From OCL to JSX: declarative constraint modeling in modern SaaS tools

Antonio Bucchiarone[*,†], Juri Di Rocco[†], Damiano Di Vincenzo[†] and Alfonso Pierantonio[†]

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

Abstract

The rise of Node.js in 2010, followed by frameworks like Angular, React, and Vue.js, has accelerated the growth of low-code development platforms. These platforms harness modern UIX paradigms, component-based architectures, and the SaaS model to enable non-experts to build software. The widespread adoption of single-page applications (SPAs), driven by these frameworks, has shaped low-code tools to deliver responsive, client-side experiences. In parallel, many modeling platforms have moved to the cloud, adopting either server-centric architectures (e.g., GSLP) or client-side intelligence via SPA frameworks, anchoring core components in JavaScript or TypeScript. Within this context, `OCL.js`, a JavaScript-based implementation of the Object Constraint Language, offers a web-aligned approach to model validation, yet faces challenges such as partial standard coverage, limited adoption, and weak integration with modern front-end toolchains.

In this paper, we explore `JSX`—a declarative, functional subset of JavaScript/TypeScript used in the React ecosystem - as an alternative to constraint expression in SaaS-based modeling environments. Its component-oriented structure supports inductive definitions for syntax, code generation, and querying. Through empirical evaluation, we compare `JSX`-based constraints with `OCL.js` across representative modeling scenarios. Results show `JSX` provides broader expressiveness and better fits front-end-first architectures, indicating a promising path for constraint specification in modern modeling tools.

Keywords

OCL, JSX, Modeling, SaaS, Low-Code, Jjodel, Model Validation, Declarative Constraint Specification

1. Introduction

Tools play a pivotal role in the practice and pedagogy of Model-Driven Engineering (MDE) [1, 2]. They support the specification, manipulation, and analysis of models, providing the scaffolding upon which abstractions can be formalized and operationalized. However, despite the increasing relevance of MDE in contemporary software engineering, many of its foundational tools are still rooted in outdated technology stacks. In particular, prominent platforms such as JetBrains MPS[1] and the Eclipse Modeling Framework[2] (EMF) were first released prior to 2010, while MetaEdit+[3], one of the first tools of its kind, dates back to 1995. These tools often rely on legacy desktop environments, imperative user interfaces, and monolithic architectures that lack alignment with modern user experience (UIX) paradigms. As a result, they impose a considerable cognitive and technical load on users, particularly those without formal software development background [3].

In recent years, the emergence of *single-page applications (SPAs)* has reshaped the landscape of software development, especially within *low-code development platforms* [4]. SPAs provide fluid, dynamic

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10-13, 2025

*Corresponding author.
†These authors contributed equally.
antonio.bucchiarone@univaq.it (A. Bucchiarone); juri.dirocco@univaq.it (J. D. Rocco); damiano.divincenzo@studenti.univaq.it (D. D. Vincenzo); alfonso.pierantonio@univaq.it (A. Pierantonio)
https://www.disim.univaq.it/AntonioBucchiarone (A. Bucchiarone); https://www.disim.univaq.it/JuriDiRocco (J. D. Rocco); https://www.disim.univaq.it/AlfonsoPierantonio (A. Pierantonio)
0000-0003-1154-1382 (A. Bucchiarone); 0000-0002-7909-3902 (J. D. Rocco); 0009-0004-7717-0574 (D. D. Vincenzo); 0000-0002-5231-3952 (A. Pierantonio)

[1]https://www.jetbrains.com/mps/
[2]https://projects.eclipse.org/projects/modeling.emf.emf
[3]https://www.metacase.com/products.html

interfaces that run entirely in the browser, minimizing page reloads and latency. This shift has enabled highly responsive and accessible platforms, significantly reducing the entry barrier for end users and promoting the participation of domain experts, non-programmers, and citizen developers [4]. Frameworks such as React[4], Angular[5], and Vue.js[6], often in combination with Node.js[7], have played a critical role in this transformation, with React gaining particular prominence due to its robust ecosystem, component-based architecture, and strong developer adoption [5]. According to the 2024 Stack Overflow Developer Survey[8], React, Angular, and Vue.js are used together by approximately 77. 6% professional developers, underscoring their dominant position in the modern front-end development landscape.

In response to these trends, several modern modeling platforms—such as *BESSER* [6] and *Jjodel* [7, 8, 9]—have been engineered as single-page applications (SPAs), fully embracing the front-end-first paradigm. This design change enables them to provide rich, responsive user interfaces for tasks such as model editing, validation, and visualization, all within the browser. A cornerstone of many React-based SPAs is JSX [10], a hybrid syntax that seamlessly integrates XML-based HTML with declarative and functional JavaScript constructs. JSX empowers developers to dynamically compose user interfaces and interact fluently with the state of the application [5], making it a natural fit for model-driven environments built on modern web stacks.

In the context of model-driven environments, the state of the application often encapsulates a variety of modeling artifacts, such as metamodels, user-defined models, and integrity constraints. Within this setting, JSX expressions can be repurposed not only for UI composition but also as expressive mechanisms for *querying* and *validating* these model elements. This dual functionality opens promising avenues for specifying constraints directly within the application logic, bridging the gap between front-end technologies and model semantics. This naturally leads to a fundamental question.

> *Can JSX serve as a viable alternative to traditional constraint languages such as the Object Constraint Language (OCL)?*

Although OCL remains the standard for expressing model-level constraints in MDE [11], its integration into modern Web environments is hindered by limited support. Tools such as *OCL.js*[9] attempt to bridge this gap by providing a JavaScript-based implementation of OCL, but these tools often suffer from partial standard coverage, weak community support, and fragile maintenance models, which rely on individual developers or small research groups. In contrast, JSX is natively supported in the React ecosystem and is already familiar to many software engineers and students, making it an attractive, low-overhead candidate for constraint specification in React-based modeling platforms.

In this article, we investigate the feasibility and expressiveness of JSX as a constraint language in SPA-based modeling tools. We systematically compare JSX-based expressions with OCL.js through a series of representative modeling scenarios. Our goal is to assess not only the functional equivalence of these approaches but also their ergonomics, maintainability, and alignment with contemporary modeling practices [12]. The experiments have been conducted by assuming that the artifacts are stored in the application state by means of the Jjodel Object Model (JjOM)[10] encoding.

2. Background

2.1. OCL in Modeling: Adaptation to SaaS

OCL is a formal declarative language designed to specify constraints and queries within modeling environments. Originally introduced as part of the Unified Modeling Language (UML) specification,

[4]https://reactjs.org
[5]https://angular.io
[6]https://vuejs.org
[7]https://nodejs.org
[8]https://survey.stackoverflow.co/2024/technology#most-popular-technologies-webframe-prof
[9]https://github.com/SteKoe/ocl.js
[10]https://www.jjodel.io/jjodel-object-model-jjom/

OCL has become a foundational element in MDE, providing the means to define precise, unambiguous semantics that complement visual modeling notation. It enables developers and modelers to specify a wide range of constraints, including invariants, preconditions, postconditions, and derived attributes, thus enforcing the integrity and correctness of models. Unlike graphical representations, which often lack the accuracy required for formal validation, OCL introduces a textual layer that supports model verification, validation, and transformation across a broad spectrum of modeling frameworks. One of the key strengths of OCL lies in its powerful collection and navigation operations, which allow modelers to traverse associations, access related elements, and perform complex computations on structured model elements. These capabilities make OCL an indispensable tool in the development of correct-by-construction software systems, especially in safety-critical or rigorously specified domains. OCL is widely used in metamodeling environments like Eclipse EMF[11], UML, and other MOF-based platforms to enhance the expressiveness and precision of metamodels. It allows defining structural and behavioral constraints on metamodel elements, supporting domain-specific languages (DSLs) and model-driven development (MDD) workflows. In MDE, the OCL is key to specifying transformation rules and operational semantics, notably in frameworks such as ATL [13], QVT [14], and Acceleo[12]. It is also used to query and ensure semantic correctness throughout the development lifecycle. By adding a formal textual layer to graphical models, OCL enables precise validation, enforces domain rules, and supports traceable, verifiable artifacts - contributing to robust and maintainable model-driven systems.

Low-code platforms have reshaped software development by enabling rapid application creation through visual modeling and reusable components. Most adopt the SaaS model [15], leveraging native cloud infrastructures for scalability, collaboration, and integration. This SaaS shift also affects modeling tools, traditionally desktop-based, which are now evolving toward cloud-based solutions that support real-time collaboration, versioning, and validation, while removing installation and maintenance overhead. However, OCL remains constrained in web contexts due to limited accessibility and poor integration with JavaScript toolchains. Although MDE is central to defining invariants and behavioral rules, its desktop-centric design hinders adoption in modern SaaS environments.

Addressing this gap requires a JavaScript-based OCL implementation that runs natively in the browser and aligns with declarative, reactive, and event-driven web paradigms. Such an approach would support real-time, client-side constraint validation, offering a more seamless, responsive, and user-friendly experience for both low-code and model-driven web applications.

2.2. OCL.js

OCL.js[13] is a JavaScript implementation of OCL for dynamic constraint validation on standard JavaScript objects. It allows a declarative definition of constraints, promoting separation between business logic and validation. As shown in Listing 1, after initializing the engine (lines 1–2), constraints are defined on model elements: line 3 ensures that an order contains at least one item; line 4 checks that the total price matches the sum of item prices; line 5 enforces that each item has a positive price. In line 6, an empty order is evaluated, returning `false` due to constraint violations. Thus, OCL.js supports robust, declarative rule validation within modern web-based modeling environments.

Listing 1: Fragment of the OCL.js -Engine and constraints evaluation.

```
1 const { OclEngine } = require('ocl.js');
2 const oclEngine = new OclEngine();
3 oclEngine.addOclExpression(‘context Order inv: self.items->size() > 0‘);
4 oclEngine.addOclExpression(‘context Order inv: self.totalPrice = self.items->collect(i | i.
      price)->sum()‘);
5 oclEngine.addOclExpression(‘context Item inv: self.price > 0‘);
6 const order = { id: "00001", totalPrice : 0, items:[] };
7 console.log(oclEngine.evaluate(order)); // Returns false (all the constraints are unsadisfied)
```

[11]https://projects.eclipse.org/projects/modeling.mdt.ocl
[12]https://eclipse.dev/acceleo/
[13]https://ocl.stekoe.de/

In addition to supporting standard invariants, OCL.js also enables the definition of preconditions and postconditions, which are critical for maintaining transactional integrity in software systems. For instance, in the context of a banking application, a withdrawal operation should only proceed if the account balance remains non-negative after the transaction is applied (see Listing 2). By enforcing such pre- and postconditions, OCL.js helps ensure that critical domain constraints are upheld during function execution. This not only prevents unintended state transitions but also enhances the reliability and correctness of operations in model-driven and data-intensive applications.

Listing 2: Fragment of the OCL.js - Pre and postconditions.

```
oclEngine.addOclExpression(‘
    context Account::withdraw(amount: Number): Boolean
    pre: self.balance >= amount
    post: self.balance = self.balance@pre - amount
‘);
```

Additionally, OCL.js supports the definition of derived attributes, allowing computed properties to be automatically inferred from existing model data. For example, in a payroll system, an employee's net salary can be derived from their gross salary and applicable tax deductions, ensuring consistency and reducing redundancy in the model (see Listing 3). This approach promotes consistency and eliminates redundant computations, thereby reducing the risk of errors in critical domains such as financial calculations.

Listing 3: Fragment of the OCL.js - Derived attributes.

```
oclEngine.addOclExpression(‘
    context Employee::netSalary: Number derive: self.grossSalary - self.tax
‘);
```

Designed to be fully JavaScript native, OCL.js integrates into both front-end and back-end environments. In web applications, it validates client-side forms to ensure user input adheres to business rules, while on the server, it enforces data integrity before database persistence—reducing reliance on database constraints. OCL.js is ideal for graph-based modeling tools and UML editors, enabling real-time validation in domain-specific modeling. Its compatibility with modern JavaScript frameworks makes it well suited for enterprise applications, transactional systems, and model-driven platforms.

By offering a declarative, expressive, and modular approach to constraint specification, OCL.js contributes to building more reliable, verifiable, and maintainable software systems.

2.3. Limitations of OCL.js

Although OCL.js offers a useful set of features and provides a reasonably comprehensive interpretation of the OCL standard, it reveals several critical limitations, particularly when evaluated in the context of adaptive and reflective platforms such as Jjodel. Initially, Jjodel adopted OCL.js to support model querying and constraint checking. However, as the platform evolved, it became evident that OCL.js could not accommodate the dynamic, introspective requirements of our runtime ecosystem. In response, we developed a dedicated library, JjOM, which natively supports querying and navigation of the objects models of Jjodel. The following discussion outlines the key limitations of OCL.js that motivated the design and implementation of JjOM.

One of the most fundamental limitations of OCL.js is its incomplete alignment with the official OCL specification, as explicitly acknowledged by the maintainers on the project website[14]:

> *This library does not claim to be fully compliant with the OMG OCL definition and might have slight differences.*

The disclaimer signals the potential for partial feature support, non-standard behaviors, and even semantic inconsistencies, particularly when translating formal OCL specifications into JavaScript environments. Such discrepancies introduce uncertainty into constraint evaluations and diminish

[14]https://ocl.stekoe.de/#usage

confidence in the accuracy of model validation. Another structural limitation of OCL.js lies in its static expression model. Constraints must be defined and parsed ahead of time, making the library poorly suited for dynamic modeling environments like Jjodel, where models evolve at runtime and constraint definitions may need to be generated or modified on-the-fly. The absence of support for dynamic constraint generation reduces flexibility, hampers maintainability, and limits scalability, particularly in iterative development workflows or systems that involve runtime transformation or user-defined modeling constructs. OCL.js lacks reflective capabilities, which prevents it from adapting to changes in the underlying metamodel, an essential feature for self-adaptive platforms like Jjodel. Without native reflection, developers must rely on manual updates or external tools, making integration error-prone and fragile. It also struggles with expressiveness and manageability for complex constraints. Deeply nested logic, chained operations, and advanced collection handling become hard to author and debug. The absence of dedicated tooling or intelligent feedback further reduces usability and increases the risk of validation errors in practice.

Beyond technical limitations, OCL.js faces challenges in sustainability and adoption. Although its GitHub repository[15] shows 762 commits since 2016, development has stagnated. Since September 2021, only 41 of the 269 commits were made by human contributors; the rest were automated updates, indicating minimal active maintenance. Community adoption is also low: a search for `@stekoe/ocl.js` yields just 26 usage references across 7 public repositories, several of which are inactive. This limited traction raises concerns about OCL.js as a sustainable foundation for reflective modeling platforms. These issues—partial standard coverage, limited adaptability, weak tooling, and declining support—motivate the creation of JjOM. Unlike OCL.js, JjOM supports runtime introspection, dynamic constraint evaluation, and seamless integration with Jjodel's object model, offering a more robust and modern alternative.

3. SaaS-Based Modeling: JSX and the Jjodel Object Model (JjOM)

The emergence of SaaS-based modeling environments has prompted a rethinking of how constraint definition, model navigation, and runtime interaction are handled in low-code browser-native platforms. In Jjodel, the use of React, Redux[16] and JSX is not limited to the presentation layer; it extends to the fundamental mechanics of modeling and constraint specification. At the heart of this architecture lies the Jjodel Object Model (JjOM), a runtime structure that integrates model data, visualization, and interaction through a modern, declarative web technology stack.

3.1. React and JSX in Jjodel

React serves as the backbone of the Jjodel front-end, while JSX, a declarative XML-like syntax embedded in JavaScript/TypeScript[17], acts as both a UI definition language and a mechanism to interact with model data. In this setting, JSX can be leveraged to express modeling constructs, navigation logic, and even runtime constraints in a style that is readable, composable, and tightly integrated with modern development workflows. JSX components in Jjodel are inductive building blocks that represent both visual syntax and semantic structure. Each component corresponds to a metamodel concept or model element, and its composition mirrors the hierarchical structure of the modeling languages. These components can be used not only to render visual representations, but also to perform validation and evaluation tasks, enabling declarative logic to be embedded directly into the modeling interface.

Similarly to OCL, the JSX expressions in Jjodel are based on a core set of declarative connectives inherited from JavaScript (see the first table in [16]). These expressions operate in the structured application state, which encodes both models and metamodels according to the internal schema of the JjOM as described in the next section. Every time the JSX changes, the view is transpiled at run-time into a parametrized component, having all contextual variables (like data) as parameters. This ensures it

[15]https://github.com/SteKoe/ocl.js
[16]https://redux.js.org
[17]The JSX version that uses TypeScript is usually called TSX.

is not required to transpile the component again if any variable changes, but it is sufficient to call it as a function to update the GUI. For further optimization, the view designer can define a list of dependencies that will be displayed in the GUI, and have the component update only if one of those has changed.

3.2. The Jjodel Object Model (JjOM)

The JjOM provides a structured API and a runtime infrastructure to manage modeling artifacts. It encapsulates both metamodels and models and divides the data into three interconnected submodels.

- *Data Submodel*: Represents the abstract structure, including classes (DClass), attributes (DAttribute), references (DReference), objects (DObject) and values (DValue). This submodel forms the core of the Jjodel meta-metamodel and is directly manipulable through the JjOM API.
- *Node Submodel*: Captures layout information, including spatial positioning and relationships between visual elements. It also stores validation feedback and serves as a semantic overlay for layout-driven behavior.
- *View Submodel*: Handles the concrete syntax and graphical representation of model elements, synchronizing them with the underlying data and node submodels. It also manages constraints and policies that govern model and layout updates, as well as events related to executable models.

These submodels ensure consistent handling of abstract models, layout metadata, and visual syntax within a unified native representation of the browser. JjOM offers a rich JavaScript API for querying, manipulating, and validating modeling artifacts at runtime, as detailed in [16]. Through this interface, developers can access and modify class structures, instantiate and navigate object references, define and evaluate constraints using JSX expressions, run queries and transformations throughout the model and perform dynamic validation with `validateModel()`.

For example, the API allows retrieving all attributes of a metaclass (`class.attributes`), adding new model instances (`addObject()`), and performing filtered queries using JSX-based expressions (`executeQuery()`). An extensive (but not exhaustive) presentation of the object model is given in [16].

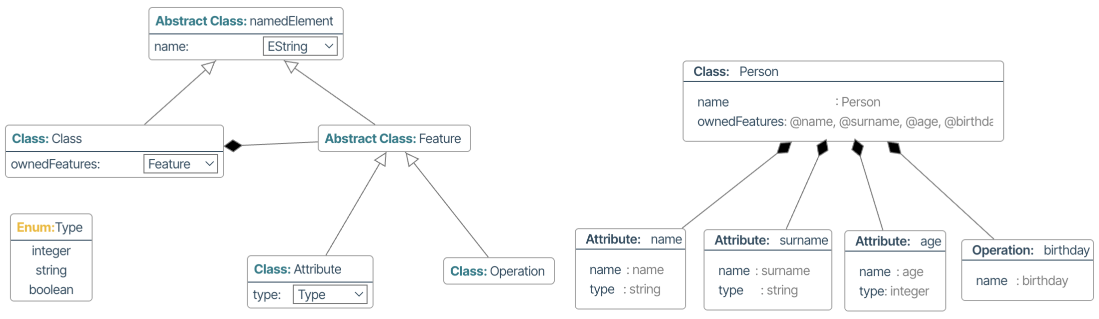
3.3. JSX for Model Navigation

In Jjodel, JSX is elevated from a user interface markup language to a fully functional constraint specification mechanism. JSX expressions are used to query and manipulate model artifacts, replacing OCL. An interesting aspect of JSX is that its familiar syntax and declarative structure lower the learning curve, especially for developers and students already proficient in JavaScript. For instance, a JSX expression such as:

```
data.$ownedAttributes.values.map(a => a.name)
```

retrieves the names of all attributes of the selected model element. Here, `data` serves a role analogous to `self` in OCL, acting as a context pointer to query the active model element. In this expression, the $ prefix is an operator used to invoke reflective access methods provided by the Jjodel runtime. It allows dynamic navigation of model elements and attributes by resolving references at run-time through the Redux-based storage. The `.values` accessor retrieves the values from a model collection (in case of a single value the operator is `.value`). Finally, the `map(a => a.name)` construct is the JavaScript array mapping function, which behaves similarly to the collect operation in OCL. Typical `JSX` use cases include accessing and manipulating attribute values, navigating relationships, applying filters or transformations to collections of model elements, executing inline validations directly within the model view, and binding logic to UI events using event-condition-action (ECA) rules.

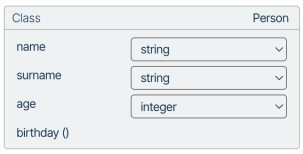
These features enable JSX to seamlessly integrate syntax definition, interaction logic, and constraint validation within a single expressive layer. To illustrate this capability, consider the metamodel of a simplified UML Class Diagram shown in Figure 1(a), where a `Class` is composed of one or more `Attribute` and `Operation` elements. For the sake of clarity, certain advanced concepts—such as typed parameters for operations and inter-class relationships—have been intentionally omitted. An example instance of this metamodel is depicted on the right-hand side of the same figure, where the `Person` class contains three attributes: `name`, `surname`, and `age`, together with a single operation `birthday`. The concrete syntax based on JSX for the element `Class`, shown in Figure 2(a), defines how instances of the metamodel element `Class` are rendered and edited. The component `<Input ... />` in line 5 enables projectional editing of the class name, allowing users to directly modify the value of the attribute `name` through the visual interface. The JSX expression on lines 9-11 dynamically selects all elements of `Feature` (i.e. attributes and operations) associated with the class by navigating through



(a) Metamodel. (b) Model.

Figure 1: UML Class Diagram.

```
<div className={'root'}>
    <div className={'header'}>
        <div className={'input-container mx-2'}>
            <b className={'object-name'}>Class</b>
            <Input data={data} field={'name'} hidden={true} autosize={true} />
        </div>
    </div>
    <div className={'body'}>
        {data.ownedFeatures.values.map(feature =>
            <DefaultNode data={feature} />
        )}
    </div>
</div>
```



Class	Person
name	string
surname	string
age	integer
birthday ()	

(a) The `ClassView` template. (b) The `Person` class.

Figure 2: The concrete syntax for `Class` elements.

`data.$ownedFeatures.values`. These features are then rendered using the `<DefaultNode ... />` component, which applies the appropriate view to each element based on its type: specifically `AttributeView` for attributes and `OperationView` for operations[18].

This result is obtained by inductively applying the views corresponding to the runtime types of the instances, making JSX highly expressive and adaptable. For instance, to render only attributes and exclude operations, one could use the following JSX expression:

```
{data.$ownedFeatures.values
    .filter(f => f.instanceof.name === 'Attribute')
    .map(a =>
        <DefaultNode data={a} />
    )
}
```

Listing 4: Fragment of the OCL.js -Engine and constraints evaluation.

In summary, JSX and the Jjodel Object Model (JjOM) offer a powerful and flexible foundation for defining, navigating, and rendering modeling constructs in modern SaaS-based environments. As illustrated in the previous example by integrating concrete syntax and constraint logic, JSX provides a potential alternative to approaches such as OCL.js. In the following section, we present an empirical evaluation that compares OCL.js and JSX in representative modeling scenarios, assessing their expressiveness, conciseness, and practical usability.

4. Empirical Validation

We compare our proposed model navigation and validation library with OCL.js, using representative OCL statements drawn from literature and practice to illustrate equivalent functionality in JjOM and OCL.js. Section 4.1 outlines the evaluation methodology and comparison criteria. Section 4.2 applies these criteria to selected examples, while Section 4.3 introduces the research questions addressed in Section 4.4..

4.1. Evaluation methodology and criteria

The evaluation methodology includes three steps:

[18]While this holds in the current example, it represents an oversimplification, views in Jjodel are applied not by type alone, but to instances that satisfy the associated view predicate. This allows for more flexible and context-sensitive rendering beyond simple type-based dispatch

Dataset Selection: We selected OCL expressions from "The Ultimate OCL Tutorial"[19], which lists 56 operations across strings, numerics, booleans, collections, iterators, and `OclAny`. After reviewing and filtering for clarity and completeness, we retained 31 representative OCL samples. Out of the original 56 operations, we selected 31 by removing duplicates and discarding examples with syntactically incorrect code that did not conform to the grammar. This filtering step ensured the validity and uniqueness of the evaluated operations.

Implementation of Expressions: Two co-authors, both proficient in JavaScript and OCL, independently implemented the samples—one using JjOM, the other using `OCL.js`. Due to the lack of metamodels and models, we limited the comparison to *syntax correctness*.

Measurement and Analysis: Implementations were evaluated using formal criteria. Metrics were averaged across datasets, and for `OCL.js`, only the OCL code was considered (excluding API invocation).

▷ **Conciseness** is evaluated by measuring the number of characters (excluding whitespace) (**COC**). In this comparison, we aim to evaluate JjOM in relation to OCL syntax. We are interested in understanding how the combined use of JjOM with JSX technology can completely eliminate the need for OCL.js within Jjodel. For this reason, we will focus on comparing the effectiveness and conciseness of JjOM code against OCL code, without taking into account the API calls from the OCL.js framework.

▷ **Readability and Understandability** are assessed qualitatively through an expert-based readability assessment (ERA) [17]. We employed a 5-point Likert scale, where 1 indicates poor readability and 5 indicates excellent readability. Each expression was independently evaluated by two experts with practical experience in MDE and proficiency in both OCL and JjOM. To ensure consistency and mitigate individual bias, the experts initially rated each expression separately. In cases where discrepancies emerged, the evaluators engaged in a focused discussion to reach a consensus, after which a single agreed-upon score was assigned. This process ensures both independent judgment and mutual calibration, following best practices in qualitative code assessment [17].

The curated dataset, Jjodel, and OCL code, along with the analyzed evaluation criteria, are reported in a Google Sheet.[20]

4.2. Explanatory Example: Equivalent Expressions and Metrics

In this section, we illustrate how the selected evaluation criteria have been applied to the representative OCL.js statements shown in Listings 1, 2, and 3. In particular, Listing 5 presents the JSX/JjOM expressions that realize the same semantics as the OCL.js example shown in Listing 1. In Jjodel, validation is performed through validation views, where each condition is associated with a dedicated view. When a specified condition is met, the corresponding view is activated, either by displaying an error message or by modifying the graphical representation of the invalid element. The final expression in the list creates an instance of the `Order` class, although the same operation can be carried out alternatively through the graphical user interface. Upon creation, the validation view for `Item` is activated, rendering its associated feedback based on the violated constraint.

```
data.$items.values.length > 0;
data.$totalPrice.value === data.$items.values.reduce((i, a) => a+=i.$price.value, 0);
data.$price.value > 0
data.parent.addObject({$id: "00001", $totalPrice: 0, $items: []}, 'Order');
```

Listing 5: Fragment of the OCL.js -Engine and constraints evaluation.

Replicating Listing 2 in Jjodel is challenging, as neither Jjodel nor JjOM natively supports pre- and post-conditions like OCL or `OCL.js`. While approximations are possible via validation views or event-driven logic, they lack explicit support for `@pre`. Nonetheless, Jjodel's architecture could support such features in future extensions through event history or scoped validation.

Listing 6 shows the JjOM version of Listing 3, using `onDataUpdate` and `UsageDeclarations` to compute derived values—e.g., `netSalary` from `grossSalary` and `tax`—only when relevant properties change. This selective dependency tracking reduces unnecessary recomputation and improves efficiency, especially in large or dynamic models, offering a precise and maintainable reactive approach.

Listing 6: Fragment of the OCL.js -Engine and constraints evaluation.

```
data.$netSalary.value = data.$grossSalary - data.$tax.value; //placed inside a onDataUpdate event of a view.
```

[19] https://modeling-languages.com/ocl-tutorial/

[20] https://docs.google.com/spreadsheets/d/1qtgYhGsg9x90H13YdAw-6gk02zP8typD/edit?usp=sharing&ouid=111608713792271140113&rtpof=true&sd=true

4.3. Research Questions

▷ **RQ$_1$ (Coverage)**: *To what extent does the proposed JjOM library support expressing standard OCL statements commonly used in MDE, as represented by examples from "The Ultimate OCL Tutorial"?*

This question directly addresses the completeness and expressiveness of the JjOM approach compared to OCL, ensuring that it adequately covers typical modeling scenarios found in practical contexts.

▷ **RQ$_2$ (Effectiveness and Usability):** *How does JjOM compare with OCL.js regarding conciseness, structural complexity, readability, and adaptability to metamodel changes when implementing representative model constraints?*

This question specifically targets the metrics outlined in your evaluation criteria, providing a comprehensive comparison of JjOM and OCL.js in terms of practical use and maintainability.

4.4. Results

This section presents the findings from our comparison between JjOM and OCL.js, based on the representative dataset derived from The Ultimate OCL Tutorial (see Section 4.1). Our analysis addresses the two research questions defined in Section 4.3 and is guided by the evaluation criteria previously outlined.

RQ$_1$: Our evaluation confirms that JjOM is capable of expressing the majority of structural OCL constructs represented in the selected dataset. The analysis was conducted on 31 OCL examples extracted and curated from the OCL Tutorial, covering a broad range of core operations, including invariants, derived attributes, and iterator-based collection expressions. As a result, our evaluation of **Expressiveness** is scoped to structural and declarative OCL constructs, which JjOM is designed to support directly. Behavioral contracts fall outside the current capabilities of JjOM and were not evaluated in this study. Among the 31 examples, JjOM was able to fully express 29 (93.5%). We were not able to rewrite two OCL statements because they are using per- and post-conditions. Conversely, OCL.js fails in 6 cases (80.6 of examples), primarily due to its lack of native support for global quantification via `allInstances` and limited facilities for date manipulation, both of which are present in the dataset. In conclusion, JjOM demonstrates broad coverage and practical expressiveness across standard OCL constructs relevant for structural model validation, with only marginal gaps attributable to its current design focus and the lack of behavioral constraint constructs in the dataset.

RQ$_2$: To assess it, we computed metrics across the implemented expressions in both technologies, focusing on conciseness, and qualitative readability using the representative dataset of 31 OCL expressions described in Section 4.1. Regarding **Conciseness**, contrary to what might be expected from a JavaScript-based API, JjOM expressions were, on average, more concise than their OCL.js equivalents. The mean character count for JjOM was 64.8, compared to 74.2 for OCL.js (we calculated the cost of ownership (COC) based on the expression addressed by both approaches). This result suggests that JjOM's idiomatic expression style—despite involving explicit `.value` and `.values` accessor—still enables more compact constraint definitions. In contrast, OCL.js expressions tend to include verbose constructs (e.g., context declarations, full path navigation, and wrapper methods) that contribute to code bloat, especially in complex model traversals.

Readability was assessed by two independent experts using a 5-point Likert scale. After individual scoring and post-review discussion to resolve discrepancies, an agreed score was recorded for each expression. The average score for OCL.js was 4.4, while JjOM scored 4.8, indicating a clear preference for JjOM in terms of perceived clarity and developer friendliness. Experts particularly appreciated JjOM's use of familiar JavaScript idioms and its integration within the reactive view-based architecture of Jjodel. Although OCL.js offers direct mapping to OCL syntax, it was perceived as less accessible and more syntactically rigid.

These results suggest that JjOM offers superior practical usability compared to OCL.js, achieving wider coverage, more concise expressions, and higher readability, all while maintaining comparable structural complexity. For scenarios focused on structural validation in reactive modeling environments, JjOM presents itself as a more modern and developer-friendly alternative.

5. Conclusions and Future Works

The adoption of SPA frameworks like React, Angular, and Vue.js has redefined the architecture of modern software systems, including low-code platforms. In this landscape, modeling tools are shifting toward SaaS-based, browser-native solutions, as exemplified by Jjodel. This evolution demands new approaches to constraint specification beyond traditional, desktop-bound solutions.

This paper addressed a central question:

> *Should developers rely on* `OCL.js`*, which adapts OCL for the web but lacks full standard coverage and broad support, or adopt* `JSX`*, a robust, widely adopted web technology not originally intended for model validation but deeply integrated into the modern web stack?*

Through an empirical study of 31 modeling scenarios, we found that JSX (via JjOM) delivers strong expressiveness and coverage of OCL constructs, with high readability (4.4 vs. 4.0) and effective alignment with modern JavaScript idioms. While some advanced features like `@pre` remain unsupported, workarounds exist, and JSX proves to be a promising alternative to `OCL.js` in SPA-based environments. Future work will address current gaps by extending JjOM with support for state tracking, scoped validation, and a refined API to reduce verbosity while preserving semantic clarity.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-4o to grammar and spelling check.

Acknoledment

This work has been partially supported by the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY).

References

[1] D. Schmidt, Model-driven engineering, IEEE Computer 39 (2006) 41–47.

[2] J. Kienzle, S. Zschaler, W. Barnett, T. Sağlam, A. Bucchiarone, S. Abrahão, E. Syriani, D. Kolovos, T. Lethbridge, S. Mustafiz, S. Meacham, Requirements for modelling tools for teaching, Software and Systems Modeling 23 (2024) 1055–1073.

[3] A. Bucchiarone, J. Cabot, R. F. Paige, A. Pierantonio, Grand challenges in model-driven engineering: an analysis of the state of the research, Software and Systems Modeling 19 (2020) 5–13.

[4] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, M. Wimmer, Correction to: Low-code development and model-driven engineering: Two sides of the same coin?, Software and Systems Modeling 21 (2022) 1687–1687.

[5] A. Banks, E. Porcello, Learning React: functional web development with React and Redux, O'Reilly Media, Inc., 2017.

[6] I. Alfonso, A. Conrardy, J.-S. Sottet, J. Cabot, Building besser: an open-source low-code platform, arXiv preprint arXiv:2405.13620 (2024).

[7] A. Bucchiarone, J. Di Rocco, D. Di Vincenzo, A. Pierantonio, Modeling in jjodel: Bridging complexity and usability in model-driven engineering, arXiv preprint arXiv:2502.09146 (2025).

[8] J. Di Rocco, D. Di Ruscio, A. Di Salle, D. Di Vincenzo, A. Pierantonio, G. Tinella, Jjodel–a reflective cloud-based modeling framework, in: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, 2023, pp. 55–59.

[9] D. Di Vincenzo, J. Di Rocco, D. Di Ruscio, A. Pierantonio, Enhancing syntax expressiveness in domain-specific modelling, in: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, 2021, pp. 586–594.

[10] Meta (Facebook), Introducing jsx, https://reactjs.org/docs/introducing-jsx.html, 2023. Accessed: 2025-04-10.

[11] J. Cabot, M. Gogolla, Object constraint language (ocl): a definitive guide, in: International school on formal methods for the design of computer, communication and software systems, Springer, 2012, pp. 58–90.

[12] I. Alfonso, A. Conrardy, J. Cabot, Towards the interoperability of low-code platforms, arXiv preprint arXiv:2412.05075 (2024).

[13] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, Science of computer programming 72 (2008) 31–39.

[14] W. Bast, M. Belaunde, X. Blanc, K. Duddy, C. Griffin, S. Helsen, M. Lawley, M. Murpree, S. Reddy, S. Sendall, MOF QVT final adopted specification, OMG document ptc/05-11-01 (2005).

[15] V. Choudhary, Software as a service: Implications for investment in software development, in: 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07), IEEE, 2007, pp. 209a–209a.

[16] Jjodel Website, JSX Connectives and JjOM APIs, https://www.jjodel.io/jsx-for-model-navigation/, 2025. Accessed: 2025-04-11.

[17] D. Oliveira, R. Bruno, F. Madeiral, F. Castor, Evaluating code readability and legibility: An examination of human-centric studies, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 348–359.