VitruviusOCL: A Declarative Language for Multi-Model Consistency Specification

Max Oesterle
Karlsruhe Institute of Technology
Karlsruhe, Germany
uiysg@student.kit.edu

Arne Lange
Karlsruhe Institute of Technology
Karlsruhe, Germany
arne.lange@kit.edu

Nathan Hagel
Karlsruhe Institute of Technology
Karlsruhe, Germany
nathan.hagel@kit.edu

Terru Stübinger
Karlsruhe Institute of Technology
Karlsruhe, Germany
stuebinger@kit.edu

Ralf Reussner
Karlsruhe Institute of Technology
Karlsruhe, Germany
ralf.reussner@kit.edu

Anne Koziolek
Karlsruhe Institute of Technology
Karlsruhe, Germany
koziolek@kit.edu

Abstract

Large cyber-physical systems normally need to be described using multiple domain-specific modeling languages, so as to allow stakeholders to focus on their respective concerns. However, that multiplication of languages comes at the cost of fragmenting the overall system description, with the inherent consequences of the redundant representation of information and the higher likelihood of cross-model inconsistency. Unfortunately, in existing declarative constraint languages for multi-model environments, the explicit correspondence structures to link views across metamodel boundaries are not addressable as first-class primitives. Consequently, those languages also cannot express invariants that span these correspondence structures.

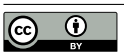
We present VitruviusOCL, a declarative constraint language and toolchain for view-based modeling built on the formal foundation of OCL$^\sharp$ [23] and the Virtual Single Underlying Model concept of the Vitruvius framework [11]. VitruviusOCL introduces the ~ operator as a first-class, axiomatic primitive for accessing the Vitruvius Correspondence Model, together with an explicit, metamodel-qualified navigation syntax for heterogeneous multi-model access, thereby enabling declarative, genuinely n-ary cross-metamodel constraints. VitruviusOCL is delivered as a full prototype. We evaluate VitruviusOCL in a qualitative feasibility study across two case studies representing diverse domains. We demonstrate that VitruviusOCL can express multi-model constraints and specify intended behavior during design space exploration prior to identifying a concrete repair action, thereby making it a valuable part of the engineering process.

CCS Concepts

• **Software and its engineering** → Formal software verification.

Keywords

Object Constraint Language, Model-Driven Development, Model-Driven Engineering, Consistency, Correspondence



MODELS Companion 2026, Málaga, Spain

ACM ISBN 979-8-4007-2903-4/2026/10
https://doi.org/10.1145/3837062.3838896

ACM Reference Format:
Max Oesterle, Arne Lange, Nathan Hagel, Terru Stübinger, Ralf Reussner, and Anne Koziolek. 2026. VitruviusOCL: A Declarative Language for Multi-Model Consistency Specification. 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/10.1145/3837062.3838896

1 Introduction

The complexity of modern systems continues to grow, involving a large number of stakeholders [17]. View-based modeling [1] addresses this challenge by allowing different stakeholders to work with specialized, concern-specific views of a system. However, since each view typically corresponds to a dedicated modeling language, system knowledge becomes distributed across heterogeneous models. This distribution introduces redundancies in the system description and, more critically, gives rise to inconsistencies between models [11]. This often leads to redundant representations of the same information across viewpoints, which in turn raises the problem of inconsistencies among models or views [9]. Such inconsistencies are critical in valid product design, for example in agile systems engineering.

Given that a single system is typically described through a multitude of heterogeneous models and views, maintaining consistency among all of them throughout the system's lifecycle becomes a central challenge. Undetected inconsistencies may surface only during late integration stages, where resolving them requires substantially more effort than if caught earlier [9], increasing the cost of both development and maintenance. Consider a construction engineer who increases a brake pad thickness parameter in the corresponding CAD model without propagating the change to the Brake Specification: the CAD model's pad thickness now silently diverges from the specification, and nothing in the toolchain flags the mismatch. Approaches such as Vitruvius [11] aim to address exactly this kind of silently-divergent, framework-unmanaged inconsistency by maintaining an explicit, kept-up-to-date correspondence model as part of an integrated environment based on a *Virtual Single Underlying Model* (V-SUM), which relates models through these correspondence structures and provides a unified conceptual backbone for view-based modeling.

In the current state of the Vitruvius framework, consistency is primarily preserved through imperative *Reactions* for change-driven

transformations [10, 14] which are thereby enabled to propagate changes across related models by reacting to modifications and updating corresponding elements. While *Reactions* support automated consistency preservation at the operational level, there is still no declarative way to define and check invariants over heterogeneous, correspondence-linked models. Furthermore, consistency serves as an important heuristic during design space exploration [26], guiding engineers toward suitable design decisions, but *Reactions* are not always applicable here: they require a repair action to be specified in advance, whereas at the outset of an exploration, suitable repairs are often not yet known. Vitruvius's constraint language, OCL, adds a further limitation: the OCL standard does not define any notion of correspondence between models [18], so OCL is restricted to a single-metamodel context and cannot natively navigate across models via correspondence links, that is, it is inherently *correspondence-unaware*. That leaves developers without a lightweight, declarative means to specify and validate consistency across heterogeneous models by navigating correspondences directly, that is, in a *correspondence-aware* manner.

In this paper, we introduce *VitruviusOCL*, a declarative constraint language for multi-model environments like the Vitruvius framework. VitruviusOCL extends OCL with correspondence-aware navigation semantics, enabling developers to formulate consistency constraints across heterogeneous models. In detail, our contributions are as follows: **(C1): Declarative correspondence-aware constraint language.** We define a conservative extension of OCL$^\sharp$ that introduces qualified cross-metamodel navigation and a first-class correspondence operator, enabling the specification of type-safe invariants over heterogeneous models without requiring an explicit merge of metamodels or models. **(C2): Implementation and tooling.** We provide a full prototype of VitruviusOCL, comprising an ANTLR4-based parser, a static type checker, and an LSP plugin for VS Code. We evaluate the approach in a qualitative feasibility study across two case studies from distinct domains. **(C3): Constraint annotation extensions.** We extend the invariant syntax with optional `@severity` and `@message` annotations, providing a five-level severity classification and template-interpolated diagnostic messages that address known shortcomings of OCL identified by Kolovos et al. [13].

The paper is organized as follows. Section 2 introduces our running example, and Section 3 presents the necessary foundations. Section 4 describes the core language features of VitruviusOCL. Section 5 covers the implementation and tooling. Section 6 presents the feasibility evaluation. Section 7 discusses related work, and Section 8 concludes the article, with a detailed account of future work.

2 Running Example

Our running example involves a complex engineering process where heterogeneous models overlap semantically, requiring coordination across different organizational roles, engineering environments, and design artifacts. We adapt an established multidisciplinary case study [6] centered around the development and validation of an automotive brake system. Compared to the original case study, we replace the Validation Engineer role and its associated test-bench-based validation step with a Software Engineer role and the AUTOSAR metamodel. All other roles and artifacts are adopted unchanged.

2.1 Involved Engineering Roles

The development and validation lifecycle of the brake system is driven by four core stakeholder roles, each operating within distinct frameworks and utilizing specialized standalone applications: a **Product Manager** who defines high-level physical requirements and safety-critical attributes within a declarative Brake Specification Model, a **Construction Engineer** who translates specification requirements into precise 3D geometric shapes using CAD software, a **Software Engineer** who designs the electronic control units and embedded control algorithms within the AUTOSAR toolchain, and a **Simulation Engineer** who models and evaluates behavioral execution logic under virtual driving scenarios using Simulink.

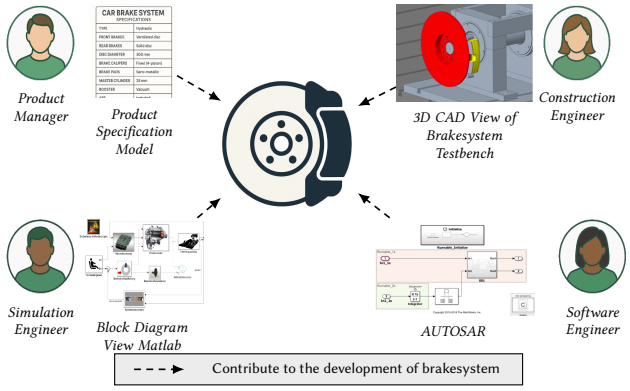


Figure 1: Overview of involved roles and views on the brake system and respective models, adapted from [6].

2.2 Heterogeneous Domain Metamodels

The running example spans four domain-specific metamodels, each capturing a distinct engineering perspective of the brake system. The *Brake Specification Metamodel* formalizes the physical component structure from a requirements standpoint: a `BrakeSystem` aggregates typed `BrakeComponent` instances, concretely `BrakeDisk`, `BrakeCaliper`, and `BrakePad`, each carrying geometric measurements and component-specific properties such as thickness bounds or wear-warning flags.

The *CAD Metamodel* represents the geometric construction layer, organizing `Part` elements within `Namespace` hierarchies. Typed `Parameter` elements (e.g., `NumericParameter` with an associated `Unit`) serve as stable reference points for dimensional values that must remain consistent across models. The *AUTOSAR Metamodel* captures the component-based software architecture deployed on the vehicle's ECUs, distinguishing between composite software structures (`CompositionSwComponentType`) and atomic executable blocks (`AtomicSwComponentType`) that communicate through typed `PortPrototype` interfaces. Finally, the *Simulink Metamodel* describes the behavioral layer, representing signal-processing logic as

[0]The AUTOSAR illustration is adapted from https://www.mathworks.com/products/autosar.html.

hierarchically nested `Block` and `Subsystem` elements, augmented by `Parameter` objects for calibration variables whose values are subject to multi-model consistency requirements.

3 Foundations

In this section, we introduce the foundational concepts underlying our approach.

3.1 View-Based Software Development

View-based development addresses the challenges of separation of concerns and the reduction of accidental complexity in large cyber-physical and software-intensive systems. Stakeholders operate on *views*: partial, concern-specific representations that expose only the information relevant to a particular role or task [1].

Each view is classified by a *ViewType* that determines the semantic role of the view and the category of concepts it may contain, and is governed by a *ViewPoint* that encapsulates partial knowledge about the system and domain from a specific perspective [1, 4]. These concepts form the basis for the Vitruvius approach discussed next.

3.2 The Vitruvius Framework and the Virtual Single Underlying Model

A naive view-based approach would consolidate all stakeholder concerns into a single metamodel, avoiding redundancy at the cost of domain separation and coordinated evolution. Vitruvius instead introduces the *Virtual Single Underlying Model (V-SUM)* [11], which integrates multiple models conforming to distinct metamodels into a unified model space. Consistency specifications govern change propagation and maintain consistency across model boundaries.

Definition 3.1 (V-SUM, [11]). Let $\langle M_{\text{VSUM}} \rangle \in I_{M_1} \times \cdots \times I_{M_n}$ be a tuple of models conforming to metamodels $M_1, \ldots, M_n$, and let VSUMM be the corresponding V-SUM metamodel. A *virtual SUM (V-SUM)* is a structure

$$\text{VSUM} := \left(\langle M_{\text{VSUM}} \rangle, \text{VSUMM}\right)$$

such that the tuple of models $\langle M_{\text{VSUM}} \rangle$ is consistent with respect to the consistency rules $\langle CR_{\text{VSUMM}} \rangle$ defined by the V-SUM metamodel.

The V-SUM metamodel provides a *change application function* `App`, which applies a change to the VSUM and computes a new, consistency-preserving state of the model tuple. Applying changes exclusively through `App` ensures that every reachable state remains consistent, making a V-SUM behave like an ordinary SUM free of inconsistencies.

3.3 The Object Constraint Language

The Object Constraint Language (OCL) [18] is the standard declarative language for specifying invariants over models. Defined by the Object Management Group as a companion to UML, OCL provides a side-effect-free expression language.

OCL constraints are evaluated in the context of a specific metaclass, and the `self` reference denotes the instance of that class being validated.

```
context BrakeDisk
    inv: self.diameterInMM > 0
```

Navigation expressions traverse structural references defined in the enclosing metamodel. The language supports a type hierarchy, including primitive types, classifiers from the subject metamodel, and parameterized collection types (`Set`, `Bag`, `Sequence`, `OrderedSet`), all of which carry statically verifiable type information.

3.4 OCL Limitations

Although the Object Constraint Language (OCL) is the de facto standard for declarative constraint specification in model-driven engineering, its design exhibits several limitations when applied to multi-metamodel settings such as Vitruvius.

First, OCL is inherently *single-metamodel scoped*. Expressions are evaluated within the context of a single metamodel, and type references are resolved only within that metamodel's namespace. Consequently, OCL cannot express constraints that span independently defined metamodels.

Second, OCL provides no mechanism for associating violated invariants with severity levels or human-readable diagnostic messages. As a result, all violations are reported uniformly, requiring developers to manually determine both their cause and importance. Kolovos et al. [13] identify this lack of meaningful feedback as a major practical limitation of OCL.

Third, OCL is *not type-safe*. Its semantics include the special values `null` and `invalid`, allowing expressions to fail at runtime due to undefined navigations or partial expressions. According to the designers of OCL♯, this behavior stems from a *functional/relational impedance mismatch* between OCL's functional expression language and the relational structure of UML-style models [23], making runtime failures common where static type errors would be preferable.

Of these limitations, OCL♯ addresses only the lack of static type safety by redefining OCL's evaluation model while retaining its familiar syntax. The single-metamodel scope and the absence of severity and diagnostic information remain unresolved. VitruviusOCL addresses both directly, as described in Section 4.

3.5 OCL♯: A Type-Safe Relational OCL Dialect

OCL♯ is a relational, type-safe variant of OCL proposed by Steimann et al. [23] to address the *functional/relational impedance mismatch* in OCL: because OCL maps association ends with different multiplicities to fundamentally different expression types (scalars, nullable properties, or collections), navigation expressions must explicitly handle each case, and navigating via a `null` value raises an `invalid` error at runtime. OCL♯ resolves this by making *every* expression evaluate to a collection. Multiplicity 1 becomes the singleton type ¡*C*!, multiplicity 0..1 the nullable type¿*C*?, and multiplicity $*$ an ordinary set or sequence. Navigation is then uniform across all multiplicities: accessing a property on an empty receiver yields *[ ]*, which serves as the uniform representation of "no object" regardless of multiplicity, eliminating both `null` and `invalid` from the language entirely. Since every expression in OCL♯ evaluates to a collection, the distinction between scalar navigation (`.`) and collection navigation (`->`) becomes meaningless; OCL♯ therefore uses `.` as the sole navigation operator.

Every typed entity in OCL$^\sharp$ carries a *ctype* $\chi = \tau^{[l,r]}_{(\mu,\omega)}$, combining a member type τ, a multiplicity interval $[l, r]$, a uniqueness designator μ, and an orderedness designator ω. Throughout this paper, we use the shorthands ¡τ! for $\tau^{[1,1]}_{(\bot,\bot)}$ (exactly one value), ¿τ? for $\tau^{[0,1]}_{(\bot,\bot)}$ (zero or one value), and τ for $\tau^{[0,*]}_{(u,\bar{o})}$ (a set of values). Typing judgements have the form $\mathcal{M}_1; \Gamma \vdash e : \chi$. The key rules used in this paper are **all instances**, **property access**, and **select**. **All instances** retrieves all direct and indirect instances of a class as $\{C\}$. **Property access** $e.p$ navigates from a collection of objects to the values of their property p (navigating from an empty receiver returns the empty collection *[ ]*. **Select** filters a collection by a boolean predicate, reducing the lower multiplicity bound to 0 [23].

The central result of OCL$^\sharp$ is its *safety theorem* [23]: if $\mathcal{M}_1 \vdash \mathcal{M}_0$, i.e., model $\mathcal{M}_0$ conforms to its metamodel $\mathcal{M}_1$, and $\mathcal{M}_1; \Gamma \vdash e : \chi$, then e always evaluates to a value of ctype χ: no expression can get stuck or produce `invalid`. Type safety is a consequence of well-typedness, established once at compile time.

3.6 Cross-Metamodel Consistency and the Coextension Operator

In the context of Vitruvius, a more fundamental limitation emerges: consistency relations typically span instances of different metamodels, such as a `BrakeDisk` element and the CAD elements representing its geometry. OCL cannot express such relations because navigation is confined to a single metamodel namespace.

Pascual et al. [21] address this limitation through a *coextension operator*. Denoted by $\sim$, the operator defines a relation between elements of different metamodels.

Definition 3.2 (Coextension, adapted from [21]). Let M_1 and M_2 be two disjoint metamodels, and let $a \in M_1$ and $b \in M_2$, and let

$$corr \subseteq \mathcal{I}(a) \times \mathcal{I}(b)$$

be a set of explicitly declared instance-level correspondence links. The *coextension relation* $\sim$ between a and b holds if and only if there exists at least one correspondence link between an instance of a and an instance of b, i.e.,

$$a \sim b \iff \exists\, x \in \mathcal{I}(a),\, y \in \mathcal{I}(b).\ (x, y) \in corr.$$

Conceptually, coextension aligns with the Vitruvius correspondence model [11], as both represent instance-level links between elements of different metamodels. Pascual et al. [21] further show that the cardinality of coextension mappings affects the complexity of generated proof obligations, providing a formal argument for integrating correspondence navigation directly into the constraint language.

4 The VitruviusOCL Language Features

VitruviusOCL conservatively extends OCL$^\sharp$ with three mechanisms addressing the limitations identified in Section 3.4 by introducing a qualified cross-metamodel navigation, a coextension operator adapted from [21], and constraint annotations for severity and diagnostic information.

4.1 Qualified Cross-Metamodel Navigation

OCL confines navigation to the metamodel of the enclosing context and therefore cannot reference instances from other metamodels. VitruviusOCL removes this restriction by interpreting $\mathcal{M}_1$ as the VSUM, which aggregates all registered metamodels into a single typing context. To avoid name clashes between metaclasses originating from different metamodels, all type references are qualified as MM::C, where *MM* denotes the metamodel name and *C* the metaclass name.

Definition 4.1 (Multi-Metamodel Static Model). The *multi- metamodel static model* is a combined mapping that assigns to each fully-qualified class name $MM::C$ its superclass (if any) and to each property name $MM::C.p$ its ctype χ similar to [23].

Two metaclasses $MM_1::C$ and $MM_2::C$ sharing the same local name C are *distinct types* whenever $MM_1 \neq MM_2$. Subtyping is only defined within the same metamodel: $MM::C_1 <: MM::C_2$ iff $\mathcal{M}_1$ records $MM::C_2$ as a transitive superclass of $MM::C_1$. Cross-metamodel subtyping does not exist, even if two metaclasses share the same local name.

The `allInstances()` typing rule from OCL$^\sharp$ lifts directly to qualified names:

$$\frac{MM::C \ \in \ dom(\mathcal{M}_1)}{\mathcal{M}_1 \ \vdash \ [\![\texttt{all}\ MM::C]\!] \ : \ \{MM::C\}}$$

The type checker enforces that *MM* resolves to a registered metamodel and that *C* is declared within it, so foreign-type references are validated statically.

The following extract from the running example illustrates cross-metamodel navigation. The constraint retrieves all `Namespace` instances from the `cad` metamodel, selects those whose name matches `self.componentId`, filters their parameters for `Coordinate` instances, and checks that their x-coordinate does not exceed the disk radius:

```
context brakesystem::BrakeDisk inv:
  cad::Namespace.allInstances()
    .select(n | n.name == self.componentId).parameters
    .select(p | p.oclIsTypeOf(cad::Coordinate))
    .forAll(p | p.oclAsType(cad::Coordinate).x
    <= self.diameterInMM / 2)
```

The expression `cad::Namespace.allInstances()` has type `cad :: Namespace`, preserving metamodel provenance throughout navigation.

The MM::C notation separates two concerns that are conflated in the original coextension approach of Pascual et al. [21]. While Featherweight OCL uses $\sim$ for both cross-metamodel access and correspondence lookup, VitruviusOCL delegates the former to MM::C and reserves $\sim$ exclusively for correspondence queries.

4.2 The Coextension Operator

While qualified `allInstances()` provides unrestricted access to foreign types, many consistency invariants concern only corresponding elements. VitruviusOCL adopts the coextension concept from Pascual et al. [21] and extends it with formal OCL$^\sharp$ typing rules, tag and type filtering, and a primitive `~` operator implemented via the Vitruvius Correspondence API.

The Correspondence API. The Vitruvius framework manages correspondences internally via a dedicated Ecore metamodel (`CorrMM`), whose relevant structure is:

Metaclass	Property	Ctype
`CorrMM::Correspondence` (abstract)	`leftEObjects`	{*any*}
	`rightEObjects`	{*any*}
	`tag`	¿*String*?
`CorrMM::ManualCorrespondence`	extends `CorrMM::Correspondence`	

Unlike the domain metamodels of the case study (cad, autosar, simulink), CorrMM is not registered as a standalone EPackage in the VSUM. The qualified navigation scheme MM::C from Section 4.1, however, relies precisely on this registration, since it requires the type checker to resolve MM against a known metamodel in M_1. Correspondence instances are therefore not reachable through regular qualified navigation in the first place; access to CorrMM is instead provided exclusively through the dedicated Vitruvius Correspondence API at runtime.
The ~ operator is consequently the sole means by which a constraint can reach correspondence information. Its evaluation is dispatched at runtime via the Vitruvius Correspondence API, which abstracts over the internal representation of `CorrMM` and keeps constraint syntax stable across implementation choices.

Motivating Example. Consider the brake system case study introduced in Section 2. A consistency invariant requiring that every `AUTOSAR::Signal` corresponding to a `Simulink::OutPort` carries a matching data type can be expressed as:

```
context AUTOSAR::Signal
inv: Simulink::OutPort.allInstances()
       .select(p | p ~ self)
       .forAll(p | p.dataType = self.dataType)
```

The ~ operator in the `.select` filters exactly those `OutPort` instances that Vitruvius has recorded as corresponding to the current `Signal`. Because this pattern occurs frequently in practice, VitruviusOCL provides `.select(~)` as a shorthand for `.select(x | x ~ self)`. Throughout the remainder of this paper, we use this shorthand wherever the full binary form adds no clarity.

Operator Definition and Typing Rule. In VitruviusOCL, the abstract correspondence relation *corr* of Definition 3.2 is realised by the Vitruvius correspondence model: the set of pairs (a, b) such that some `CorrMM::Correspondence` instance c satisfies c.`leftEObjects`.`includes`(a) and c.`rightEObjects`.`includes`(b). Following Pascual et al. [21], this is captured by the desugaring:

$$a \sim b \;\equiv_{def}\; \texttt{CorrMM::Correspondence.allInstances()}.\texttt{exists}(c \mid c.\texttt{leftEObjects.includes}(a) \text{ and } c.\texttt{rightEObjects.includes}(b))$$

This desugaring is an OCL$^\sharp$ translation of the coextension definition of Pascual et al. [21], which in Featherweight OCL grounds ~ in an explicit set of correspondence pairs. The ~ operator is consequently the sole means by which a constraint can reach correspondence information.

Filter Forms. Two derived notations extend the binary operator with additional filtering. Both are syntactic sugar over ~ and expand exactly one level, to expressions involving ~ and standard OCL$^\sharp$ operations:
Without filter. For a receiver expression E:

$$E\texttt{.select(~)} \;\equiv\; E\texttt{.select}(x \mid x \sim \texttt{self})$$

Type filter. For a receiver expression E and a classifier C:

$$E\texttt{.select(~, Type = C)} \;\equiv\; E\texttt{.select}(x \mid x \sim \texttt{self and } x\texttt{.oclIsKindOf}(C))$$

Tag filter. For a receiver expression E and a tag literal t:

$$E\texttt{.select(~, Tag = } t\texttt{)} \;\equiv\; E\texttt{.select}(x \mid x \sim \texttt{self and corrTag}(x, \texttt{self}, t))$$

where `corrTag` is a primitive predicate of type ¡*bool*! supplied by the Correspondence API, returning `true` if the correspondence entry linking x and `self` carries the tag t.

Type Safety.

Proposition 4.2 (Type Preservation of ~). *Let E be a well-typed expression with $\mathcal{M}_1; \Gamma \vdash E : \tau^{[l,r]}_{(\mu,\omega)}$ where $\mu \in \{u, \bar{u}\}$ and $\omega \in \{o, \bar{o}\}$, that is, E has a collection ctype rather than a singleton or nullable ctype. Then E.`select`$(x \mid x \sim \texttt{self})$, as well as both filter forms, are well-typed with result ctype $\tau^{[0,r]}_{(\mu,\omega)}$.*

Proof. Consider the base case E.`select`$(x \mid x \sim \texttt{self})$. The predicate body $x \sim \texttt{self}$ is well-typed by the typing rule of ~: $\mathit{scalar}(\mathit{mtype}(\chi_x))$ holds because E is a flat collection by assumption ($\mu \in \{u, \bar{u}\}$ and $\omega \in \{o, \bar{o}\}$ excludes singleton and nullable ctypes), and $\mathit{scalar}(\mathit{mtype}(\chi_{\texttt{self}}))$ holds because `self` is bound to an object of the invariant context class. The predicate body therefore has type ¿*bool*?. The indirect translation rule for `.select` of [23] assigns the result ctype $\tau^{[0,r]}_{(\mu,\omega)}$, reducing only the lower bound to 0. The Safety Theorem of [23] then applies, given that $\mathcal{M}_1; \Gamma \vdash E : \tau^{[l,r]}_{(\mu,\omega)}$ and $\mathcal{M}_1$ is well-typed with respect to $\mathcal{M}_0$.

For the filter forms, the expansions introduce an additional conjunct in the predicate body. The type filter adds x.`oclIsKindOf`(C), which has type ¡*bool*! by the standard OCL$^\sharp$ rule for `oclIsKindOf`. The tag filter adds `corrTag`(x, `self`, t), which has type ¡*bool*! by the axiom for `corrTag`. In both cases, the conjunction of ¿*bool*? with ¡*bool*! yields ¿*bool*?: since the nullable type subsumes the singleton type under the subtype relation of [23] (i.e., ¡*bool*! <: ¿*bool*?), the least upper bound of the two conjunct types is ¿*bool*?, so the predicate body type is unchanged and the `.select` rule applies identically. □

Together, qualified cross-metamodel navigation and the coextension operator equip VitruviusOCL with the means to formulate consistency constraints across heterogeneous model boundaries in a concise and type-safe manner. Section 6 shows both constructs at work in the brake system case study.

4.3 Constraint Annotations

The mechanisms introduced so far determine only whether a constraint is satisfied. Kolovos et al. [13] identify the absence of severity classifications and meaningful diagnostic messages as important practical limitations of OCL. These limitations are addressed by VitruviusOCL using two optional invariant annotations (specifically, `@severity` and `@message`), which together constitute our contribution C3.

Syntax and Well-Formedness. Annotations are not OCL♯ expressions and do not participate in the typing relation $\mathcal{M}_1; \Gamma \vdash e : \chi$. They are instead metadata attached to an invariant declaration at parse time and are therefore orthogonal to the type system established in Section 3.5. Formally, an annotated invariant has the surface form

`context` *MM*::*C* `inv` *n*: [`@severity` *s*] [`@message` *m*] *e*

where e is a well-typed constraint expression with $\mathcal{M}_1; \Gamma \vdash e : \chi$ and $single(mtype(\chi)) \Rightarrow mtype(\chi) = bool$, $s \in \mathcal{S}$ is a severity literal, and m is a message template. The presence or absence of the annotations does not affect the derivation of χ. In particular, the well-typedness of e is checked independently of both annotations. Severities are never relevant to the original proof of the Safety Theorem [23], which therefore continues to apply to e unchanged.

Severity Levels. The severity domain $\mathcal{S}$ comprises five totally ordered, reserved keywords:

`INFO` < `WARNING` < `MINOR` < `MAJOR` < `CRITICAL`

When the `@severity` annotation is absent, the default value `WARNING` is assigned. Severities are plain labels, independent of OCL♯'s type system.

Message Templates. A message template m is a string literal supporting single-level interpolation. Let $attrs(C)$ denote the set of attribute names declared on metaclass *MM*::*C*. The set of well-formed interpolation holes is:

$$Interp(C) = \{\texttt{\{self\}}\} \cup \{\texttt{\{self.}a\texttt{\}} \mid a \in attrs(C)\}$$

At evaluation time, each interpolation expression `{self}` is replaced by the string representation of the context object, and each `{self.`a`}` is replaced by the string representation of $\mathcal{M}_0(self.a)$.

Well-formedness of m is checked statically: the type checker verifies that every interpolation expression in m belongs to $Interp(C)$, where C is the context class of the enclosing invariant.

The following excerpt from the case study illustrates both annotations. Since a violation of this constraint has direct safety relevance, it is classified as `CRITICAL` and the message identifies the affected caliper by name:

```
context brakesystem::BrakeCaliper inv cadThickness:
    @severity CRITICAL
    @message "CAD pad thickness exceeds disk
              thickness for {self.componentId}"
    self.brakePads.forAll(pad | ...)
```

Diagnostic Reporting. When an annotated invariant is violated, the resulting diagnostic contains the configured `@severity` classification and the rendered `@message`. Diagnostics are reported in document order and are not ranked or aggregated by severity. The severity assigned at specification time is preserved unchanged in every diagnostic produced by the invariant.

5 Implementation & Tool Support

VitruviusOCL is implemented as a standalone Java library on top of ANTLR4 [20] and the Eclipse Modelling Framework (EMF) [24]. The compiler is structured as a three-pass pipeline operating on an ANTLR4 parse tree.

5.1 MetamodelWrapperInterface: the Runtime Boundary

The central abstraction of the implementation is the `MetamodelWrapperInterface`, which all three compiler passes receive as a constructor argument. Neither the type checker nor the evaluator knows anything about how metamodels are loaded or where model instances live. The interface exposes two categories of operations: metamodel resolution, which maps qualified names such as `cad::Namespace` to EMF `EClass` objects and enumerates their instances at runtime, and correspondence access, which exposes the Vitruvius correspondence model to enable the ~ operator to query inter-model links at constraint-check time. The concrete implementation `VSUMWrapper` delegates directly to a live Vitruvius V-SUM.

5.2 `Pass 1`: Symbol Table and Scope Annotation

`SymbolTableBuilder` is the first ANTLR4 visitor over the parse tree. It registers every `context` declaration, every iterator variable (`select`, `forAll`, etc.), and every `let`-binding in a two-level scope hierarchy (`GlobalScope` / `LocalScope`). Crucially, it annotates parse-tree nodes with their corresponding `Scope` objects via a `ScopeAnnotator`. `Pass 2` retrieves these annotations to enter the same scope rather than rebuilding it. The scope builder is executed as a separate pass so that the type checker has consistent scope information available across the entire tree before any type inference begins.

5.3 `Pass 2`: Type Checking and the Receiver-Type Stack

`TypeCheckVisitor` performs static type inference by traversing the parse tree and storing a `Type` value for each node in a `ParseTreeProp erty<Type>` map. The non-trivial design challenge is
cross-metamodel *navigation chains*: an expression like

```
cad::Namespace.allInstances().select(n | n.name =
                    self.componentId)
```

involves consecutive receiver-type transitions that must be threaded through the visitor without a return-value channel. The solution is a `Deque<Type> receiverStack` maintained in `visitPrimaryWithNav`: each navigation step peeks the current receiver type, computes the result type of the operation, and replaces the stack top accordingly. Iterator variables in `select` and `forAll` are bound to the element type of the current collection, and property access is resolved against the EMF structural feature metadata of the receiver class via the `MetamodelWrapperInterface`.

The stack discipline also handles *implicit collect*: when property navigation is applied to a multi-valued collection receiver, `typeCheckPropertyAccess` automatically wraps the result in a `Set(T)` rather than producing a type error. EMF structural feature metadata (`isOrdered`, `isUnique`) is consulted to choose among `Set`, `Sequence`, `Bag`, and `OrderedSet` automatically.

5.4 `Pass 3`: Evaluation

`EvaluationVisitor` mirrors the structure of the type checker but produces `Value` objects instead of `Type` objects, advancing

through navigation chains at runtime using the same push/pop protocol on a parallel `Deque<Value> receiverStack`. Rather than consulting the metamodel wrapper for type resolution, it calls `getAllInstances(EClass)` to materialise `allInstances()` as a concrete `List<EObject>`, and traverses EMF structural features via `EObject.eGet(EStructuralFeature)` for property access.

Two design decisions are worth noting. First, all values are represented as `Value`, a thin wrapper around `List<Object>`, so the evaluator never special-cases singletons vs. collections. This is the direct implementation counterpart of the OCL$^\sharp$ principle that every expression evaluates to a collection: predicates written for a single element work unchanged on a collection without additional dispatch logic. Second, `Pass 3` runs on the same parse tree already annotated by `Pass 2`, consuming the `ParseTreeProperty<Type>` node-type map directly to resolve type-dependent dispatch. For example, choosing the correct `EStructuralFeature` when property names are overloaded across inherited classes.

Annotation processing is split across `Pass 2` and `Pass 3`. `TypeCheck Visitor` handles `@severity` and `@message` statically: `visitSever ityAnnotation` validates that the severity keyword is one of the five reserved values, and `visitInvCS` rejects duplicate annotations within the same invariant. Both are compile-time checks that produce precise diagnostics without touching the constraint body. `Pass 3` (`EvaluationVisitor`) re-traverses the parse tree directly when an invariant evaluates to `false`: `extractSeverity()` and `extractCustomMessage()` read the `annotationCS()` children of the invariant node, resolve any `{self}` or `{self.attr}` placeholders via `EObject.eGet` on the context instance, and fall back to the literal placeholder text of the attribute that cannot be resolved. `Pass 1` does not participate in annotation processing.

6 Evaluation

This evaluation [19] aims to demonstrate the expressiveness of VitruviusOCL as a declarative constraint language and also its practical utility during early design phases of multi-disciplinary engineering workflows.

Note, our primary contribution in this work is the *language itself* and not any particular runtime optimization. For that reason, we conducted a qualitative feasibility study rather than using performance benchmarks. Our goal is not to measure performance but to show that (1) multi-model constraints spanning several metamodels can be expressed, and also that (2) VitruviusOCL surfaces inconsistencies where detection would require otherwise either a manual inspection across heterogeneous toolchains (which is cost-intensive) or a known repair action (which is normally unavailable in early phases of design-space exploration).

The scenarios used in this evaluation are derived from actual challenges encountered in a Formula Student racing vehicle development project. These scenarios have the advantage of systematically covering the types of qualitatively distinct consistency relationships which arise when different engineering disciplines evolve their models independently.

6.1 Evaluation Setup

The constraints are implemented as automated unit tests operating on the four Ecore metamodels of our running example (Section 2). Each test (a) applies a targeted change to one or more model instances, (b) evaluates the affected VitruviusOCL constraints, (c) verifies that the evaluation reports either *satisfied* or *violated*, and (d) produces a diagnostic carrying a severity level and a human-readable message.

Two scenarios are evaluated. The first concerns a design change to the brake caliper that leaves the Brake Specification Model out of sync with the CAD model. The second concerns an architectural integration in AUTOSAR that is not yet reflected in the Simulink behavioral model. Both scenarios are chosen because they instantiate the same structural condition: a targeted change has been made to one model, the downstream impact across the remaining models is not yet resolved, and no *Reaction* can be authored because the correct repair action has not been determined at this stage of the development process. VitruviusOCL is therefore not evaluated as a verification layer on top of completed *Reactions*, but in the role it is specifically designed for: surfacing inconsistencies before any repair logic exists.

6.2 Scenario 1: Design Change in the Brake Caliper Causes Specification–CAD Inconsistency

The Product Manager decides to switch from a monobloc to a two-piece brake caliper design to improve repairability and reduce cost. The construction Engineer updates the CAD model accordingly, introducing a new `Namespace` named `BrakeCaliper_TwoPiece` with its associated geometry `Parameter` entries and removing the namespace previously representing the monobloc geometry. The Brake Specification Model, however, still contains a `BrakeCaliper` instance carrying the `specificationType` of the old monobloc design. The downstream impact of this change remains partially unresolved: it is not yet clear whether the Brake Specification Model must simply be updated to reflect the new component type, or whether the change requires a broader design iteration touching adjacent parts, mounting interfaces, and the bill of materials. As established in Section 6.1, no *Reaction* can be authored here: writing a rule that blindly updates the `specificationType` in the Brake Specification Model would risk masking deeper structural misalignments that require human judgment to resolve. VitruviusOCL, by contrast, only states what must hold: every `BrakeCaliper` in the Brake Specification Model must correspond to a `Namespace` of matching *name* in the CAD model. The constraint is annotated with `@severity CRITICAL`, because the absence of a corresponding CAD namespace renders impossible any geometric validation of the affected caliper. Now, upon evaluating this constraint after the CAD update, the tool reports a **violation** with severity `CRITICAL` and the diagnostic: *"BrakeCaliper with specificationType 'Monobloc-Caliper' has no corresponding namespace in the CAD model."* The violation is immediately actionable as an engineering signal that the specification and CAD model have diverged, while leaving the resolution to the next design iteration.

6.3 Scenario 2: Integration of ABS and Brake Over Travel Switch Causes Architecture–Behavior Inconsistency

A new brake system is being developed that combines the ABS functionality and the Brake Over Travel Switch into a single integrated software component. The Software Engineer updates the AUTOSAR architecture model accordingly: the previously separate `ApplicationSwComponentType` instances `ABSController` and `BrakeOverTravelSwitchController` are replaced by a single `ApplicationSwComponentType` named `IntegratedBrakeABSController` within the `CompositionSwComponentType` `BrakeControlComposition`, and the `AssemblySwConnector` linking the two former components is removed. The Simulation Engineer, however, continues working with a `SimulinkModel` that still contains two separate `SubSystem` blocks connected by an explicit `SingleConnection` carrying the wheel-speed signal between them.

The simulated brake behavior no longer reflects the actual system architecture, and any validation activities building on the existing simulation rest on outdated assumptions. Restructuring the Simulink model requires its own design iteration before the correct block boundaries and signal topology can be fixed. Propagating the architectural change at this time would require knowing which blocks to merge and how to reconnect the signal paths, information that is only available after the Simulation Engineer has completed the redesign. The following constraint covers these concerns:

```
context autosar::CompositionSwComponentType inv
autosarSimulinkComponentCorrespondence:
  let atomicNames : Set(String) =
    self.components
      .select(proto | proto.type
      .oclIsKindOf(autosar::AtomicSwComponentType))
      .collect(proto | proto.shortName)
      .asSet()
  in
  let simulinkNames : Set(String) =
    simulink::SubSystem.allInstances()
      .select(~, Type = simulink::SubSystem)
      .collect(b | b.name)
      .asSet()
  in
  simulinkNames.includesAll(atomicNames)
```

The constraint is annotated with `@severity MAJOR`, as an unmatched architectural component implies that the simulation no longer reflects the actual system structure, thereby invalidating any behavioral validation based on it. It also shows the multi-model capacity of VitruviusOCL, which enables the formulation of n-ary constraints. After the AUTOSAR change is applied, constraint evaluation reports a **violation** with severity `MAJOR` and the diagnostic: *"BrakeControlComposition: unmatched AUTOSAR atomic components have no corresponding Simulink sub-block."* The diagnostic makes the architectural drift between the two layers immediately legible.

Once the Simulink model is updated to replace the two separate sub-blocks with a single `SubSystem` named `IntegratedBrakeABSController` and the inter-block `SingleConnection` is removed, the constraint returns `satisfied` with no diagnostic output, confirming that the architectural integration is now consistently reflected across both models.

6.4 Discussion

Both scenarios demonstrate that VitruviusOCL fulfills its intended role as a consistency constraint language for heterogeneous multi-domain engineering models. In each case, a targeted change to one model instance produced a correctly classified violation with a meaningful, localized diagnostic; further, the subsequent corrective update to the affected model resolved the constraint to satisfaction, without any repair logic being specified. This supports our core claim about this approach: *Declarative constraints can be evaluated independently of whether a repair action exists.* We conclude, therefore, that declarative constraints are applicable throughout the entire development lifecycle and not only in phases where automated propagation has been designed and implemented.

Crucially, VitruviusOCL is not merely a verification layer on top of *Reactions*, that is, our language is not only a tool for checking whether an already-specified *Reaction* produced the right result. VitruviusOCL fills a genuinely different role in the Vitruvius engineering process: it enables consistency to be expressed and monitored in precisely those situations where *Reactions* cannot yet be fully specified. Both scenarios instantiate exactly this situation: the Construction Engineer's caliper design change and the Software Engineer's architectural integration decision are both made before the downstream engineering discipline has determined how to respond. VitruviusOCL allows these decisions to be registered as open inconsistencies that accumulate and drive the next design iteration, rather than passing unobserved until a later, more expensive stage of validation, making the language the first mechanism in Vitruvius that extends consistency management into the early, exploratory phases of multi-disciplinary design.

The two scenarios further illustrate that the language covers qualitatively different consistency relationships. The first scenario concerns *name-based* multi-model consistency: a component identifier in one metamodel must match a corresponding named element in another. The second scenario concerns *structural* multi-model consistency: the set of named elements in one model must be reflected as a corresponding set of named elements in another. The fact that both relationships can be expressed within the same OCL dialect, navigating across metamodel boundaries through VitruviusOCL's inter-model navigation extension, indicates that the language provides sufficient expressive power for the heterogeneous constraint patterns that arise in practice.

To assess generalizability, we additionally applied VitruviusOCL to another case study based on the original model of the Corona-Warn-App [27], an open-source contact tracing application commissioned by the German government, developed by SAP and Deutsche Telekom during the COVID-19 pandemic, and downloaded more than 48 million times [7]. Its source code and documentation are publicly available, and the system has previously served as a case study in architecture-based analyses [7] and uncertainty analysis [5]. Based on the multi-model environment for the Corona-Warn-App defined in [5], we formulated 34 constraints spanning five metamodels/domains (the Palladio Component Model (PCM), Architecture Decision Records (ADR), C4-model diagrams at the Context and Container levels, and issue tickets). All constraints type-checked successfully and evaluated as expected, exhibiting

the same pattern of correctly classified violations and satisfactions observed in the brake system case study,.

Threats to Validity. The present evaluation is a qualitative feasibility study and is therefore subject to limitations regarding external validity. The change scenarios of the main case study are derived from real engineering problems encountered in a student project, and the constraints reflect consistency requirements that arise when heterogeneous models evolve independently. Nevertheless, the case study remains deliberately small in scope. The Corona-Warn-App case study[27] represents a software system of non-trivial scale and architectural complexity. However, the constraints were manually constructed by the authors to match the available model instances rather than being derived from documented consistency requirements of the original project. This introduces a risk of confirmation bias, as the constraints were written with knowledge of the model structure and may therefore favor consistency relationships that the language can express particularly well. An evaluation based on independently sourced constraints from domain engineers unfamiliar with VitruviusOCL would provide stronger evidence of practical applicability.

7 Related Work

We compare existing approaches to VitruviusOCL along four dimensions: architectural handling of multi-model scenarios, scalability to more than two metamodels, support for correspondence navigation, and declarative versus operational specification.

The Epsilon Validation Language (EVL) [13] addresses several practical shortcomings of OCL, including missing error messages and the absence of severity levels, and enables genuine multi-model access through the Epsilon Model Connectivity layer via the `Model!Type` notation [12]. When applied to Vitruvius, however, EVL offers no native correspondence support: constraints exploiting V-SUM correspondence relationships must manually traverse the correspondence metamodel. VitruviusOCL abstracts this traversal entirely through the ~ operator as a first-class language primitive. On diagnostic feedback, VitruviusOCL adopts EVL's insight that severity levels and violation messages are essential, but provides five ordered severity levels rather than EVL's binary `constraint`/`critique` distinction.

Viatra Query Language (VQL) [28] supports declarative structural queries over EMF models through incremental graph pattern matching and allows querying across multiple metamodels within a single query. For multi-model consistency checking, however, VQL relies on application-specific traceability metamodels [3] that the constraint author must design and maintain separately. The ~ operator of VitruviusOCL instead navigates the V-SUM correspondence model directly, without requiring the constraint author to specify any additional traceability infrastructure.

Stünkel et al. [25] proposes *comprehensive systems* as a formal foundation for multi-model consistency management, in which heterogeneously typed local models are represented as a single global artifact. The approach introduces *commonality witnesses*, graph-like objects formalized in the same presheaf category as the local models, together with partial projection morphisms that map each witness to its corresponding element in the respective model component. Consistency constraints are specified as predicates from the Diagrammatic Predicate Framework (DPF) [22] and evaluated on the comprehensive system directly, with OCL as one tool for constraint formalization via library-level extension methods. Compared to VitruviusOCL, the two approaches differ in how correspondences arise, how they are represented, and how they are accessed in constraints: in comprehensive systems, correspondences are specified declaratively as structural relationships between metamodel elements, mathematically homogeneous with the models they relate, and exposed through library-level helper methods such as `commonalities()`, where correspondence navigation relies on predicate implementations or runtime checks for correctness; in Vitruvius, they are established through *Reactions* and managed as typed instance-level links in an explicit correspondence model. VitruviusOCL instead introduces the ~ operator as a first-class language primitive with qualified cross-metamodel navigation syntax `MM::C`, enabling a static type checker to verify correspondence navigation expressions at compile time. Type safety is thus an intrinsic property of the language rather than a concern delegated to external implementations or runtime checks.

The Deep Object Constraint Language (DOCL) [15] and its host environment Doreen [2] represent an alternative architectural response to the same challenge that motivates VitruviusOCL. Where Vitruvius preserves the independence of existing metamodels and connects them through an explicit correspondence model, Doreen pursues an *essential* Single Underlying Model (SUM) in which all domain knowledge is consolidated into one redundancy-free model from the outset. DOCL provides the constraint layer for this ecosystem, extending OCL with operators for navigating the linguistic and ontological dimensions of a deep model, dimensions that OCL cannot express. It also provides a way to define simplified views as a subset of the SUM [16]. The critical difference to VitruviusOCL lies in the *direction* of that extension: DOCL navigates *vertically* across ontological instantiation levels within a single model space, whereas VitruviusOCL navigates *horizontally* across the boundaries of conceptually independent, co-existing metamodels connected by a correspondence model. These are orthogonal extensions of OCL, each necessary within its respective architectural context, and neither subsumes the other.

8 Conclusion and Future Work

VitruviusOCL is a conservative extension of OCL$^\sharp$ that specifies type-safe invariants over heterogeneous models without requiring an explicit merge of metamodels or models. Because its semantics depend only on the host framework's correspondence model, VitruviusOCL is not inherently tied to the Vitruvius V-SUM and is, in principle, applicable to any heterogeneous metamodel environment exposing a comparable correspondence API. Our feasibility evaluation across two case studies from distinct domains confirms that this design generalizes across qualitatively different correspondence structures.

Future work on VitruviusOCL proceeds in four directions. **(1)** We plan a performance analysis to assess how VitruviusOCL scales with the size of M_1. **(2)** Constraint evaluation currently yields a boolean result per invariant with an optional severity level and message, but does not trace elements transitively affected through

the correspondence model. We plan to address this via causal diagnostics and a low-code authoring interface for diagram-based modeling environments. **(3)** *Reactions* specify operationally how changes are propagated, while VitruviusOCL constraints specify declaratively what consistency properties must hold; a violated constraint signals an inconsistent VSUM state but cannot restore it. We see two complementary directions: deriving constraints from existing *Reactions*, and synthesizing *Reactions* from a given constraint set. This separation of detection and repair is particularly relevant during design space exploration [26], where models are deliberately kept in transient, incomplete states while comparing alternatives and a premature repair would commit to one resolution too early. **(4)** Adapting the LLM4MTLs workflow [8] to VitruviusOCL is a natural next step: as a low-resource domain-specific language, it is unlikely to be well covered by general-purpose training data, and the ANTLR grammar, case studies, and test suite presented in this paper already constitute a few-shot corpus for such prompting. Beyond generation for VitruviusOCL itself, constraints capturing structural invariants of source and target metamodels could semantically prune invalid transformation candidates before execution, complementing grammar-constrained decoding [8] at the semantic rather than the syntactic level.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263 and by the Topic Engineering Secure Systems of the Helmholtz Association (HGF) and supported by KASTEL Security Research Labs, Karlsruhe. This research was edited by our textician, Daniel Shea.

References

[1] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2010. Orthographic Software Modeling: A Practical Approach to View-Based Development. In *Evaluation of Novel Approaches to Software Engineering*, Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski (Eds.). Springer, Berlin, Heidelberg, 206–219. doi:10.1007/978-3-642-14819-4_15

[2] Colin Atkinson and Christian Tunjic. 2021. A deep view-point language and framework for projective modeling. *Information Systems* 101 (Nov. 2021), 101440. doi:10.1016/j.is.2019.101440

[3] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. 2015. VIATRA 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations*, Dimitris Kolovos and Manuel Wimmer (Eds.). Springer International Publishing, Cham, 101–110. doi:10.1007/978-3-319-21155-8_8

[4] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, Ludwik Finkelstein, and Michael Goedicke. 1992. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering* 02, 01 (March 1992), 31–57. doi:10.1142/s0218194092000038

[5] Nathan Hagel, Johannes Mäkelburg, Martin Armbruster, Bowen Jiang, Benedikt Jutz, Lars König, Arne Lange, Fabian Reinbold, Robin Maisch, Thomas Weber, Maribel Acosta, and Anne Koziolek. 2026. UncertainTree: Analyzing Multi-Model Consistency under Uncertainty. In *2026 ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS)*.

[6] Nathan Hagel, Johannes Mäkelburg, Claus Hammann, Thomas Weber, Thomas Alexander Völk, Francesco Urbano, Patrick Grycz, Katharina Bause, Minakshi Kaushik, Vincenzo Scotti, Akhila Bairy, Maike Schwammberger, Maribel Acosta, Albert Albers, Anne Koziolek, and Tobias Düser. 2025. Explainability in Automated Cross-Domain Model-Driven Brake System Development. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 204–211. doi:10.1109/ASEW67777.2025.00047

[7] Sebastian Hahner, Robert Heinrich, and Ralf Reussner. 2023. Architecture-Based Uncertainty Impact Analysis to Ensure Confidentiality. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 126–132. doi:10.1109/SEAMS59076.2023.00026

[8] Bowen Jiang, Nathan Hagel, Haowei Cheng, Benedikt Jutz, Arne Lange, Weixing Zhang, Rahul Sharma, Ralf Reussner, and Anne Koziolek. 2026. LLM4MTLs: Automated Generation and Empirical Evaluation of Model Transformation Languages. *Journal of Object Technology* (2026). doi:10.5445/IR/1000194182

[9] Robbert Jongeling, Federico Ciccozzi, Jan Carlson, and Antonio Cicchetti. 2022. Consistency Management in Industrial Continuous Model-Based Development Settings: A Reality Check. *Software and Systems Modeling* 21, 4 (Aug. 2022), 1511–1530. doi:10.1007/s10270-022-01000-5

[10] Heiko Klare. 2022. *Building Transformation Networks for Consistent Evolution of Interrelated Models*. KIT Scientific Publishing.

[11] Heiko Klare, Max Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. 2021. Enabling Consistency in View-Based System Development — The Vitruvius Approach. *Journal of Systems and Software* 171 (Jan. 2021), 110815. doi:10.1016/j.jss.2020.110815

[12] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2006. The Epsilon Object Language (EOL). In *Model Driven Architecture — Foundations and Applications*, Arend Rensink and Jos Warmer (Eds.). Springer, Berlin, Heidelberg, 128–142. doi:10.1007/11787044_11

[13] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2009. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Rigorous Methods for Software Construction and Analysis*, Jean-Raymond Abrial and Uwe Glässer (Eds.). Springer, Berlin, Heidelberg, 204–218. doi:10.1007/978-3-642-11447-2_13

[14] Max Emanuel Kramer. 2017. Specification Languages for Preserving Consistency between Models of Different Languages. doi:10.5445/IR/1000069284

[15] Arne Lange. 2023. *Deep Multi-Style, Multi-Pattern Modeling with DOCL*. Ph. D. Dissertation. Universität Mannheim, Mannheim, Germany.

[16] Arne Lange, Colin Atkinson, and Christian Tunjic. 2020. Simplified view generation in a deep view-based modeling environment. In *International Conference on Systems Modelling and Management*. Springer, 163–179.

[17] Stephan Murer and Bruno Bonati. 2010. *Managed Evolution: A Strategy for Very Large Information Systems*. Springer Science & Business Media.

[18] Object Management Group. 2014. About the Object Constraint Language Specification Version 2.4. OMG Standard. https://www.omg.org/spec/OCL/

[19] Max Oesterle, Arne Lange, Nathan Hagel, Terru Stübinger, Ralf Reussner, and Anne Koziolek. 2026. *VitruviusOCL: A Declarative Language for Multi-Model Consistency Specification – Replication Package*. doi:10.5281/zenodo.21114933

[20] Terence Parr. 2014. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, Dallas, TX.

[21] Romain Pascual, Arne Lange, Thomas Weber, Lars König, Michael Kirsten, and Terru Stübinger. 2025. Towards Examining the Complexity of Consistency. In *Proceedings of the ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Institute of Electrical and Electronics Engineers, 663–672. doi:10.1109/MODELS-C68889.2025.00091

[22] Adrian Rutle. 2014. *Diagram Predicate Framework: A Formal Approach to MDE*. Ph. D. Dissertation. University of Bergen.

[23] Friedrich Steimann, Robert Clarísó, and Martin Gogolla. 2025. Meet OCL$^{\#}$, a Relational Object Constraint Language. *Software and Systems Modeling* 24, 6 (Dec. 2025), 1737–1761. doi:10.1007/s10270-025-01286-1

[24] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2nd ed.). Addison-Wesley Professional, Boston, MA.

[25] Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. 2021. Comprehensive Systems: A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing* 33, 6 (Dec. 2021), 1067–1114. doi:10.1007/s00165-021-00555-2

[26] Ciprian Teodorov, Joeri Exelmans, Salvador Martinez, Sylvain Guérin, and Hans Vangheluwe. 2026. The Design Multiverse: A Scientific Model for Design Evolution and Co-evolution. In *International Conference on Software Engineering, NIER Track*. Rio de Janeiro (Brazil), France. doi:10.1145/3786582.3786830

[27] The Federal Government of Germany. 2021. Corona-Warn-App. https://www.bundesregierung.de/breg-de/corona-warn-app-faq-1758636.

[28] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling* 15, 3 (2016), 609–629. doi:10.1007/s10270-016-0530-4