a feature `list` typed `string` with upper bound 5 and current values `['1','a2','b','4c']`. If the metamodel is revised so that `list` has `type=int` and upper bound 3, the existing values become non-conformant and are adapted on-the-fly (string→int casting followed by multiplicity enforcement), yielding `[1, 2, 0]`. To restore consistency, whenever the list is queried for its values, the JOM API returns a transformed copy cast to `int` and truncated to the new upper bound. When casting is impossible (e.g., `'b'`), the default value for the target type is used (for integers, `0`), so both the GUI and JOM queries show `[1, 2, 0]`.

Now suppose that the user changes the third value from 0 (originally `'b'`) to the integer −3, and then restores the type of feature and the upper bound to `string` and 4. Reading the values will now return `['1','a2','-3','4c']` rather than `['1','2','-3',"]`. In line with the principle of lossless transformation, the property retains the original user input; the only value lost is `'a2'` because it was explicitly overwritten by the user during the edit.

For lower-bound increases in *attributes*, the queried values are made consistent by padding with the default entries. For lower-bound increases in *references*, automatic resolution is not possible: missing values appear as `null` placeholders, which trigger validation views and are left to the user to resolve. Similarly, if a class gains or loses a feature (e.g., due to adding/removing an extended relationship), all instances of that class dynamically gain or lose the corresponding slots; if a feature is later restored, its original values reappear, preserving losslessness.

Framework and protocol. We evaluated co-evolution support following the tool-agnostic framework of Tolvanen et al. [36]. The framework varies (i) the *location of change* (metamodel, constraints, notation) and (ii) the *nature of change* (add, rename, remove, change), producing 12 scenarios; for each scenario, it assesses the impact on six locations (metamodel, constraints, notation, generator, tooling, models). Each location receives a score 1-5 (half-steps allowed) based on the behavior of the editor/tool; the *overall* score of the scenario is the minimum of its six subscores. We also record the effort/time to carry out each scenario, as suggested by the framework. To maintain comparability, we reuse the Gothic state machine example and the same evaluation procedure.

Scenarios exercised. We executed all scenarios applicable to the current set of features of Jjodel. As Jjodel does not yet provide general *constraint* support, scenarios whose location of change is "constraints" are marked as not applicable (the framework allows 'X' for unsupported functionality). The results and detailed information per scenario can be found in [36].

4 Jjodel in practice

To illustrate the practical applicability of Jjodel, we employ a simple but expressive expression language, specifically developed by the authors for this study. Although not based on an existing external case study, the language was designed as a didactic artifact, providing a minimal but sufficiently expressive structure to demonstrate key features of the platform. Its abstract syntax, shown in Fig. 11, defines a hierarchy of abstract classes, including `Expression`, `Binary Expression`, `Commutative Expression`, and `Non Commutative Expression`, along with compositional references such as `left` and `right`. This structure

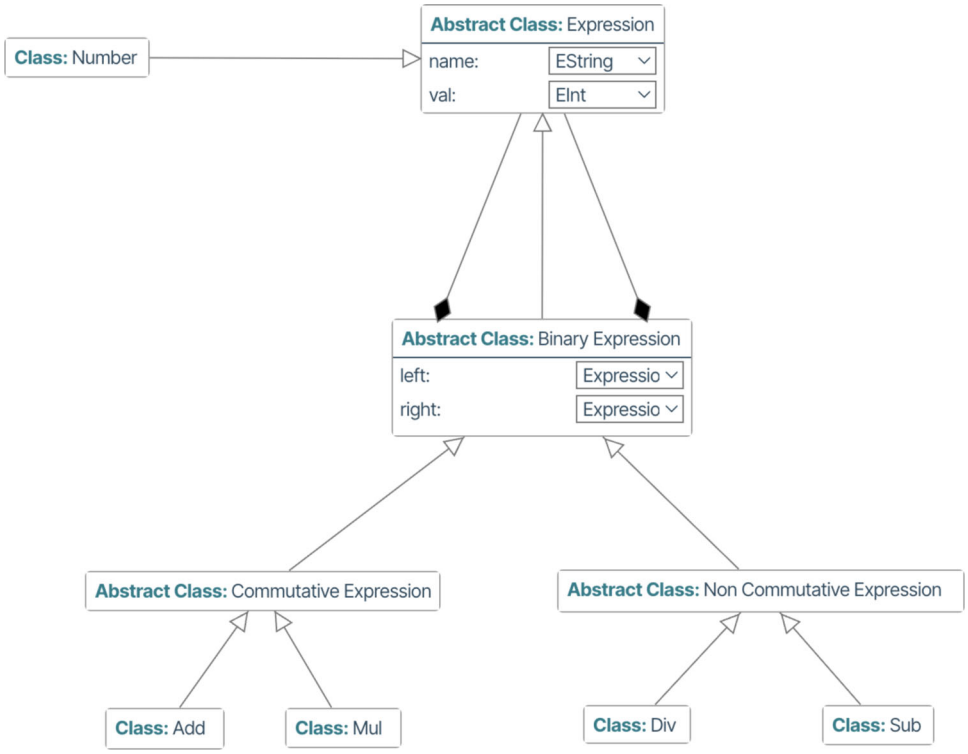


Fig. 11 The metamodel of a simple Expression Language.

supports both atomic constructs (e.g., `Number`) and operations (e.g., `Add`, `Sub`, `Mul`, `Div`). Each expression carries an attribute `val` that denotes its computed value, whether a literal constant or the result of an evaluated operation. This formalization captures the basic semantics of algebraic expressions while supporting their positional and syntactic representation, as discussed in the remainder of this section. It should be noted that in this example, we deliberately *pollute* the abstract syntax by embedding evaluative information, namely, computed values directly within the syntax nodes. Although Jjodel offers a principled mechanism for associating such information through its notion of *state*, thereby preserving a clear separation between structural and behavioral concerns, the adoption of this mechanism involves engaging with the underlying computational model. Given that a formal exposition of the computational foundations falls outside the scope of this work, we have opted for a more straightforward approach that integrates the evaluation results (`val`) within the abstract syntax itself. A rigorous treatment of the state mechanism and its role within the execution semantics of Jjodel is deferred to future research.

For example, consider the following expressions.

$$1000 - ((212 + 2) + 102) = 684 \quad (1)$$

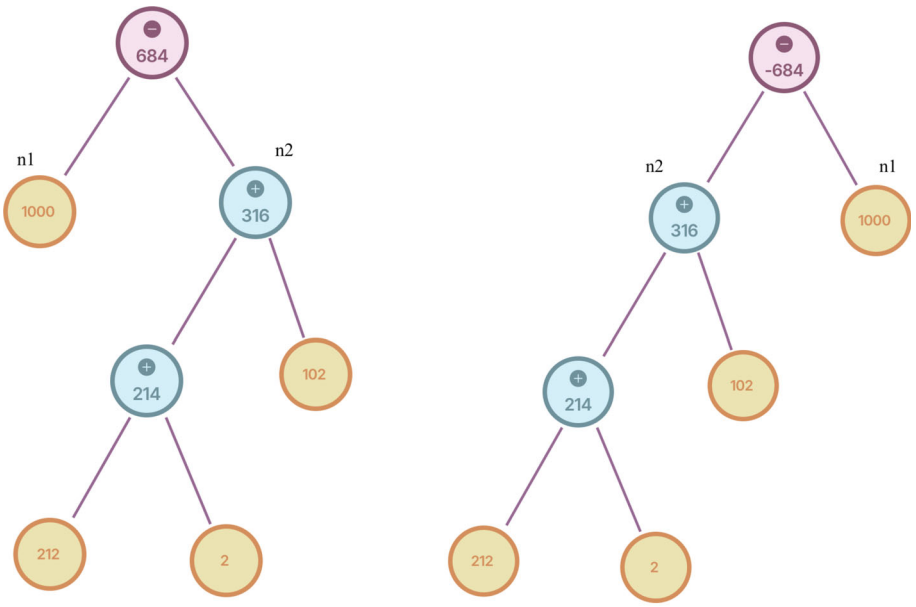
$$((212 + 2) + 102) - 1000 = -684 \quad (2)$$

These expressions consist of nested binary operations, with subtraction acting as the root operation. Figure 12 illustrates their tree-based representations, each rendered using a dedicated concrete syntax, that is, a viewpoint. The corresponding abstract syntax is provided in Fig. 13. In particular, the two models differ only in the configuration of the references `left` and `right` in `Sub_1`, which, based on the geometric positioning in Fig. 12, point to the same model elements, namely `N4` and `Add_1`, but in reverse roles.

Figure 13 shows the *visual* viewpoint consisting of dedicated views for each metamodel concept: `NumberView`, `AddView`, `SubView`, `DivView`, `MultView`, and `Edge`). Their computed values differ due to the positional nature of algebraic notation. For instance, the subtraction, being non-commutative, depends on operand order (e.g., $(10 - 5) \neq (5 - 10)$) as reflected in the references `left` and `right` in the subtraction instances. To address this, Jjodel encodes layout information that can be used to detect when an operand is dragging to the right or left of the other operand. Each time this occurs, an event is detected, and an action exchanges the elements referred to by `left` and `right`.

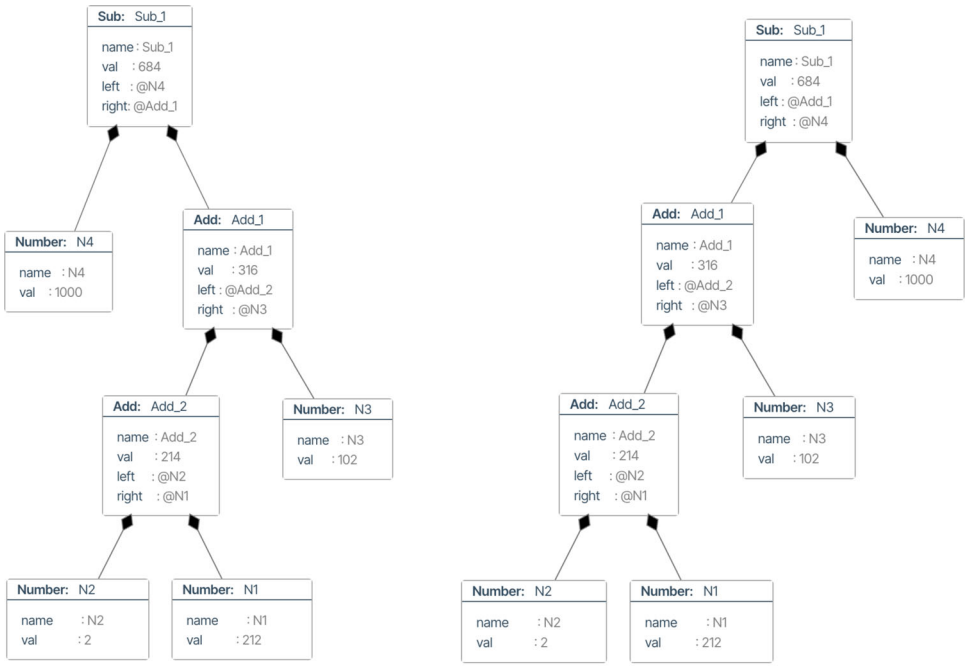
Jjodel leverages its viewpoints (as shown in Fig. 14) to manage these semantic aspects. For example, `NumberView` renders the numbers as colored nodes, assigning values to their `val` property using JSX templates and CSS rules. Composite operations such as addition (`AddView`) and subtraction (`SubView`) are governed by ECA rules [38].

They consist of three components:



(a) Expression (1) (b) Expression (2)

Fig. 12 Expressions illustrating the influence of positional semantics.



(a) Expression (1) (b) Expression (2)

Fig. 13 Abstract syntax of expressions (1) and (2).

– Event, in this case, the event `onDataUpdate` is triggered whenever the data, including the layout, associated with a model element change;
– Condition, defining the predicate to determine the elements to which a view applies;
– Action, specifies how values are computed or updated using features defined in the metamodel (e.g., `$val`, `$left`, `$right`). For example, Fig. 15 shows the `onDataUpdate` event handler for the added instances.

The subtraction action is similar; however, as already mentioned, it is necessary to exchange the two subtraction operands whenever needed, as illustrated in the following `onDragEnd` rule:

```
if (data.$left.value.node.x < data.$right.value
    .node.x) {
    let temp = data.$left.value;
    data.$left.value = data.$right.value;
    data.$right.value = temp;
}
```

Fig. 14 Viewpoints and their corresponding views.

Fig. 15 `onDataUpdate` event handler implementation for addition instances.

Although this action does not belong to the intrinsic semantics of the language, it reflects the behavior induced by the user's interaction with the notation as mediated by the editor environment.

A key feature of Jjodel lies in its ability to seamlessly propagate dynamic updates. With reference to Fig. 16, whenever we change the value of a number, an inductive sequence of action executions is performed. Let us see more in detail what happens if we change, for example, the value contained in the node n_0.

step 1 At time t_0, the user updates the value of the expression e_0 associated with node n_0, modifying it from 212 to 112. Note that the figure depicts only the resulting value after the modification;

step 2 The change in step 1 triggers an `onDataUpdate` event on the addition node n_2 and the associated sematic rule is executed, updating the expression e_1 to 114 at time $t_1 > t_0$;

step 3 The update cascades to expression e_2 on node n_5, recalculating the value to 216 at time $t_2 > t_1$;

step 4 Finally, in turn the action associated with the subtraction re-evaluate the expression e_4 on node n_6 at time $t_3 > t_2$. At this stage, no further action is required.

By automating dependency resolution, Jjodel ensures consistency and eliminates redundant computations, allowing users to focus on high-level design tasks.

5 Reflections and lessons learned

The development of Jjodel offered an opportunity to reflect on a range of architectural, usability, and integration choices made in the context of building a modern, reactive MDE platform. Although some insights align with broader engineering practices, their specific application in the domain of modeling tool development presents nuances that may be of interest to the MDE community. This section reports these insights with a focus on practical challenges, design trade-offs, and the rationale behind technical decisions.

5.1 Technology choices

The adoption of React, Redux and, more marginally, Node.js played a decisive role in shaping the platform architecture. The component-based structure of React enabled modularity and customization of concrete syntax, while Redux provided a serializable, action-based state management that facilitated real-time collaboration. However, these technologies required a substantial change in development paradigms and required iterative adaptations to align with metamodel-centric design.

Initial iterations of the architecture, including an early implementation based on Angular[14], proved less suitable due

[14] AngularJS 1.8.2 was adopted during its official long-term support period. After support ended in December 2021, only minimal updates followed, with maintenance shifting to community-led initiatives. Meanwhile, newer frameworks such as React and Vue.js have continued to evolve at a different pace. Although AngularJS remains in use in some legacy systems, its declining ecosystem and increasing security concerns highlight the importance of assessing long-term maintenance when relying on discontinued technologies.

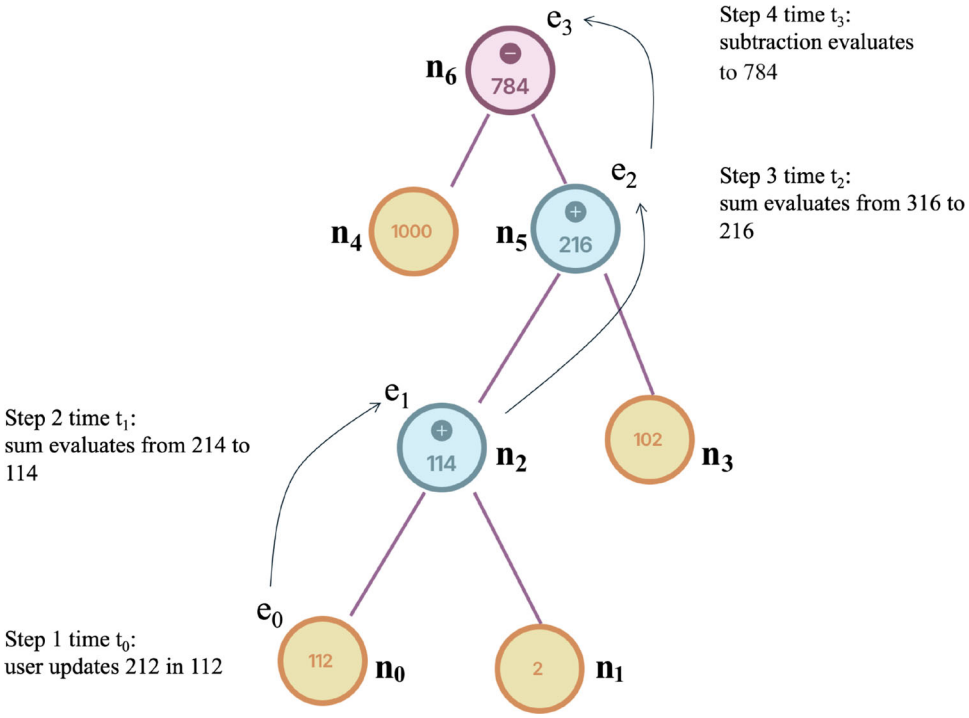


Fig. 16 Dynamic update propagation workflow.

to the lack of runtime interpretation mechanisms and the comparatively steeper learning curve. Subsequent attempts to compensate for these limitations through custom templating introduced additional complexity, leading to maintenance overhead and performance bottlenecks. Ultimately, React emerged as the most viable alternative, offering a favorable balance between usability and integration flexibility. In particular, its native support for JSX-based templating aligns well with the platform requirements for dynamic syntax definition.

These shifts underline the importance of selecting technologies not only based on popularity or capabilities, but also on their alignment with the requirements of extensible and reactive modeling platforms. Importantly, several insights, such as the use of JavaScript proxies (see Sect. 5.3) to intercept and manage model edits, emerged not from up-front planning, but through trial, failure, and the reconsideration of core assumptions.

5.2 Balancing flexibility and simplicity

Designing a platform that balances advanced modeling capabilities with accessibility for non-expert users proved to be a significant challenge. To address this, Jjodel adopted a progressive disclosure strategy [33], a widely recognized HCI principle aimed at managing the complexity of the interface. In this approach, core features are presented upfront, while more advanced functionalities are deferred to user-configurable options. This allows users to gradually explore the capabilities of the platform without being overwhelmed at first. Informal feedback from early deployments in graduate-level courses supports the efficacy of this strategy: students unfamiliar with MDE reported that the reduced complexity of the interface facilitated initial participation and eased the learning curve.

5.3 Lowering adoption barriers for MDE tools

A primary design goal of Jjodel was to address and remove practical barriers that often hinder the adoption of Model-Driven Engineering (MDE) tools, such as complex installation procedures, extensive configuration requirements, and reliance on specialized runtime environments. By adopting a web-based architecture in which the core logic runs entirely in the browser, Jjodel eliminates the need for local setup, back-end customization, or code generation. This design choice ensures that the tool can be accessed directly through standard web interfaces, making it immediately available to a broader audience.

To further ease onboarding, the platform's features were deliberately mapped to familiar notations: keywords are aligned with UML/OCL, syntax definitions are based on JSX/SVG/CSS, and dynamic behaviors are expressed through declarative JSX expressions. The JOM (Jjodel Object Model), the metamodeling architecture of the platform, was specifically designed to support intuitive navigation and manipulation of the model elements.

The use of JavaScript proxies proved especially valuable for transparently intercepting property accesses and updates, enabling the integration of custom platform behaviors in a decoupled way without imposing constraints on modeling syntax or compromising runtime performance. This infrastructure supports concise navigational expressions, allows platform-specific behaviors to be triggered during query evaluation, and provides a flexible foundation for extending semantics or introducing side effects without altering the core query syntax.

An integrated interactive console further reduces the adoption barrier by providing validation, auto-completion, and live previews, allowing new users to experiment and learn progressively. Together, these design decisions focus on making Jjodel accessible, approachable, and deployable without the traditional overhead of MDE tools, thus encouraging broader experimentation and use in both academic and industrial contexts.

5.4 Rethinking concrete syntax integration

The integration of React facilitated a dynamic rendering model for concrete syntax that departed from the static templates commonly used in MDE tools. Users could define visual viewpoints with interactive, component-based behaviors. This flexibility proved essential in supporting layout-sensitive languages, such as the expression DSL, where spatial positioning carries semantic weight.

Beyond its use in the implementation of the platform itself, React also served as the foundation for defining concrete syntax through its native templating mechanism. The approach enabled language designers to specify the rendering of model elements using declarative JSX expressions, which can embed navigational queries and reactive behaviors directly within the syntax templates. In this way, models can be queried and visualized within the same abstraction, allowing for a tighter integration between concrete syntax and semantic content. The React rendering pipeline and state management supported live editing, enabling immediate feedback and structural consistency during user interactions. This dual use of React, both as an implementation framework and as a syntax definition mechanism, demonstrates its versatility in the context of modeling tools.

In this respect, we have conducted a more detailed analysis of the use of JSX for modeling in [7], where we provide an extended comparison between OCL and JSX queries in terms of expressiveness, usability, and integration potential. Although the full implications of this integration merit further exploration, our experience suggests that modern front-end frameworks can play a fundamental role in redefining the architecture and usability of MDE tools.

5.5 Feedback-driven refinement

The development of Jjodel adopted an iterative user-centered approach, in which continuous interaction with users, from graduate students to experienced modelers, guided key design refinements. Early feedback highlighted the importance of features such as immediate validation, undo/redo support, and layout guidance. These observations directly informed the implementation of reactive syntax updates, serializable application state, and contextual editing cues.

To date, Jjodel has been adopted by a relatively large number of users, particularly in academic contexts. Although the platform has been accessed by a broader audience, our feedback analysis is based on a subset of more than 30 individuals, primarily graduate students and researchers from the University of L'Aquila in Italy and the Mälardalens University in Sweden, who interacted with the tool in the context of courses and research activities. Although this does not constitute a formal empirical evaluation, the qualitative feedback obtained from this group provided meaningful information that helped align the functionality of the platform with the needs and expectations of the user.

5.6 Navigating trade-offs in standard compliance

Departing from Eclipse GLSP was an explicit architectural decision in Jjodel. GLSP adopts a server-centric interaction model: modeling artifacts reside on the server, and the client, typically implemented with TypeScript/SVG and Sprotty[15] (which uses a Snabbdom-based virtual DOM), communicates via a graph/command protocol. In such a setup, modifications to the backend model (by users or automation) are not inherently visible on the client unless propagated through protocol-specific messages and handlers. In essence, most of the "intelligence" in GLSP is placed on the backend.

Jjodel places most of the decision logic on the client. Its Object Store encodes *Ecore* structure and semantics natively, so client-side representations and editing operations directly reflect classes, attributes, references, multiplicities, and containment/opposites. Avoiding a protocol mapping layer enables lightweight reflective co-evolution, validation, and live type changes without intermediate translations.

This difference in architecture does not preclude Ecore compatibility; in contrast, Jjodel aligns closely with Ecore concepts while remaining independent of GLSP/EMF. Interoperability can be provided via import/export mechanisms. The choice reflects our goals of immediate feedback and reflective behavior in the browser; other contexts may reasonably favor a server-centric design.

[15] https://sprotty.org.

5.7 Limitations of OCL.js and emerging alternatives

Our initial integration of OCL.js[16], an implementation of the Object Constraint Language in JavaScript, was eventually reconsidered due to limited language coverage and lack of ongoing maintenance. To address this, we experimented with the use of JSX-based expressions, already part of the templating language in Jjodel, to specify constraints and predicates.

Although this solution improved internal consistency and simplified user interaction, it also raised new questions about expressiveness, formality, and alignment with modeling standards. In particular, JSX lacks a formal semantic foundation in the context of MDE, which led us to examine how well it aligns with OCL in terms of navigation, constraint specification, and expressiveness. To this end, we conducted an in-depth empirical comparison between OCL and JSX, presented in our recent work [7], where we evaluated their relative strengths in a curated set of modeling scenarios.

The investigation confirmed that JSX can be a practical alternative to OCL.js to express structural constraints when used within a single-page application architecture like Jjodel. It offers improved conciseness, readability, and usability in modern web-based modeling environments. Moreover, JSX is increasingly familiar to students and early-career developers due to its widespread use in frontend development, which facilitates its adoption in educational and prototyping contexts. However, more research is needed to evaluate its suitability as a constraint language and to develop formal foundations for its long-term use in model-driven engineering.

5.8 Lessons from failure

Unsurprisingly, some of the most instructive lessons emerged from design failures and their resolution. Early architectural decisions, such as adopting Angular or trying to design a custom templating engine, proved unsustainable and hindered progress. The eventual transition to React, informed by these experiences, enabled a simpler and more robust interface.

A particularly productive insight came from adopting JavaScript proxies to intercept model updates, which allowed intuitive syntax (e.g., `model.name = "foo"`) to trigger backend logic such as undo operations or validation checks without requiring users to learn complex method-based APIs.

In the context of state management, Redux was initially adopted for its support of serializable states and, more critically, for its ability to serialize state transitions, which enables consistent change propagation across multiple clients. Although originally motivated by architectural constraints, this decision unexpectedly facilitated features such as history tracking, undo and redo operations, and reproducible bug scenarios. This outcome illustrates how initial design constraints can influence the emergence of secondary capabilities, depending on the architectural context. Although the consequence is quite obvious in hindsight, it suggests that certain design choices should be evaluated early in the process to better understand the opportunities and challenges they may entail.

[16] https://ocl.stekoe.de/.

6 Related work

This section reviews various MDE platforms, discussing their features, strengths, and limitations to position Jjodel within the current landscape of MDE tools. Hutchinson et al.[21] provide extensive empirical insights into the adoption of MDE through a survey and multiple industry case studies, identifying critical factors beyond technical aspects, such as organizational change, management support and social dynamics, as essential determinants of successful MDE implementation. Complementing these findings, Whittle et al.[37] specifically explore the role of tools in the adoption of industrial MDE, highlighting that tooling issues are often intertwined with social and organizational contexts. They propose a detailed taxonomy of tool-related considerations based on industry data, which categorizes technical, internal, and external organizational, as well as social factors affecting the use and effectiveness of MDE tools. Both works underscore that successful adoption of MDE relies not only on tool features and maturity, but also significantly on how tools align with organizational processes, culture, and user expectations. These findings resonate with the rationale behind Jjodel, which emphasizes ease-of-use, intuitive interaction, and alignment with modern collaborative workflows, addressing many of the organizational and social barriers identified by previous studies.

MetaEdit+ [22] is a mature and widely recognized platform for domain-specific modeling (DSM). It uses the GOPPRR meta-metamodeling language [23] to define domain-specific concepts and relationships, offering extensive customization capabilities. Despite its introduction in 1995 and its reliance on an outdated technology stack, MetaEdit+ features a reflective and integrated architecture that provides robust operational support through co-evolution mechanisms. These capabilities simplify complex tasks, such as ensuring metamodel consistency and managing artifacts, challenges that remain prevalent in many other MDE tools. In comparison, Jjodel inherits the strengths of MetaEdit +, such as its reflective and integrated architecture. Leveraging a native cloud architecture and cutting-edge front-end technologies, Jjodel offers a more *transparent* user experience by minimizing the accidental complexity, alongside support for automated syntax co-evolution.

GMF (Graphical Modeling Framework)[17] and Eugenia[18], both built on the Eclipse Modeling Framework (EMF), facilitate the specification and automated generation of graphical modeling editors. GMF provides a framework for defining diagram editors by combining EMF-based domain models with graphical definitions, while Eugenia streamlines this process by allowing the graphical aspects to be inferred from annotated Ecore models, reducing manual configuration effort. Although these tools are powerful for defining graphical representations, their reliance on Eclipse-based infrastructures, complex configurations, and generative workflows can pose significant challenges for non-expert users. Furthermore, their generative and component-based architectures make runtime adaptation more difficult, which can limit flexibility in iterative and agile development contexts. In contrast, Jjodel addresses these limitations through an intuitive low-code environment with dynamic syntax customization, offering a more adaptable and accessible option for modern team-based modeling scenarios.

Sirius[19], also built on the Eclipse ecosystem, introduces viewpoint-based modeling, allowing users to define and work with multiple perspectives in a single model. While this approach reduces programming effort, Sirius remains configuration-intensive and lacks native support for dynamic syntax co-evolution, limiting its adaptability in iterative and distributed development workflows. In contrast, the reflective architecture of Jjodel incorporates built-in governance mechanisms, often absent in tools such as Sirius, that simplify complex tasks such as coevolution and validation of metamodels, providing a more streamlined and robust modeling experience.

Building on the concepts of its predecessor, Sirius Web [16] transitions to a native web environment, using React to deliver a modern interface. It introduces collaborative modeling features and supports various representation types, including diagrams, forms, and Gantt charts. While these advancements address some limitations of Sirius Desktop, Sirius Web still suffers from significant configuration overhead and limited dynamic adaptability, which constrain its effectiveness in highly iterative and fast-paced projects.

Sprotty[20] and Gentleman [25] address narrower roles in the MDE ecosystem. Sprotty is a web-based visualization framework focused on rendering lightweight models, while Gentleman provides a flexible projectional editing interface for directly modifying abstract syntax trees. Although these tools excel in specific use cases, they lack deeper integration with metamodel evolution. Jjodel combines the simplicity and lightweight design of tools like Sprotty and Gentleman with advanced capabilities such as automated syntax co-evolution, making it a more comprehensive solution.

A recent paper by Metin et al. [29] explores the challenges and lessons learned from working with GLSP and developing several GLSP-based modeling tools. The authors present a reusable reference architecture designed to facilitate the development and operation of GLSP-based web modeling tools. The reference architecture is then instantiated using BigUML as a running example. The paper highlights the procedural approach, key lessons learned, and critical reflections on the challenges and opportunities associated with using GLSP. In contrast to Jjodel, which natively supports cloud-based modeling, GLSP-based web modeling tools require the ad hoc implementation of the domain language, adding an additional layer of complexity.

HyperGraphOS [9] is a generalized and semantically rich data store based on hypergraphs. It is designed to support dynamic, complex, and interconnected knowledge representations. Although its architecture is not specifically tailored for modeling tools, it provides a flexible foundation for storing and querying structured knowledge. This flexibility can be relevant for managing model repositories and facilitating semantic reasoning in model-driven engineering contexts. In contrast to Jjodel, which emphasizes a web-based environment for interactive modeling and the rendering of concrete syntax, HyperGraphOS delivers backend infrastructure to manage complex and interconnected data structures. However, it does not address the user-facing aspects of modeling tool composition or visualization.

Wüest et. al [39] present FlexiSketch, a collaborative sketching tool that enables users to create and formalize diagrammatic notations on the fly. Designed for co-located settings, the tool allows participants to sketch simultaneously on tablets while incrementally defining types and relationships, which are synchronized across devices via a shared desktop interface. Its lightweight metamodeling features support the creation of ad hoc notations during early-stage design meetings, fostering engagement and shared understanding. Although FlexiSketch emphasizes flexibility and low-effort formalization, particularly suitable for early ideation and sketch-based workflows, Jjodel focuses on providing a more structured low-code modeling environment with integrated support for syntax co-evolution and reflective governance. Both tools highlight the value of bridging informal modeling with formal representation, although they cater to different usage contexts and modeling needs.

GenMyModel[21] is a cloud-based modeling platform that provides a collaborative environment to create custom UML, BPMN, and diagrams. It features real-time editing, version control, and model sharing, making it suitable for distributed teams and large-scale enterprise projects. However, while

[17] https://projects.eclipse.org/projects/modeling.gmp.

[18] https://eclipse.dev/epsilon/doc/eugenia/.

[19] https://eclipse.dev/sirius/.

[20] https://sprotty.org/.

[21] https://www.genmymodel.com/.

it allows for extensibility through custom profiles, it lacks mechanisms for defining domain-specific syntax or supporting the dynamic co-evolution of metamodels. Furthermore, in 2024, GenMyModel no longer offers a free tier for new users, which can limit its accessibility for educational and exploratory purposes.

Golra et al. [17] propose an interactive and agile methodology to develop graphical DSLs, grounded in the concept of free modeling. Their approach allows users to define both models and metamodels simultaneously, with loosely coupled conceptual and representational elements. Implemented through the Openflexo Free Modeling Editor [3], the environment supports dynamic interaction in four modeling spaces: concept model, conceptual metamodel, representation model, and representation metamodel. Although Jjodel shares a similar goal of enabling flexible modeling (e.g., Jjodel supports the use of untyped objects during modeling, allowing users to progressively refine model elements without enforcing early formalization), it focuses on ease of use through a structured low-code environment that simplifies the definition of graphical concrete syntaxes.

Many of the platforms surveyed possess industrial-grade or commercial-technology readiness levels (TRL), making them well suited for high-stakes, large-scale projects. In contrast, Jjodel remains an academic endeavor, designed to explore and adapt the principles and solutions commonly found in large-budget projects and low-code platforms. Translating these into accessible and innovative approaches, Jjodel makes a unique contribution to the field. However, this academic focus also highlights its limitations in directly competing with fully commercialized solutions, particularly in terms of scalability, robustness, and enterprise-level support. However, the initial adoption of Jjodel in academic settings has yielded promising results. Although a systematic evaluation of its impact on teaching and learning outcomes has not yet been conducted, preliminary feedback has been highly encouraging. Students have demonstrated significant engagement and understanding, even in the absence of extensive learning resources at the time of the course[22]. These early findings indicate that Jjodel has the potential to become a valuable educational tool, particularly in model-driven engineering courses.

7 Conclusions and future work

In this article, we present Jjodel, a cloud-based reflective modeling platform designed to address the challenges of cognitive complexity and usability of MDE. By emphasizing a low-code user-friendly approach and integrating advanced features such as syntax viewpoints and real-time validation, Jjodel bridges the gap between conceptual research and practical application in MDE. Through a case study and a comprehensive analysis of its architecture and functionalities, we demonstrate how Jjodel empowers educators, professionals, and researchers to efficiently create, refine, and adapt domain-specific languages and models.

A core objective of Jjodel is to simplify the teaching of MDE concepts by providing an interactive and intuitive environment. Instructors often face significant challenges in adopting existing tools, which are frequently based on outdated technology stacks, and struggle to align them with modern teaching practices. By integrating familiar front-end technologies, such as JSX templates, Jjodel lowers the entry barrier, making it more accessible to students and allowing MDE to be effectively introduced at the undergraduate level.

Key innovations, including modular syntax viewpoints, position Jjodel as a promising contribution to the advancement of MDE tools.

These features are expected to reduce cognitive barriers and improve flexibility and scalability, potentially making Jjodel a versatile tool for a wide range of use cases.

The development of Jjodel provided critical information on the trade-offs and challenges inherent in the building of modern MDE tools. Implementing advanced features, such as undo/redo functionality, highlighted the need for robust state management and synchronization features, especially in collaborative contexts where ambiguity can arise around whether to revert individual or shared operations. Although a full explanation of the mechanisms is beyond the scope of this paper, we plan to detail in future work the design of the Jjodel Operation Recorder, a component responsible for tracking user actions, which also supports the emerging collaborative features of the platform. Furthermore, the adoption of the React unidirectional data flow architecture enabled powerful functionalities such as semantic zooming and dynamic syntax customization, but also presented scalability challenges when applied to large-scale models. These experiences also identified the importance of interdisciplinary approaches, such as taking advantage of insights from related fields like online gaming, to tackle complex design challenges.

Looking ahead, Jjodel's development will focus on addressing scalability for industry-scale models by adopting advanced optimization techniques from the broader communities of MDE and distributed systems. Key areas for future work include the following challenges:

- Enhance rendering and synchronization mechanisms to efficiently support larger and more complex models.
- Advancing automation for metamodel and model co-evolution to accommodate a broader range of use cases and minimize manual intervention.

[22] For more details on the survey results, see https://www.jjodel.io/student-survey/.

- Developing sophisticated conflict resolution mechanisms and version control systems tailored for distributed teams.
- Exploring integration with existing MDE platforms and tools, such as the Epsilon Playground[23], to enable functionalities such as the execution of model-to-model transformations directly within the environment of Jjodel.
- Consolidating ongoing experiments with large language model (LLM) agents to further enhance modeling capabilities.
- Expanding the Jjodel community by releasing open source components, encouraging contributions and establishing an organization dedicated to securing funding and ensuring the sustainability of the project over the long term.
- Investigating collaborative modeling features by supporting synchronous and asynchronous interactions among distributed users, with particular attention to conflict prevention and visual coordination mechanisms. Initial prototypes are being developed, and this direction will be prioritized in upcoming iterations.
- Conducting qualitative and quantitative evaluations with end users, especially students in MDE courses, to assess usability, learning impact, and modeling effectiveness. These experiments, currently in progress[24], will form the basis of a dedicated empirical study.

While Jjodel is currently more aligned with academic experimentation and prototyping, we are actively working to improve its Technology Readiness Level (TRL) [28] to better meet the needs of industrial users. Enhancing robustness, performance, and integration capabilities remains a key ongoing challenge and focus of our future work. By continuing to evolve and address these challenges, Jjodel aims to establish itself as a cornerstone tool for MDE, serving both as a practical resource for immediate application and as a foundation for innovation in the field. Its commitment to reducing complexity and supporting adaptability is designed to remain relevant as MDE practices continue to evolve.

7.1 A roadmap for the future

The lessons learned during the development of Jjodel provide a roadmap for the future of MDE platforms. Reducing cognitive complexity, fostering collaboration, and supporting adaptability are essential to ensure widespread adoption of MDE practices. These principles make Jjodel a valuable resource for practitioners today and a foundation for future innovation.

[23] https://eclipse.dev/epsilon/playground/.

[24] https://www.jjodel.io/student-survey/.

However, sustaining such advances requires more than technical expertise: it also demands robust community support and funding. Unlike in the early 2000s, when national and international research programs funded tool development, today's funding landscape is less aligned with such objectives. This shift raises important questions about the responsibility for developing and maintaining high-quality modeling platforms.

Although opinions within the MDE community vary, we advocate a shared commitment to advancing both foundational theories and practical solutions. High-quality tools are essential to demonstrate the effectiveness of MDE and ensure its continued relevance in various domains. By working together, the MDE community can build platforms that empower users, enhance education, and drive innovation across industries.

Funding Open access funding provided by Università degli Studi dell'Aquila within the CRUI-CARE Agreement.

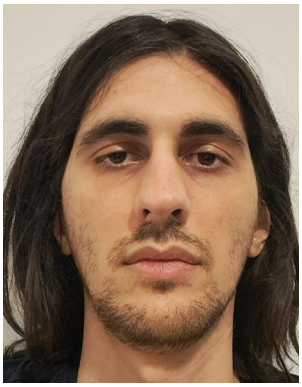
References

1. Abrahão, S., Bordeleau, F., Cheng, B., Kokaly, S., Paige, R., Stöerrle, H., Whittle, J.: User experience for model-driven engineering: challenges and future directions. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 229–236. IEEE (2017)
2. Alfraihi, H., Lano, K.: Trends and insights into the use of model-driven engineering: a survey. In: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 286–295 (2023)
3. Bach, J.-C., Beugnard, A., Champeau, J., Dagnat, F., Guérin, S., Martínez, S.: 10 years of model federation with openflexo: challenges and lessons learned. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, pp. 25–36 (2024)
4. Bork, D., Langer, P., Ortmayr, T.: A vision for flexible glsp-based web modeling tools. In: IFIP Working Conference on The Practice of Enterprise Modeling, pp. 109–124. Springer (2023)
5. Brambilla, M., Cabot, J., Wimmer., M.: Model-Driven Software Engineering in Practice: Second Edition. Morgan & Claypool Publishers, 2nd edition (2017)
6. Brooks, F.P., Bullet, N.S.: Essence and accidents of software engineering. IEEE Comput. **20**(4), 10–19 (1987)
7. Bucchiarone, A., Rocco, J.D., Vincenzo, D.D., Pierantonio, A.: From ocl to jsx: Declarative constraint modeling in modern saas

tools. In: Proceedings of the OCL 2025 Workshop co-located with STAF 2025, Koblenz, Germany (2025)
8. Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A Definitive Guide, pp. 58–90. Springer, Berlin, Heidelberg (2012)
9. Ceravola, A., Joublin, F.: Hypergraphos: a modern meta-operating system for the scientic and engineering domains. arXiv preprint arXiv: 2412.10487 (2024)
10. Céret, E., Calvary, G., Dupuy-Chessa, S.: Flexibility in mde for scaling up from simple applications to real case studies: illustration on a nuclear power plant. In: Proceedings of the 25th Conference on l'Interaction Homme-Machine, IHM '13, pp. 33–42, New York, NY, USA. Association for Computing Machinery (2013)
11. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 2008 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 222–231. IEEE (2008)
12. De Carlo, G., Langer, P., Bork, D.: Advanced visualization and interaction in glsp-based web modeling: realizing semantic zoom and off-screen elements. In: Proceedings of the ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems, MODELS '22, page 221–231, New York, NY, USA., Association for Computing Machinery (2022)
13. Di Rocco, J., Di Ruscio, D., Di Salle, A., Di Vincenzo, D., Pierantonio, A., Tinella, G.: Jjodel: a reflective cloud-based modeling framework. In: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 55–59. IEEE (2023)
14. Di Vincenzo, D., Di Rocco, J., Di Ruscio, D., Pierantonio, A.: Enhancing syntax expressiveness in domain-specific modelling. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 586–594. IEEE (2021)
15. Frisch, M., Dachselt, R., Brückmann, T.: Towards seamless semantic zooming techniques for uml diagrams. In: Proceedings of the 4th ACM Symposium on Software Visualization, pp. 207–208 (2008)
16. Giraudet, T., Bats, M., Blouin, A., Combemale, B., David, P.-C.: Sirius web: Insights in language workbenches—an experience report. J. Object Technol. **23**(1), 1 (2024)
17. Golra, F.R., Beugnard, A., Dagnat, F., Guerin, S., Guychard, C.: 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 '16, pp. 24–34, New York, NY, USA. Association for Computing Machinery (2016)
18. Heidegger, M.: Being and time. Translated by john macquarrie and edward robinson. First English Edition (1927)
19. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope: automating coupled evolution of metamodels and models. In: Drossopoulou, S. (ed.) ECOOP 2009: Object-Oriented Programming, pp. 52–76. Springer, Berlin, Heidelberg (2009)
20. Hili, N.: A Metamodeling framework for promoting flexibility and creativity over strict model conformance. In: Proceedings of the 2nd Workshop on Flexible Model Driven Engineering (FlexMDE 2016), volume 1694 of Flexible Model Driven Engineering, pp. 2–11, Saint-Malo, France, Oct. 2016. Davide Di Ruscio and Juan de Lara and Alfonso Pierantonio, CEUR-WS
21. Hutchinson, J., Whittle, J., Rouncefield, M.: Model-driven engineering practices in industry: social, organizational and managerial factors that lead to success or failure. Sci. Comput. Program. **89**, 144–161 (2014)
22. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment, pp. 109–129. Springer, Berlin, Heidelberg (2013)
23. Kelly, S., Tolvanen, J.-P.: Domain-specific modeling: enabling full code generation. Cutter IT J. **18**(5), 28–34 (2005)
24. Kienzle, J., Zschaler, S., Barnett, W., Sağlam, T., Bucchiarone, A., Abrahão, S., Syriani, E., Kolovos, D., Lethbridge, T., Mustafiz, S., Meacham, S.: Requirements for modelling tools for teaching. Softw. Syst. Model. **23**(5), 1055–1073 (2024)
25. Lafontant, L.-E., Syriani, E.: Gentleman: a light-weight web-based projectional editor generator. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20, New York, NY, USA. Association for Computing Machinery (2020)
26. Liebel, G., Badreddin, O., Heldal, R.: Model driven software engineering in education: a multi-case study on perception of tools and uml. In: 2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T), pp. 124–133 (2017)
27. López-Fernández, J.J., Cuadrado, J.S., Guerra, E., Lara, J.: Example-driven meta-model development. Softw. Syst. Model. **14**(4), 1323–1347 (2015)
28. Mankins, J.C. et al.: Technology readiness levels. White Paper, April, 6(1995):1995 (1995)
29. Metin, H., Bork, D.: A reference architecture for the development of GLSP-based web modeling tools. Softw. Syst. Model. (2025). https://doi.org/10.1007/s10270-024-01257-y
30. Piaget, J.: The Origins of Intelligence in Children. International Universities Press, New York. Translated by Margaret Cook (1952)
31. Pierantonio, A.: Transparency of tools: beyond usability in modeling tools. In: Keynote at the 13th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2025). https://modelsward.scitevents.org/KeynoteSpeakers.aspx?y=2025 (2025)
32. Pirolli, P., Card, S.K., Van Der Wege, M.M.: Visual information foraging in a focus+ context visualization. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 506–513 (2001)
33. Spillers, F.: Progressive disclosure: The best interaction design technique. https://www.experiencedynamics.com/progressive-disclosure-the-best-interaction-design-technique/ (2010). Experience Dynamics. Accessed 31 May 2025
34. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering. Management. Wiley, Hoboken (2006)
35. Tolvanen, J.-P., Kelly, S., Di Rocco, J., Pierantonio, A., Tinella, G.: A framework for evaluating tool support for co-evolution of modeling languages, tools and models. In: Software and Systems Modeling, pp. 1–28 (2024)
36. Tolvanen, J.-P., Kelly, S., Di Rocco, J., Pierantonio, A., Tinella, G.: A framework for evaluating tool support for co-evolution of modeling languages, tools and models. Softw. Syst. Model. **24**(2), 311–338 (2025)
37. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial adoption of model-driven engineering: Are the tools really the problem? In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) Model-Driven Engineering Languages and Systems, pp. 1–17. Springer, Berlin, Heidelberg (2013)
38. Widom, J., Ceri, S.: Active Database Systems: Triggers and Rules for Advanced Database Processing (Morgan Kaufmann, 1995)
39. Wüest, D., Seyff, N., Glinz, M.: Flexisketch team: collaborative sketching and notation creation on the fly. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 685–688. IEEE (2015)

Antonio Bucchiarone is currently an Associate Professor at the University of L'Aquila and a Visiting Fellow at Fondazione Bruno Kessler (FBK) in Trento, Italy. He specializes in adaptive systems, motivational digital systems using gamification, and the design and development of educational tools. His research focuses on models and tools for adaptive systems, modeling and development languages for motivational digital systems, and educational tools supported by and based on Model-Driven Engineering. Antonio Bucchiaronehas contributed to numerous national and European R&D projects, including the Erasmus+ project ENCORE. He is a Senior Member of IEEE and also serves as an Associate Editor for several journals, including IEEE Software, Journal of Object Technology (JOT), IEEE Transactions on Education, IEEE Transactions on Games, IEEE Transactions on Intelligent Transportation Systems (T-ITS), and IEEE Technology and Society.

Juri Di Rocco is a tenure-track assistant professor at the Department of Information Engineering, Computer Science, and Mathematics of the University of L'Aquila. He is interested in all aspects of software language engineering. His main research interests focus on various aspects of Model-Driven Engineering (MDE), including domain-specific modeling languages, recommender systems for MDE, modeling repositories, and mining techniques. He is currently also working on AI-driven software engineering, empirical software engineering methodologies, and collaborative modeling platforms. He serves and has served on the organization and program committees of more than 50 international events, including MODELS, STAF, ICSE, and SANER. Web: https://jdirocco.github.io.



Damiano Di Vincenzo graduated in Computer Science with a thesis on encoding the Ecore and EMF frameworks within the Angular/Node.js ecosystem. He holds a Master's degree in Web Technologies. Since 2020, he has been the principal designer of the Jjodel modeling platform and currently serves as Co-Chief Architect of the overall project. His research and development work has led to several publications on topics related to Jjodel, covering language design, platform architecture, and model-driven engineering.

Alfonso Pierantonio is a Professor of Software Engineering at the University of L'Aquila, Italy. His expertise centers on Model-Driven and Language Engineering, particularly co-evolution techniques, consistency management, and tool development. He has published more than 190 scientific articles and has been instrumental in organizing major international conferences, including MoDELS and STAF. Alfonso serves as Editor-in-Chief of the Journal of Object Technology and sits on the editorial and advisory boards of Software and System Modeling and Science of Computer Programming. He has chaired several events, ECMFA 2018 (PC Chair), STAF 2015, and MoDELS 2023 (General Chair), and is a Steering Committee member for ACM/IEEE MoDELS and other key initiatives. He co-leads multiple research and industrial initiatives. Recently, he launched the Jjodel project (https://jjodel.io), designed to integrate state-of-the-art front-end technologies into next-generation model-driven platforms.