ReOCL

An Incremental OCL Core for Reactive Web-Based Modeling

Francisco Martínez-Lasaca
Universitat Oberta de Catalunya
Barcelona, Spain
fmartinezlasa@uoc.edu

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

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

Abstract

Constraint languages such as OCL are commonly used to add precision to domain models and meta-models, typically built using UML class diagrams and Ecore, respectively. However, current constraint languages and their tooling are poorly aligned with modern web-based modeling stacks, which demand reactive execution environments that immediately respond to user interactions.

To address this gap, this paper presents ReOCL, a restricted OCL core designed for incremental, reactive constraint validation for the web. ReOCL defines a compilation strategy that maps invariants into reactive signals —like those found in Angular or Preact— to immediately update invariant status upon user changes. The approach relies on delta-based incremental maintenance of collection views, at constant cost per element added or removed, and on transactions that can be committed or rolled back to support @pre handling. We present a formalization of ReOCL and a TypeScript implementation.

We evaluate our approach against adjacent constraint checking web frameworks, showing that ReOCL trades OCL expressiveness for update costs that stay constant, not linear, as models grow, gaining orders of magnitude on large incremental workloads.

CCS Concepts

• **Software and its engineering → Constraint and logic languages**.

Keywords

Reactive OCL, Object Constraint Language, incremental constraint checking, web development

ACM Reference Format:

Francisco Martínez-Lasaca, Esther Guerra, and Juan de Lara. 2026. ReOCL: An Incremental OCL Core for Reactive Web-Based Modeling. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3837062.3839069

1 Introduction

In domain-specific language (DSL) design, meta-models capture structure, but they cannot by themselves fully rule out ill-formed models. To do so, the Object Constraint Language (OCL) [15] is typically leveraged to complement the meta-model with constraints.

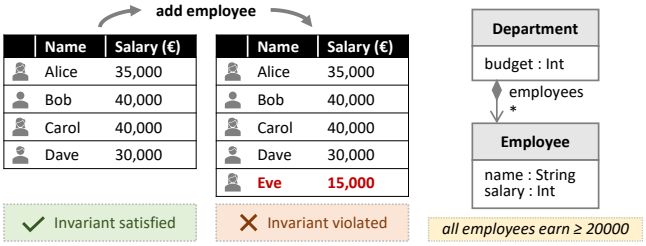


Figure 1: When an employee is added, the invariant updates its running state using only the incoming salary, rather than iterating over all employees.

However, most OCL tooling targets batch validation in desktop-based setups, such as the Eclipse IDE. This execution style does not fit the cloud-based, low-code platforms to which model-driven solutions are increasingly being migrated [19], where users expect immediate feedback after each UI interaction. Consequently, constraint checking should be reactive, with validation states evolving together with the interfaces that display them. Treating OCL invariants as derived computations over model state achieves this goal, avoiding costly re-evaluations, as Figure 1 illustrates.

This paper presents **ReOCL**, a restricted OCL core for TypeScript-based web applications that guarantees constant-time incremental updates for stable, constant-cost predicates. ReOCL defines how invariants compile into reactive signals embeddable in TypeScript frameworks, and provides efficient incremental views for collection operators (e.g., select or forAll) using membership deltas.

The contributions of this paper include:

(1) The ReOCL language and its formalization: reactive semantics for incremental collections, transactional semantics for @pre handling, and a compilation strategy targeting signals.
(2) A TypeScript library implementation.
(3) An evaluation against adjacent web frameworks, analyzing feature coverage trade-offs and demonstrating constant-time updates on large incremental workloads.

Paper structure. Section 2 positions ReOCL against related work. Sections 3–6 develop the approach, covering the core language, reactive semantics, transactional semantics, and compilation strategy. Section 7 describes the TypeScript implementation, which Section 8 evaluates. Section 9 closes with conclusions and future work.

MODELS Companion 2026, Málaga, Spain

ACM ISBN 979-8-4007-2903-4/2026/10
https://doi.org/10.1145/3837062.3839069

2 Related Work

To motivate our approach, we position ReOCL with respect to works on OCL foundations, incremental OCL and reactive maintenance, and web-based modeling, before analyzing the research gap.

OCL foundations. The OCL standard by the Object Management Group (OMG) defines a side-effect-free language for specifying invariants and queries and for defining operations over models [15]. However, it lacks a fully formal semantics, a shortcoming that has motivated theoretical works to formalize OCL and study its properties. In particular, Brucker and Wolff formalize its semantics, including four-valued logic, and provide a verification environment built on Isabelle/HOL, resulting in HOL-OCL [5] and Featherweight OCL [4]. On the tooling side, Eclipse OCL,[1] USE [9], and Dresden OCL [8] offer established support for OCL parsing, validation, and integration in MDE workflows. Despite this support, interest in OCL research has declined [20], and most tooling remains tied to desktop environments. As modeling moves to the web, revitalizing the language calls for execution environments suited to this setting.

Incremental OCL and reactive maintenance. Incremental techniques avoid re-evaluating OCL constraints from scratch after each model change, as in Cabot and Teniente's technology-agnostic, event-based evaluation approach [7]. Oriol and Teniente later analyze the limits of incremental OCL checking and constraint maintainability: checking general OCL constraints is not semidecidable, whereas OCL_{FO} —a relational-algebraic core— can be maintained in polynomial time [16]. They also extend the original approach with delta-maintained aggregations for sum, count, and size; witness counts for existential information; and repairs for violated constraints, while generating fewer constraints. Other OCL-specific techniques use impact analysis for dynamic constraint checking [10] or target incremental batch execution of OCL-based model transformations [11]. Finally, and beyond OCL, VIATRA supports event-driven model maintenance using the Rete algorithm, achieving efficient processing of model updates [3].

Web-based modeling. The growth of low-code development platforms (LCDPs) has renewed interest in web-based modeling through MDE techniques [19]. However, OCL.js,[2] the main web-oriented OCL implementation, has partial feature coverage, limited adoption, and weak integration with modern front-end toolchains [6]. As a consequence, web-oriented platforms make different language choices. For instance, Jjodel, a cloud-based collaborative workbench for creating visualizations and evolving DSLs [17], uses a lightweight interpretation of OCL to define views and an event-driven architecture to trigger validations, while its invariants rely on native JavaScript higher-order functions,[3] which integrate seamlessly with JSX, Jjodel's React-based framework. In turn, BESSER [1] employs B-OCL, a custom OCL interpreter in Python. Finally, Dandelion+ [14] supports validation through web-based workflows using the Epsilon Validation Language (EVL) [12]. Overall, these platforms reveal a lack of uniformity in constraint checking.

[1] https://projects.eclipse.org/projects/modeling.ocl
[2] https://ocl.stekoe.de/
[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#array_methods_and_empty_slots

Prior work reveals a twofold gap. On the one hand, formal and incremental OCL approaches lack a web-oriented execution environment. On the other, web-based platforms lack a formal foundation for reactive checking. ReOCL addresses this gap by combining a formally defined, restricted OCL core with incremental and reactive views atop a TypeScript framework.

3 The ReOCL Language

This section defines the scope, syntax, static semantics, and dynamic semantics of ReOCL.

3.1 Overview and scope

Web development must reconcile rich interactivity with efficient computation. Reactive programming offers a sweet spot: signals avoid recomputing large chunks of application state by maintaining a lightweight dependency graph between state and derived values, at a moderate memory cost [2]. ReOCL follows this paradigm with an OCL core compilable into reactive signals. Since OCL —and even more so incremental OCL— is not semidecidable [16], ReOCL focuses on features allowing efficient maintenance, including typed navigation and higher-order functions over collections.

Scope. ReOCL is deliberately restricted to ensure efficient incremental maintenance. It excludes features that are hard to track incrementally, such as iterate and sortedBy, and avoids global dependencies by dropping allInstances support. Attribute-sensitive predicate changes (e.g., mutating e.salary in place, rather than removing the element from its collections, mutating it, and adding it back) are not supported, as they would require element-level dependency tracking, which breaks constant-time updates.

Regarding semantics, ReOCL replaces OCL's four-valued logic with an option-valued partial semantics for undefinedness, avoiding substantial complexity [13]. Its type system is simplified as well: base types are restricted to Bool, Int, and String, and a single generic Collection type —a multiset allowing duplicates but no observable order— replaces specialized kinds such as Bag or Set. Integers also keep sum exact under repeated additions and subtractions, unlike floating point. Finally, optional association ends are collections of size zero or one rather than nullable references, so collection operators apply uniformly without null-specific cases (cf. [18]).

3.2 Syntax

Let $\mathcal{M}$ be a meta-model defining a set of meta-classes $\mathscr{C}$. For each class $C \in \mathscr{C}$, the meta-model defines $\text{fields}(C) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$, a set of typed fields. The meta-model also defines an inheritance hierarchy relation $\sqsubseteq$ over $\mathscr{C}$, assumed well-behaved, i.e., a partial order that respects field inheritance. We write $C_1 \sqsubseteq C_2$ if C_1 is C_2 or a subclass of C_2. ReOCL defines the following types:

$$\tau ::= \text{Bool} \mid \text{Int} \mid \text{String} \mid \text{Object}(C) \mid \text{Collection}(\tau), \text{ where } C \in \mathscr{C}.$$

A ReOCL expression e is defined by the following grammar:

$$\begin{aligned} e ::= {} & \text{true} \mid \text{false} \mid n \mid \text{"}s\text{"} \\ & \mid \text{self} \mid x \\ & \mid e.f \mid e.f@\text{pre} \\ & \mid e_1 \oplus e_2 \mid \text{not } e \end{aligned}$$

$$
\begin{aligned}
&\mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif} \\
&\mid e_1 \to \text{select}(x \mid e_2) \mid e_1 \to \text{reject}(x \mid e_2) \\
&\mid e_1 \to \text{collect}(x \mid e_2) \\
&\mid e_1 \to \text{forAll}(x \mid e_2) \mid e_1 \to \text{exists}(x \mid e_2) \mid e_1 \to \text{one}(x \mid e_2) \\
&\mid e_1 \to \text{isUnique}(x \mid e_2) \\
&\mid e \to \text{size}() \mid e \to \text{sum}() \mid e \to \text{isEmpty}() \mid e \to \text{notEmpty}() \\
&\mid e \to \text{oclIsKindOf}(C) \mid e \to \text{oclIsTypeOf}(C),
\end{aligned}
$$

where n is an integer literal, s is a string literal, x is a variable name, and $f \in \text{fields}(C)$ for some $C \in \mathscr{C}$. Binary operators ($\oplus$) comprise arithmetic (+, -, *, /), comparisons (<, >, <=, >=), equality (=, <>), and Boolean connectives (and, or, implies, xor).

Finally, invariants have the form context C inv m : e, where $C \in \mathscr{C}$, m is the name of the invariant, and e is a well-typed Boolean expression in a typing environment with self $\mapsto$ Object(C).

3.3 Static semantics

Figure 6 (in the appendix) presents the typing rules of ReOCL. They follow the expected OCL typings for the supported core, with two particularities.

First, field navigation is typed directly from the meta-model. If e has type Object(C) and fields(C) declares a field $f : \tau$, then $e.f$ has type τ (cf. T-Nav). Pre-state navigation is typed in the same way: $e.f$@pre has the same static type as $e.f$ (cf. T-Pre). Thus, @pre does not introduce a separate type-level construct; its difference from ordinary navigation is captured by the dynamic semantics.

Second, the typing rules separate collection operators by the kind of value they produce, which determines how each is maintained incrementally. **View-producing operators** —select, reject, and collect— return collection types and denote intermediate views, whereas **terminal operators** —forAll, exists, one, isUnique, size, sum, isEmpty, and notEmpty— return scalars and close a pipeline.

3.4 Dynamic semantics

We now define the non-reactive meaning of well-typed expressions. Figure 7 (in the appendix) presents representative evaluation rules, and the full specification is available online.[4]

Values and results. A well-typed expression denotes a value:

$$
\begin{aligned}
v \in \text{Val} ::= \text{VTrue} \mid \text{VFalse} \mid \text{VInt}(n) \mid \text{VString}(s) \\
\mid \text{VObj}(oid, C) \mid \text{VColl}(vs).
\end{aligned}
$$

As evaluation may fail, ReOCL results are optional: option(Val) ::= Some(v) | None, with None marking failure. Following [13], this option-valued partial semantics handles undefinedness in place of OCL's four-valued logic, dropping support for null and invalid.

Evaluation and state. To produce a result, evaluation reads an environment γ binding variables (including self), a current store σ, and a pre-state store σ_{pre}:

$$[\![e]\!](\gamma, \sigma, \sigma_{\text{pre}}) \in \text{option}(\text{Val}).$$

Both stores map locations to values: $\ell = \text{fieldStateId}(C, oid, f)$ identifies field f of object oid in class C. The current store is total, so $\sigma(\ell)$ always yields a value. Conversely, the pre-state store is partial, yielding one only for fields recorded when the transaction began. They differ at navigation: if e evaluates to VObj(oid, C), ordinary navigation reads the current store, and @pre, the pre-state store:

$$
\begin{aligned}
e.f \quad &\text{reads from} \quad \sigma(\text{fieldStateId}(C, oid, f)), \\
e.f\text{@pre} \quad &\text{reads from} \quad \sigma_{\text{pre}}(\text{fieldStateId}(C, oid, f)).
\end{aligned}
$$

Undefinedness. Three situations yield None: division by zero, an iterator body that evaluates to None or to a value of the wrong kind, and a Boolean or arithmetic operator applied to non-Boolean or non-integer operands. Two helpers formalize the checks involved: expectBool(v) returns Some(b) for Boolean v and None otherwise, and boolVal(b) maps a Boolean back to VTrue or VFalse. Propagation to the enclosing expression depends on the construct.

Strict constructs require all relevant subexpressions to be defined. For select, reject, collect, one, and isUnique, the iterator body runs over every element the operator needs, and any failure yields None. Arithmetic and comparison operators are strict as well, as are not and xor, which require defined Boolean operands. The aggregates size, isEmpty, and notEmpty require a defined source collection, and sum requires integer elements. The type tests oclIsKindOf and oclIsTypeOf are defined over objects, and yield None otherwise.

Short-circuiting constructs, in contrast, evaluate only the subexpressions needed, so only those can fail: forAll and exists observe undefinedness only in evaluated predicates, and the connectives and, or, and implies stay two-valued, yielding None unless every evaluated operand is VTrue or VFalse.

Finally, a result r is *true-valued*, written isTrue(r), when it matches Some(VTrue). An invariant body e is then satisfied when:

$$\text{isTrue}([\![e]\!](\gamma, \sigma, \sigma_{\text{pre}})).$$

4 Reactive Semantics

Section 3.4 defined the non-reactive meaning of ReOCL expressions. This section explains reactive maintenance of collection-valued expressions via membership delta propagation and operator pipelines.

4.1 Reactive values and delta streams

ReOCL interprets collection operators as view transformations, unlike JavaScript's higher-order functions, which eagerly re-traverse the entire array after every change, even after a single insertion.

ReOCL maintains two kinds of reactive value: a *scalar*, notifying its dependents on change, and a *collection view*, pairing its content S with a stream of membership deltas ADD(v) or REMOVE(v).

View-producing operators (e.g., select) turn one view into another; terminal ones (e.g., forAll) consume a view and maintain a scalar. Each delta updates the view directly, so cost is independent of $|S|$. As S is a multiset, ADD(v) inserts one occurrence and REMOVE(v) withdraws exactly one, leaving duplicates untouched. Operators act occurrence-wise: a mapped view holds one image per occurrence, sum adds every duplicate, and isUnique counts each key separately. If v occurs twice in S, REMOVE(v) leaves one occurrence and decreases sum by v.

Importantly, since select and collect retain their content, views can be bound directly to the interface, which then updates incrementally: ReOCL serves not only to check invariants, but also to render derived collections from the same expressions.

[4] https://github.com/ReOCL/reocl/blob/main/docs/dynamic_semantics.pdf

4.2 View-producing operators

An operator is *lifted*, written $\widehat{\text{op}}$, when it consumes input deltas, may update internal state, and emits zero or more output deltas. For instance, lifted select emits only the deltas whose element satisfies the predicate:

$$\widehat{\text{select}}(p)(\text{ADD}(v)) = \begin{cases}\{\text{ADD}(v)\}, & \text{if } p(v)\\ \varnothing, & \text{otherwise.}\end{cases}$$

$$\widehat{\text{select}}(p)(\text{REMOVE}(v)) = \begin{cases}\{\text{REMOVE}(v)\}, & \text{if } p(v)\\ \varnothing, & \text{otherwise.}\end{cases}$$

Both cases apply the same test and emit the matching delta. The dual of $\widehat{\text{select}}$ is $\widehat{\text{reject}}$:

$$\widehat{\text{reject}}(p) = \widehat{\text{select}}(\lambda x.\neg p(x)).$$

For collect, the lifted operator maps each input membership delta to a corresponding output membership delta. On addition, it computes the transformed value and emits an ADD delta; on removal, it emits the removal of the corresponding mapped value:

$$\widehat{\text{collect}}(g)(\text{ADD}(v)) = \{\text{ADD}(g(v))\}$$
$$\widehat{\text{collect}}(g)(\text{REMOVE}(v)) = \{\text{REMOVE}(g(v))\},$$

assuming that g is stable between deltas and evaluates in constant time. These definitions assume p and g are defined on every element of the stream; otherwise, the whole expression is undefined.

4.3 Terminal operators

Terminal operators keep a small piece of state (e.g., a counter, occurrence counts, or a running total) from which the result is read:

forAll This operator maintains the number of violating elements $b = |\{v \in S \mid \neg p(v)\}|$. The result is true exactly when $b = 0$.

exists and one These operators maintain the number of matching elements $m = |\{v \in S \mid p(v)\}|$, so that $\text{exists}(p) := m > 0$ and $\text{one}(p) := m = 1$.

isUnique This operator maintains the count $c(y) = |\{v \in S \mid g(v) = y\}|$ of elements sharing each key y produced by the iterator body g, and the number $d = |\{y \mid c(y) > 1\}|$ of duplicated keys. A delta adjusts one count, and the result is true exactly when $d = 0$, so it is read in constant time.

size This operator maintains a counter $k = |S|$. An ADD delta increments k, while a REMOVE delta decrements it.

sum This operator maintains a running total $s = \sum_{v \in S} v$. An ADD(v) delta increments s by v, while a REMOVE(v) delta decrements it by v.

isEmpty and notEmpty These operators maintain a counter $k = |S|$, as size does. The operator isEmpty checks whether $k = 0$, while notEmpty checks whether $k > 0$.

Overall, if predicates and keys evaluate in constant time, every operator above updates its state in constant time per delta, without revisiting the view. Lifted operators also agree with the eager ones: applying deltas gives the same content and the same scalar as evaluating the operator again over the updated collection.

4.4 Chaining pipelines

Each operator initializes its internal state by scanning its input view once, at $O(N)$ cost, where $N = |S|$ is the source collection size. From then on, membership deltas flow through a sequence of operator stages. For example,

$$S \rightarrow \text{select}(x \mid p(x)) \rightarrow \text{collect}(x \mid g(x)) \rightarrow \text{sum}()$$

comprises three stages: two view-producing transformers followed by a single terminal aggregate.

In such a pipeline, each delta is processed in time independent of N, provided predicates are stable and evaluate in constant time. Its maintenance cost is $O(k \cdot a)$, with k the fixed number of stages and a the cost of one predicate or mapping evaluation. Since both are constant for a given invariant, the cost stays $O(1)$ per delta. This assumes that each stage emits at most one delta, as every core operator does. Only implicit navigation over collections would emit several deltas, and it is not part of the core (cf. T-Nav). More generally, cost scales with the stages of an invariant and with the number of invariants, since each maintains its own pipeline.

Recall, however, that this guarantee holds only when predicates stay stable: attribute-sensitive mutations would need element-level dependency tracking to avoid invalidating maintained summaries. Preserving constant-time updates also rules out operations with non-constant worst-case behavior, such as closure, iterate, sortedBy, and nested aggregates over navigated subcollections.

5 Transactional Semantics

Reactive maintenance propagates model updates to dependent invariant results, but propagation alone cannot prevent invalid states from becoming visible or persistent. ReOCL therefore supports *transactions* that wrap state mutations and enforce invariant validity, providing atomicity and @pre reads.

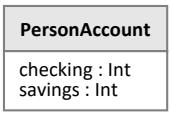


Figure 3: A tiny meta-model.

Consider the meta-model in Figure 3. An invariant such as self.checking > 0 prevents negative balances through rollback, while an invariant such as

```
context PersonAccount inv conservation:
self.checking + self.savings =
self.checking@pre + self.savings@pre
```

requires access to the pre-transaction state. Without transactions, the invalid state persists after a violation, and the pre-transaction values are lost.

5.1 Transactional configurations

Transactions are modeled by configurations $\langle \sigma, \sigma_{\text{pre}}, \mathcal{I} \rangle$, with $\mathcal{I}$ being the set of watched invariants. A transaction bounds a group of mutations: it records the state before the group, evaluates the watched invariants against the mutated store, and either keeps the mutations or restores the previous values.

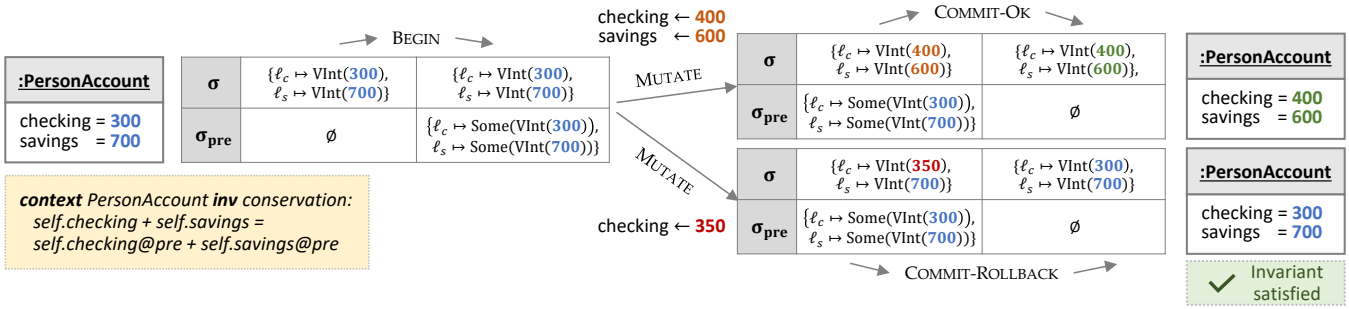


Figure 2: Example of two transactions: one that commits successfully and one that rolls back.

The pre-state store gives @pre a stable meaning and supplies rollback values: $\sigma_{\text{pre}}(\ell) = \text{Some}(v)$ records that ℓ held v at transaction start, and None that nothing was recorded. It is field-local: $e.f$@pre reads field f's entry, whereas object-graph pre-state navigation would require copying reachable structure and is outside ReOCL's scope.

Commits are guarded by valid_all, which holds when every watched invariant $c \in \mathcal{I}$ is true-valued:

$$\text{valid\_all}(\sigma, \sigma_{\text{pre}}, \mathcal{I}) \iff \forall c \in \mathcal{I}.\ \text{isTrue}(c(\sigma, \sigma_{\text{pre}})).$$

In particular, we define the following transition rules:

$$\frac{}{\langle \sigma, \varnothing, \mathcal{I} \rangle \xrightarrow{\text{BEGIN}} \langle \sigma, \text{snapshot}(\sigma), \mathcal{I} \rangle}\ \textsc{Begin}$$

$$\frac{}{\langle \sigma, \sigma_{\text{pre}}, \mathcal{I} \rangle \xrightarrow{m} \langle \sigma[m], \sigma_{\text{pre}}, \mathcal{I} \rangle}\ \textsc{Mutate}$$

$$\frac{\text{valid\_all}(\sigma, \sigma_{\text{pre}}, \mathcal{I})}{\langle \sigma, \sigma_{\text{pre}}, \mathcal{I} \rangle \xrightarrow{\text{COMMIT}} \langle \sigma, \varnothing, \mathcal{I} \rangle}\ \textsc{Commit-Ok}$$

$$\frac{\neg\text{valid\_all}(\sigma, \sigma_{\text{pre}}, \mathcal{I})}{\langle \sigma, \sigma_{\text{pre}}, \mathcal{I} \rangle \xrightarrow{\text{COMMIT}} \langle \sigma[\sigma_{\text{pre}}], \varnothing, \mathcal{I} \rangle}\ \textsc{Commit-Rollback}$$

Here, $\sigma[m]$ is the store after applying mutation m. Rollback $\sigma[\sigma_{\text{pre}}]$ restores values recorded in σ_{pre}, defaulting to σ otherwise. Finally, while snapshot(σ) formally copies the entire store, an implementation can lazily copy only mutated entries.

5.2 Conservation example

Consider Figure 2, over the meta-model of Figure 3, watching the conservation invariant c_{cons}. For a PersonAccount with identifier oid, write

$$\ell_c = \text{fieldStateId}(\text{PersonAccount}, oid, \text{checking}), \text{ and}$$
$$\ell_s = \text{fieldStateId}(\text{PersonAccount}, oid, \text{savings}).$$

The initial store σ_0 maps $\sigma_0(\ell_c) = \text{VInt}(300)$ and $\sigma_0(\ell_s) = \text{VInt}(700)$, and Begin populates σ_{pre} with both entries. The top transaction transfers 100 from savings to checking, satisfying $\mathcal{I} = \{c_{cons}\}$ and committing. The bottom one raises checking to 350 without touching savings, violating c_{cons} and rolling back to σ_0 (in blue). In both cases, all watched invariants are satisfied at the end.

6 Compilation to Signals

This section compiles ReOCL expressions into TypeScript signals that preserve their semantics. Currently, the strategy is a specification that developers apply by hand: the library provides the runtime primitives it targets, and an OCL front-end automating it remains future work.

We write the compilation function as $\mathcal{T}[\![e]\!]_\Gamma = E$, where Γ is the typing environment and E a generated TypeScript expression. It is compositional: scalar navigation becomes signal reads, Boolean and arithmetic expressions become derived expressions, view-producing operators become view transformers, and terminal operators become maintained aggregate signals. The meta-model $\mathcal{M}$ parameterizes the compiler with fields and inheritance.

6.1 Target signal model

The target compilation language, inspired by the JavaScript signals proposal by Technical Committee 39 (TC39),[5] consists of four reactive primitives that form a signal graph:

Mutable signals (Signal<T>) They hold scalar values and notify dependents on change. Each scalar model field maps to a mutable signal.

Computed signals (ReadonlySignal<T>) They derive scalar values from other signals, with dependencies tracked dynamically by the framework. Scalar invariant bodies and intermediate scalar expressions map to these wrappers.

Reactive collection views (TypedReactiveCollection<T>) They represent collection-valued expressions and expose delta streams consumed by subsequent operators.

Maintained aggregates (ReadonlySignal<T>) They represent terminal computations such as size, sum, or forAll. They update via delta streams rather than dependency tracking, but expose a read-only signal interface to dependents. With no recursive constructs in the core, the signal graph is acyclic by construction, and the runtime rejects any cycle a hand-written definition routes through a computed signal.

The strategy maps a meta-class C to a TypeScript class $[\![C]\!]$ whose fields are the reactive encodings of fields(C). Scalar fields become mutable signals, collection fields become reactive collections, and invariants or terminal aggregates expose read-only signals registered with the transactional runtime.

[5] https://github.com/tc39/proposal-signals

For example, the meta-class Department from the meta-model in Figure 1, together with the minSalary invariant, which ensures that all employees earn at least 20,000, compiles to:

```
class CompiledDepartment {
  budget$: Signal<number>
   employees$: TypedReactiveCollection<Employee>
  minSalary$: ReadonlySignal<boolean>
}
```

Note that the invariant signal minSalary$ is automatically updated when employees$ emits deltas.

6.2 Scalar expressions

We first consider scalar expressions, whose compilation either reads field signals or builds computed signals over them. Local variables compile directly to their TypeScript counterparts ($\mathcal{T}[\![x]\!]_\Gamma = x$). Field navigation compiles to the reactive field:

$$\mathcal{T}[\![e.f]\!]_\Gamma = \mathcal{T}[\![e]\!]_\Gamma.\text{f\$}.$$

The target TypeScript type determines how this reference is used. Scalar fields have type Signal<T> and are read through their value property; collection-valued fields have type TypedReactiveCollection<T> and expose collection operations and delta streams. Pre-state navigation compiles to a lookup in the pre-state store:

$$\mathcal{T}[\![e.f@\text{pre}]\!]_\Gamma = \$\text{pre}(\mathcal{T}[\![e]\!]_\Gamma.\text{f\$}).$$

The runtime primitive $pre is defined only during an active transaction. Outside a transaction, evaluating $pre safely yields None. This primitive realizes the field-local pre-state lookup defined in Section 5.1.

Arithmetic and order comparisons compile to strict helpers (e.g., oclAdd, oclDiv, or oclLt) that unwrap signal values, check for non-integer operands or division by zero, and otherwise yield None. Equality (oclEq, oclNeq) is total instead, as it compares values of any kind. The strict connectives not and xor compile to oclNot and oclXor, two-valued over defined Booleans and None on other operands. In contrast, and, or, and implies need no helper: they compile to short-circuiting conditionals that evaluate the right operand only when the left one does not already settle the result. As dependencies are discovered dynamically, the unevaluated branch registers none, preserving the option-valued semantics.

Type tests compile to methods on the compiled object, as in $\mathcal{T}[\![e \to \text{oclIsKindOf}(C)]\!]_\Gamma = \mathcal{T}[\![e]\!]_\Gamma.\text{oclIsKindOf}(C)$, and analogously for oclIsTypeOf. Since the class of an object never changes, these tests need no reactive dependency and remain stable predicates within collection views.

6.3 Collection pipelines

View-producing collection operators compile to view transformers applied to the compiled source. For example:

$$\mathcal{T}[\![e_1 \to \text{select}(x \mid e_2)]\!]_\Gamma = \mathcal{T}[\![e_1]\!]_\Gamma.\text{select}(x \Rightarrow \mathcal{T}[\![e_2]\!]_{\Gamma,x:\tau}).$$

The operators collect and reject compile analogously through collect and select under the negated predicate, as explained in Section 4.2.

Terminal aggregate operators compile to maintained aggregate signals. The compilation of exists, one, isUnique, isEmpty, and notEmpty is analogous:

$$\mathcal{T}[\![e \to \text{size}()]\!]_\Gamma = \mathcal{T}[\![e]\!]_\Gamma.\text{size}(),$$
$$\mathcal{T}[\![e \to \text{sum}()]\!]_\Gamma = \mathcal{T}[\![e]\!]_\Gamma.\text{sum}(),$$
$$\mathcal{T}[\![e_1 \to \text{forAll}(x \mid e_2)]\!]_\Gamma = \mathcal{T}[\![e_1]\!]_\Gamma.\text{forAll}(x \Rightarrow \mathcal{T}[\![e_2]\!]_{\Gamma,x:\tau}).$$

Note that this approach differs from eager array execution using JavaScript's higher-order functions. The expression:

$$c \to \text{select}(x \mid p(x)) \to \text{collect}(x \mid q(x)) \to \text{sum}()$$

compiles to an efficient graph of collection views and aggregate nodes, rather than to repeated calls to JavaScript's native filter, map, and reduce higher-order functions.

6.4 Invariant compilation

An invariant context C inv m : e compiles to a reactive signal in the class for C. If e is a Boolean collection aggregate (i.e., forAll, exists, one, isUnique, isEmpty, or notEmpty), it already returns a live ReadonlySignal<boolean> maintained via delta streams, and is emitted directly:

$$\text{const } m\$ = \mathcal{T}[\![e]\!]_\Gamma;$$

Integer aggregates such as size and sum return numeric signals and become invariant bodies only when embedded in Boolean expressions, such as comparisons.

If e is a scalar expression (or contains aggregates nested inside scalar operators), the body is wrapped in a computed signal that re-evaluates when its dependencies change:

$$\text{const } m\$ = \text{computed}(() \Rightarrow \text{isTrue}(\mathcal{T}[\![e]\!]_\Gamma));$$

The wrapper yields true exactly when the compiled expression produces a defined Boolean true result, and false otherwise, realizing the isTrue predicate of Section 3.4.

7 ReOCL in practice

This section showcases how to use the ReOCL TypeScript library in practice to define constraint-backed web interfaces. The library is written in TypeScript atop Bun,[6] and is open source on GitHub.[7] It leverages @preact/signals-core,[8] Preact's framework-agnostic core signals package, whose dynamic dependency tracking and synchronous lazy evaluation natively mirror OCL's short-circuiting semantics.

Building on the compilation strategy of Section 6, which developers currently apply by hand, the library evaluates the resulting expressions within Preact's dependency graph. Our web application, which is available online,[9] demonstrates this with three examples: an employee management demo (Figure 1), a transactional banking demo (Figure 4), and a pipeline demo rendering every stage of select → collect → sum. This last demo binds the maintained select and collect views directly to the interface, showcasing ReOCL's ability to maintain views that reactive web frameworks can render directly.

[6] https://bun.com/
[7] https://github.com/ReOCL/reocl
[8] https://preactjs.com/guide/v10/signals
[9] https://miso.es/tools/ReOCL

We illustrate the developer workflow using the transactional banking example through four steps:

Step 1: Register the meta-model. A central ReactiveStore holds the reactive state. Each meta-class is registered:

```
const store = new ReactiveStore();
store.registerClass("PersonAccount", {
  checking: { tag: "Int", initial: 300 },
  savings:  { tag: "Int", initial: 700 }
});
```

Step 2: Compile invariants into reactive signals. For each meta-class, the developer hand-writes the target class of Section 6.1, exposing field signals and invariant signals built with computed for derived expressions:

```
class CompiledPersonAccount {
   obj = store.getClass("PersonAccount")!
   .create({checking: 300, savings: 700});
   checking$ = intSignal(this.obj, "checking");
    savings$ = intSignal(this.obj, "savings");
   conservation$ = computed(() => {
     const curr = this.checking$.value + this.savings$.value;
   const prev = this.obj.preInt("checking") + this.obj.preInt("savings");
   return curr === prev;
   });
 }

const pa = new CompiledPersonAccount();
```

Accessors such as preInt realize the field-local pre-state reads of Section 5.1: they wrap the $pre primitive of Section 6.2 and unwrap its option-typed result into a native number, defaulting to the current field value when no pre-state is recorded, as happens outside a transaction. Those behind intSignal, strSignal, and boolSignal project reads alike, falling back to their type's default; expressions observing undefinedness read the option-typed value instead. In Figure 4, each invariant carries its OCL source for display only, while the signal beside it performs the check.

Step 3: Wrap mutations in transactions (optional). A transaction watches the compiled invariants and wraps mutations in a Begin → Mutate → Commit-Ok/Rollback cycle from Section 5.1. The transaction layer automatically rolls back the store if any watched invariant is violated at commit time:

```
const tx = store.transaction(pa.conservation$);
  const personAccount = pa.obj;
  function doTransfer() {
  tx.begin();
  tx.mutate(() => {
  personAccount.setInt("savings", pa.savings$.value - amount);
personAccount.setInt("checking", pa.checking$.value + amount);
  });
  tx.commit(); // rolls back if conservation$ is false
}
```

Step 4: Integrate into UI components. A button click triggers the transaction, and compiled signals drive the rendering of the resulting state. Leveraging Preact's signals integration, plain JSX components re-render whenever a signal they read changes:

```
function TransactionDemo() {
   return <div>
```

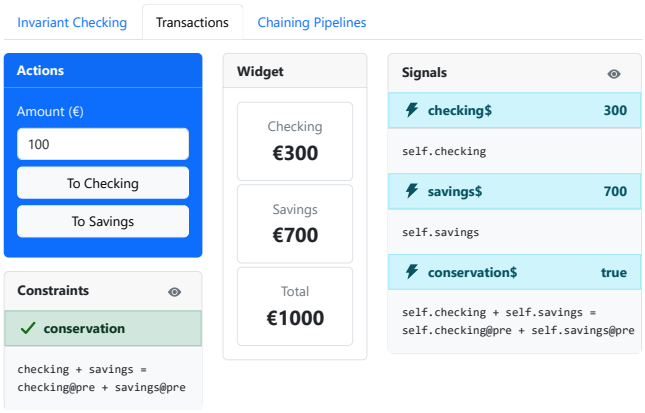


Figure 4: ReOCL transaction example demo in the browser.

```
    <button onClick={doTransfer}>To Checking</button>
    <BalanceCard />
    </div>;
  }

function BalanceCard() {
   return <div>
   <p>Checking: {pa.checking$.value}</p>
   <p>Savings: {pa.savings$.value}</p>
   <p>Conservation: {pa.conservation$.value ? "OK" : "VIOLATED"}</p>
   </div>;
  }
```

Overall, ReOCL's framework-agnostic reactive graph decouples domain constraints from UI rendering, eliminating manual event boilerplate. Model mutations automatically propagate to enforce transactions and update derived states, so the frontend observes and renders results that satisfy every watched invariant at each commit.

8 Evaluation

We evaluate the approach through the following research questions:

> **RQ1.** *How does the foundation of ReOCL compare to adjacent web-based modeling validation approaches in terms of OCL feature coverage and formal foundations?*
>
> **RQ2.** *How does the ReOCL implementation compare to adjacent web-based modeling validation approaches in terms of performance?*

Experimental setup. Benchmarks were run on a Windows 11 x64 laptop (Intel® Core™ i7-1255U, 16 GB RAM). Results represent medians across 1–20 size-dependent trials using fixed seeds, with mutation and total times measured after a warm-up pass. Reproducible evaluation data is available online.[10]

8.1 RQ1. Feature coverage

We compare ReOCL against the adjacent web-based validation approaches explored in Section 2: OCL.js, B-OCL, and JSX/Jjodel.

[10] https://github.com/ReOCL/reocl/tree/main/docs/evaluation

	select	reject	collect	forAll	exists	any	one	isUnique	size	isEmpty	notEmpty	sum	union	append	asSet	at	first	last	includes	includesAll	excludes	excludesAll	including	excluding	intersection	flatten	count	sortedBy	closure	iterate	collectNested	selectByKind	selectByType	product	Coverage
OCL.js	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	—	—	—	—	—	—	—	—	—	—	—	—	65 %
B-OCL	●	●	●	●	●	—	—	—	●	—	—	—	—	—	—	—	—	—	●	—	●	—	—	—	—	—	—	—	—	—	—	—	—	—	24 %
JSX / Jjodel	●	●	●	●	●	●	⊙	⊙	●	●	●	●	⊙	●	●	●	●	●	●	⊙	●	⊙	⊙	●	⊙	●	⊙	●	—	●	●	⊙	⊙	—	79 %
ReOCL	●	●	●	●	●	—	●	●	●	●	●	●	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	32 %
ReOCL (proj.)	●	●	●	●	●	⊙	●	●	●	●	●	●	⊙	⊙	⊙	⊙	⊙	⊙	●	⊙	●	⊙	●	●	⊙	⊙	●	○	○	○	⊙	●	●	○	71 %

Legend: ● fully supported ⊙ partial support ○ unfeasible — not implemented ■ not implemented (projected support as future work)

Table 1: RQ1: Collection methods coverage for web-based constraint validators.

Collection methods coverage. Table 1 compares collection operation coverage based on OCL.js's feature matrix.[11] Support across tools is highly fragmented. JSX/Jjodel is the most complete, but leans on native JavaScript workarounds (e.g., e.sum() becomes e.reduce((acc, x) => acc + x, 0)). ReOCL implements a moderate subset, with a larger one planned: counter-based aggregates (e.g., excludes, includes, or count) share the incremental pattern of size or sum and remain future work, whereas order-dependent operators (e.g., at or first) require indexing, and hence refinements to collection support. Operators breaking constant-time incrementality (e.g., closure or iterate) remain infeasible.

High-level OCL features coverage. Table 2 extends the comparison to high-level OCL constructs. ReOCL matches existing tools on core logic and type checking (i.e., if-then-else, oclIsKindOf, and oclIsTypeOf), resolved over the inheritance hierarchy, though primitive types lack floating-point Real numbers by design. Global lookups like allInstances remain formally infeasible, introducing untrackable graph-wide dependencies that break delta-tracking. Conversely, several projected features fit the architecture well: derived properties (def) and local bindings (let) become computed signals and context variables, while type casting (oclAsType) and undefinedness checks (oclIsUndefined) fit the option-valued semantics of Section 3.4. Uniquely, ReOCL offers (partial) @pre support.

Formal foundation. To our knowledge, no other web-based approach combines delta-maintained incrementality, transaction support, and a formal semantic foundation. While OCL.js, B-OCL, and JSX rely on *ad hoc* interpreters, ReOCL is grounded in a formal denotational semantics with defined reactive and transactional evaluation layers.

To answer RQ1: currently, ReOCL implements a moderate subset of OCL features. However, the projected coverage is comparable to adjacent web-based approaches. Particularly, unlike the rest of the approaches, ReOCL is uniquely grounded in formal foundations and has @pre and transactional support.

8.2 RQ2. Performance comparison

We evaluate ReOCL against OCL.js and a native, eager JavaScript baseline (representing JSX/Jjodel) over increasing model sizes. We measure initialization time (setup), mutation time (processing one delta), and total time (setup plus sequential mutations).

Figure 5a shows $O(N)$ initialization costs for all approaches. At $N = 10^6$, ReOCL pays the highest upfront cost (3.9 s), constructing the signal graph and precomputing aggregates, whereas OCL.js and the baseline stay at ~8–161 ms by only allocating the collections and deferring evaluation until read time.

Figure 5b reports per-mutation times. ReOCL maintains a strictly constant $O(1)$ cost of ~5 µs across all sizes, validating the analysis in Section 4.4. The eager baseline and OCL.js degrade linearly to 29 ms and 630 ms at $N = 10^6$, making ReOCL between 5,700× and 10^5× faster, respectively. Direct scans, however, are marginally faster at the smallest sizes, $N \leq 100$.

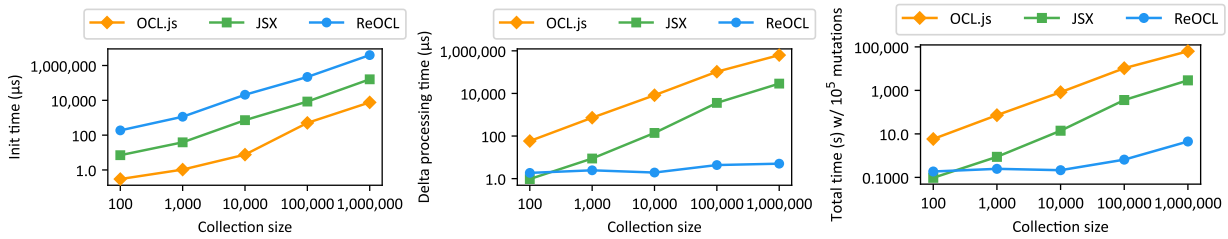
Finally, Figure 5c plots total time, projected as the measured initialization cost plus 100,000 times the median per-delta cost. ReOCL scales almost flatly, as its $O(1)$ updates amortize the initial setup. The baseline and OCL.js grow linearly, rendering ReOCL between 600× and 14,000× faster at the largest sizes.

To answer RQ2: ReOCL trades a higher $O(N)$ initialization cost for strictly flat $O(1)$ per-delta updates. For sustained reactive workloads on large models ($N = 10^6$), this architecture amortizes setup

Feature	OCL.js	B-OCL	JSX	ReOCL
String methods	●	—	●	●
Numeric operators	●	⊙	●	⊙
oclIsKind/TypeOf	●	—	⊙	●
allInstances	—	●	●	○
Navigation	●	⊙	●	⊙
@pre	—	—	—	⊙
if-then-else	●	●	●	●
let	●	—	—	●
oclAsType	●	—	●	●
oclIsUndefined	●	●	●	⊙
def	●	—	●	●
Bools/Ints/Reals/Strings	●	●	●	⊙
Tuple/Set/Sequence/Bag	—	—	⊙	⊙
Enumerations	●	●	●	●

Table 2: RQ1: High-level OCL features coverage for web-based constraint validators.

[11] https://ocl.stekoe.de/guide/compatibility.html



(a) Initialization time. **(b) Mutation time.** **(c) Total time.**

Figure 5: RQ2: Performance comparison for self.employees->select(e | e.active)->size() <= self.capacity.

costs, delivering performance gains of multiple orders of magnitude over eager JavaScript recomputation and OCL.js.

9 Conclusions and Future Work

This paper has presented ReOCL, a restricted OCL core tailored for reactive TypeScript-based web modeling. By compiling invariants into reactive signals, maintaining collection views via membership deltas, and wrapping updates in a transactional layer, ReOCL guarantees constant-time per-delta validation for the supported fragment under stable, constant-cost iterator bodies. Our evaluation shows partial, deliberately restricted feature coverage compared with existing web-based frameworks, while delivering speedups of multiple orders of magnitude for large workloads.

We are currently pursuing two lines of work. First, we are formalizing ReOCL in the Coq/Rocq proof assistant[12] to mechanize its core properties: type soundness, compilation preservation, transaction safety, and incremental correctness. Second, we are developing a compiler that parses OCL into these reactive signals, within an end-to-end TypeScript framework integrated with build systems. Developers could then write constraints in OCL files alongside their meta-models, adopting modeling without leaving the typed, component-based workflow they already use, thanks to type completion and adapter bridges for UI libraries such as React and Svelte.

We also intend to broaden the evaluation to complex interactive scenarios, such as those that involve forms with computed fields, large filtered tables, and multi-step wizards. Finally, we plan to extend the language itself with specialized collection types such as Set and Sequence, to investigate UPDATE deltas for complex mutations, and to add undefinedness-aware incremental maintenance to support full OCL four-valued logic.

Acknowledgments

This article has been funded by the SE4GenAI (PID2023-147592OB-I00) and BOOSTER (PID2024-155231OB-I00) projects funded by MCIU / AEI /10.13039/501100011033 / FEDER, EU.

References

[1] Iván Alfonso et al. 2024. Building BESSER: an open-source low-code platform. In *International Conference on Business Process Modeling, Development and Support*. Springer, 203–212.
[2] Shrinivass Arunachalam Balasubramanian. 2025. Signal-First Architectures: Rethinking Front-End Reactivity. *arXiv preprint arXiv:2506.13815* (2025).
[3] Gábor Bergmann et al. 2015. VIATRA 3: A Reactive Model Transformation Platform. In *Int. Conf. on Theory and Practice of Model Trans.* Springer, 101–110.
[4] Achim D. Brucker, Frédéric Tuong, and Burkhart Wolff. 2015. *Featherweight OCL: A Proposal for a Machine-Checked Formal Semantics for OCL 2.5*. Technical Report 1582. LRI, Univ. Paris Sud, CNRS, Centrale Supélec, Université Paris-Saclay.
[5] Achim D. Brucker and Burkhart Wolff. 2008. HOL-OCL: A Formal Proof Environment for UML/OCL. In *Fundamental Approaches to Software Engineering (FASE) (Lecture Notes in Computer Science, Vol. 4961)*. Springer, 97–114.
[6] Antonio Bucchiarone, Juri Di Rocco, Damiano Di Vincenzo, and Alfonso Pierantonio. 2025. From OCL to JSX: declarative constraint modeling in modern SaaS tools. In *STAF 2025 Workshops (CEUR Workshop Proc., Vol. 4122)*. CEUR-WS.org.
[7] Jordi Cabot and Ernest Teniente. 2006. Incremental Evaluation of OCL Constraints. In *CAiSE 2006 (Lecture Notes in Computer Science, Vol. 4001)*, Eric Dubois and Klaus Pohl (Eds.). Springer, 81–95.
[8] Birgit Demuth and Claas Wilke. 2009. Model and object verification by using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia*. 687–690.
[9] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming* 69, 1-3 (2007), 27–34.
[10] Iris Groher, Alexander Reder, and Alexander Egyed. 2010. Incremental consistency checking of dynamic constraints. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 203–217.
[11] Frédéric Jouault and Olivier Beaudoux. 2016. Efficient OCL-based Incremental Transformations. In *OCL@MODELS 2016 (CEUR Workshop Proceedings, Vol. 1756)*, Achim D. Brucker, Jordi Cabot, and Adolfo Sánchez-Barbudo (Eds.). 121–136.
[12] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2009. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*. Springer, 204–218.
[13] Kevin Lano. 2014. Null Considered Harmful (for Transformation Verification). In *VOLT@STAF 2014 (CEUR Workshop Proceedings, Vol. 1325)*, Moussa Amrani, Eugene Syriani, and Manuel Wimmer (Eds.). CEUR-WS.org, 26–35.
[14] Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara. 2026. A model and workflow-driven approach for engineering domain-specific low-code platforms and applications. *Softw. Syst. Model.* 25, 2 (2026), 579–614.
[15] Object Management Group. 2014. *Object Constraint Language (OCL) Version 2.4*. Technical Report. OMG. https://www.omg.org/spec/OCL/2.4/
[16] Xavier Oriol. 2017. *Incremental checking and maintenance of UML/OCL integrity constraints*. Ph. D. Dissertation. Polytechnic University of Catalonia, Spain.
[17] Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Damiano Di Vincenzo, Alfonso Pierantonio, and Giordano Tinella. 2023. Jjodel – A Reflective Cloud-Based Modeling Framework. In *MODELS 2023 Companion*. IEEE, 55–59.
[18] Friedrich Steimann, Robert Clarisó, and Martin Gogolla. 2025. Meet OCL♯, a relational object constraint language. *SoSyM* 24, 6 (2025), 1737–1761.
[19] Massimo Tisi et al. 2019. Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. In *STAF 2019 Co-Located Events*.
[20] Mauro Dalle Lucca Tosi and Jordi Cabot. 2025. OCL on Life Support: Can We Revitalize the Community for a Stronger Future?. In *STAF 2025 Workshops (CEUR Workshop Proceedings, Vol. 4122)*. CEUR-WS.org.

A Typing and Evaluation Rules of ReOCL

Figures 6 and 7 present the typing and evaluation rules of ReOCL.

[12] https://rocq-prover.org/

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}}\ \text{T-True} \qquad \frac{}{\Gamma \vdash \text{false} : \text{Bool}}\ \text{T-False} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \text{Int}}\ \text{T-Int} \qquad \frac{s \in \mathcal{S}}{\Gamma \vdash \text{"}s\text{"} : \text{String}}\ \text{T-String} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-Var}$$

$$\frac{\text{self} : \text{Object}(C) \in \Gamma}{\Gamma \vdash \text{self} : \text{Object}(C)}\ \text{T-Self} \qquad \frac{\Gamma \vdash e : \text{Object}(C) \quad f : \tau \in \text{fields}(C)}{\Gamma \vdash e.f : \tau}\ \text{T-Nav} \qquad \frac{\Gamma \vdash e : \text{Object}(C) \quad f : \tau \in \text{fields}(C)}{\Gamma \vdash e.f@\text{pre} : \tau}\ \text{T-Pre}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int} \quad \oplus \in \{+, -, *, /\}}{\Gamma \vdash e_1 \oplus e_2 : \text{Int}}\ \text{T-Arith}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int} \quad \oplus \in \{<, >, <=, >=\}}{\Gamma \vdash e_1 \oplus e_2 : \text{Bool}}\ \text{T-Ord} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \oplus \in \{=, <>\}}{\Gamma \vdash e_1 \oplus e_2 : \text{Bool}}\ \text{T-Eq}$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \text{Bool} \quad \oplus \in \{\text{and}, \text{or}, \text{implies}, \text{xor}\}}{\Gamma \vdash e_1 \oplus e_2 : \text{Bool}}\ \text{T-Bool} \qquad \frac{\Gamma \vdash e : \text{Bool}}{\Gamma \vdash \text{not } e : \text{Bool}}\ \text{T-Not}$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif} : \tau}\ \text{T-If} \qquad \frac{\Gamma \vdash e_1 : \text{Collection}(\tau) \quad \Gamma, x : \tau \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \to \text{select}(x \mid e_2) : \text{Collection}(\tau)}\ \text{T-Select}$$

$$\frac{\Gamma \vdash e_1 : \text{Collection}(\tau) \quad \Gamma, x : \tau \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \to \text{reject}(x \mid e_2) : \text{Collection}(\tau)}\ \text{T-Reject} \qquad \frac{\Gamma \vdash e_1 : \text{Collection}(\tau_1) \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \to \text{collect}(x \mid e_2) : \text{Collection}(\tau_2)}\ \text{T-Collect}$$

$$\frac{\Gamma \vdash e_1 : \text{Collection}(\tau) \quad \Gamma, x : \tau \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \to \text{forAll}(x \mid e_2) : \text{Bool}}\ \text{T-ForAll} \qquad \frac{\Gamma \vdash e_1 : \text{Collection}(\tau) \quad \Gamma, x : \tau \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \to \text{exists}(x \mid e_2) : \text{Bool}}\ \text{T-Exists} \qquad \frac{\Gamma \vdash e_1 : \text{Collection}(\tau) \quad \Gamma, x : \tau \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \to \text{one}(x \mid e_2) : \text{Bool}}\ \text{T-One}$$

$$\frac{\Gamma \vdash e_1 : \text{Collection}(\tau_1) \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \to \text{isUnique}(x \mid e_2) : \text{Bool}}\ \text{T-IsUnique} \qquad \frac{\Gamma \vdash e : \text{Collection}(\tau)}{\Gamma \vdash e \to \text{size}() : \text{Int}}\ \text{T-Size} \qquad \frac{\Gamma \vdash e : \text{Collection}(\tau)}{\Gamma \vdash e \to \text{isEmpty}() : \text{Bool}}\ \text{T-IsEmpty}$$

$$\frac{\Gamma \vdash e : \text{Collection}(\tau)}{\Gamma \vdash e \to \text{notEmpty}() : \text{Bool}}\ \text{T-NotEmpty} \qquad \frac{\Gamma \vdash e : \text{Collection}(\text{Int})}{\Gamma \vdash e \to \text{sum}() : \text{Int}}\ \text{T-Sum} \qquad \frac{\Gamma \vdash e : \text{Object}(C')}{\Gamma \vdash e \to \text{oclIsKindOf}(C) : \text{Bool}}\ \text{T-OclIsKindOf}$$

$$\frac{\Gamma \vdash e : \text{Object}(C')}{\Gamma \vdash e \to \text{oclIsTypeOf}(C) : \text{Bool}}\ \text{T-OclIsTypeOf}$$

Figure 6: Typing rules of ReOCL.

$$\frac{[\![e]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VObj}(oid, C))}{[\![e.f]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\sigma(\text{fieldStateId}(C, oid, f)))}\ \text{E-Nav} \qquad \frac{[\![e]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VObj}(oid, C))}{[\![e.f@\text{pre}]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \sigma_{\text{pre}}(\text{fieldStateId}(C, oid, f))}\ \text{E-Pre}$$

$$\frac{[\![e_1]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VFalse})}{[\![e_1 \text{ and } e_2]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VFalse})}\ \text{E-And-False} \qquad \frac{[\![e_1]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VTrue}) \quad [\![e_2]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(v) \quad \text{expectBool}(v) = \text{Some}(b)}{[\![e_1 \text{ and } e_2]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{boolVal}(b))}\ \text{E-And-True}$$

$$\frac{[\![e_1]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VColl}(vs)) \quad \forall v \in vs.\ [\![e_2]\!](\gamma[x \mapsto v], \sigma, \sigma_{\text{pre}}) = \text{Some}(r_v), \quad r_v \in \{\text{VTrue}, \text{VFalse}\}}{[\![e_1 \to \text{select}(x \mid e_2)]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VColl}(\{v \in vs \mid r_v = \text{VTrue}\}))}\ \text{E-Select}$$

$$\frac{[\![e]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VColl}(vs))}{[\![e \to \text{size}()]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VInt}(|vs|))}\ \text{E-Size} \qquad \frac{[\![e]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{VColl}(vs))}{[\![e \to \text{isEmpty}()]\!](\gamma, \sigma, \sigma_{\text{pre}}) = \text{Some}(\text{boolVal}(vs = []))}\ \text{E-IsEmpty}$$

Figure 7: Representative denotational evaluation rules of ReOCL.