

Deep Reactions: Realizing and Evaluating Consistency Preservation for Multi-Level, Multi-View Models

Annika Kienle
Karlsruhe Institute of Technology
Karlsruhe, Germany
annika.kienle@student.kit.edu

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

Thomas Weber
KASTEL - Dependability of
Software-intensive Systems
Karlsruhe Institute of Technology
Karlsruhe, Germany
thomas.weber@kit.edu

Lars König
Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
lars.koenig@kit.edu

Nathan Hagel
Department of Informatics
Karlsruher Institut für Technologie
Karlsruhe, Germany
nathan.hagel@kit.edu

Erik Burger
KASTEL – Institute of Information
Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
burger@kit.edu

Abstract

In view-based development, consistency plays a central role across concrete models. Consistency can be achieved automatically through consistency-preservation rules defined, for instance, in a change-driven language such as the Reactions language of the Vitruvius framework. This language, however, assumes the classical two-level architecture with a strict separation of type and instance level and can therefore only handle multi-level models through their generic linguistic structure. This forces domain questions to be encoded as not statically typed string operations on infrastructure metaclasses. Based on a previously published vision, this paper presents the Deep Reactions language. It lets a stable region of a deep model serve as an ontological metamodel, over which ontological types and features can be referenced directly and deep characterization through offspring-aware triggers can be exploited. Building on the stable code generation of the original Reactions language, a pre-processing step first transforms Deep Reactions into regular Reactions which ensures compatibility with existing implementations. Our evaluation shows that the number of referenced metamodels and the token count can be reduced while removing 83% of statically unvalidated type references.

CCS Concepts

- **Software and its engineering** → **Domain specific languages;**
- **Information systems** → *Mediators and data integration.*

Keywords

Multi-level modeling, V-SUM, View-based modeling, Vitruvius, Consistency, Deep instantiation

ACM Reference Format:

Annika Kienle, Arne Lange, Thomas Weber, Lars König, Nathan Hagel, and Erik Burger. 2026. Deep Reactions: Realizing and Evaluating Consistency Preservation for Multi-Level, Multi-View Models. 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/10.1145/3837062.3838703>

1 Introduction

With the growing size and complexity of software-intensive systems, model-driven software development manages complexity through abstraction. Two paradigms that have emerged in this area are multi-view modeling and multi-level modeling (MLM). Multi-view modeling provides several representations of a system, each tailored to a stakeholder concern, and a view-based framework, such as Vitruvius [12], keeps semantically overlapping views consistent as the system evolves. MLM instead bypasses the traditional two-level hierarchy of types and instances by admitting an arbitrary number of ontological levels orthogonal to the linguistic classification [2], which enables the precise modeling of domains whose concepts do not fit the type-instance dichotomy. A mature MLM workbench is Melanee [1], which implements strict, level-agnostic modeling.

The two paradigms have complementary strengths: what each one does well is precisely what the other lacks. Vitruvius preserves consistency between overlapping models automatically, but is bound to two-level modeling and therefore forces workarounds for domains that do not adhere to it naturally. Melanee models such domains without workarounds. Instead, it uses its strict discipline, which protects the modeler from anti-patterns and forbids relationships from crossing level boundaries. This means a domain concept that must relate to elements on several levels has to be duplicated. Keeping such duplicates consistent then meets the limit of MLM in Melanee: consistency is checked by constraints, but not automatically preserved after changes. Each paradigm thus offers exactly the capability the other is missing, yet the two have so far been investigated mostly in isolation. The few approaches that combine the two have limitations: Doreen [5] is not accessible



This work is licensed under a Creative Commons Attribution 4.0 International License.
MODELS Companion 2026, Málaga, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2903-4/26/10
<https://doi.org/10.1145/3837062.3838703>

and rests on a single redundancy-free model whose construction is not described, and OpenFlexo [10] realizes multi-level modeling through a generic type-object pattern in FML virtual models rather than by integrating a dedicated MLM workbench. Although model federation over autonomous, loosely coupled models is the defining capability of the framework, it is not what carries the multi-level hierarchy in their challenge solution, where the levels are built inside FML virtual models and federation only supplies editing tools for the resulting process language.

Weber et al. [19] took a first step by proposing to integrate Melanee’s deep models into Vitruvius. They sketched three requirements for a *deep* extension of the Reactions language [13], Vitruvius’s change-driven DSL in which each consistency preservation rule reacts to a specific kind of model change on a given metaclass and propagates that change to correlated models to restore consistency, namely that the language be aware of a deep model’s ontological structure (deepness), that changes be scoped by explicit level bounds, sketched as a `below level` clause on model references (level awareness), and that it offer reflective queries over the level structure, such as retrieving a level by its index (reflection). That work remained a vision: it neither implemented the proposed syntax nor evaluated its benefit, and it left open whether a stable metamodel can even be carved out of a deep model whose type and instance facets are, in general, intertwined. This paper closes that gap. We present a realized Deep Reactions language, a translation strategy that keeps the Vitruvius runtime untouched, and a Goal-Question-Metric (GQM) evaluation that quantifies the benefit against a shallow baseline. The contribution is guided by two research questions.

RQ1 What is required to integrate multi-level models into Vitruvius through a Deep Reactions language? Concretely, (i) can an ontology be separated into a stable part and a part subject to change? Vitruvius requires that, because it assumes its metamodels remain fixed while a system evolves. Yet in a deep model, type and instance roles are intertwined across levels. (ii) Which language features does this extension need?

RQ2 What benefits does the integration provide for the methodologist who writes consistency preservation rules over deep models?

2 Background

2.1 Multi-Level Modeling

Multi-level modeling overcomes the limitations of the two-level approach, revolving only around classes and their instances. Some domains require classes to characterize more than their immediate instances, a requirement called *deep characterization*, supported by *deep instantiation* [2]. A consequence of deep instantiation is that some model elements are types and instances simultaneously: they have an instance facet and a type facet, and are named class-objects, or *Clabjects*. The type facet is controlled by a *potency*, a non-negative integer (or a star) that bounds over how many levels a Clabject can have instances. Attributes carry an analogous notion: their *durability* bounds how many levels deep a feature remains present, and their *mutability* bounds on how many levels its value may be overwritten.

The particular flavor used here is the strict, characterization-potency-driven approach of the Level-Agnostic Modeling Language (LML) implemented in Melanee [1], whose linguistic metamodel is the Pan-Level Model (PLM). Strict modeling admits only the instance-of relationship across level boundaries and applies monotonic abstraction, so an instance of a Clabject resides exactly one level below it [3]. The linguistic and the ontological classification form the two dimensions of the Orthogonal Classification Architecture: every Clabject is linguistically an instance of a PLM metaclass such as *Entity*, and ontologically an instance of a domain concept, such as the S400 phone model from the Multi Collaborative Comparison Challenge [17], though root-level Clabjects and linguistic extensions carry no ontological type.

Instantiation chains and inheritance hierarchies together span a hierarchy of descendants below each Clabject. We call a Clabject an *offspring* of another if it is reachable by following classification and inheritance relationships transitively, in any combination and to any depth, with at least one classification step to cross a level boundary. Unlike a direct instance, which lies exactly one level below its type, an offspring reached through classification can reside an arbitrary number of levels below. This notion is central to the language presented in Section 4.

2.2 View-Based Modeling and the Reactions Language

During the development of complex systems, developers create different models that describe the same system and can therefore become inconsistent as the models of the system evolve differently. Freedom from inconsistency is necessary, though not sufficient, for correctness [9]. The projective approach avoids pairwise consistency relationships by projecting all views from a single underlying model (SUM) [16]. Because a redundancy-free SUM is hard to build for heterogeneous, pre-existing metamodels, Vitruvius realizes a *virtual* SUM [12] (VSUM): it tolerates redundancy across models and keeps the overlapping parts consistent through explicit consistency preservation rules (CPRs). Views projected from a consistent V-SUM are consistent by construction.

Vitruvius specifies CPRs in the imperative, unidirectional, change-driven Reactions language [13]. A reaction is triggered by a change of a certain kind to an object of a certain metaclass and calls routines in response. A routine has a *match* block that retrieves related elements, a *create* block that instantiates new ones, and an *update* block that performs arbitrary modifications on the target model, written in the Xtend dialect of Java [8]. Reactions are defined between metamodels and compiled to Java classes that are instantiated once when the V-SUM is initialized. There is no mechanism to load new or modified reactions into a running system. Vitruvius therefore relies on metamodels being stable for the lifetime of the V-SUM. For deep models, the only metamodel Vitruvius sees is the generic PLM. A methodologist thus references only linguistic types such as *Clabject*, *Level*, or *Attribute*, and must recover the domain meaning through string comparisons on type and attribute names. Weber et al. [19] illustrate this verbosity and propose a deep extension, which this paper realizes.

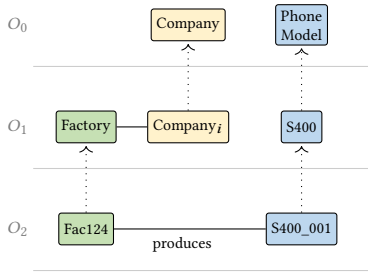


Figure 1: Schematic *FactoryAsProducer* model: a factory (green) produces devices, so Factory is introduced on O_1 . The Company hierarchy is yellow and the overlapping PhoneModel hierarchy blue. Open triangles denote instance-of.

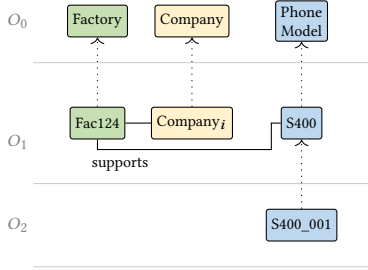


Figure 2: Schematic *FactoryAsSupporter* model: a factory supports phone models, so Factory is introduced one level higher, on O_0 . The blue PhoneModel hierarchy is shared with Figure 1 and must be kept consistent.

3 Motivating Example

The challenge domain involves three genuine ontological levels: PhoneModel (O_0) types specific phone models such as S400 (O_1), which in turn type concrete devices such as S400_001 (O_2). Because S400 plays both the role of an instance and a type within the domain, any solution must account for this multi-level structure, either by embracing it directly with MLM or by encoding it through workarounds in a two-level setting.

We reuse the mobile-phone domain from the Multi Collaborative Comparison Challenge [17] and its Melanee solution [15], also used by Weber et al. [19]. The domain describes companies, factories, produced devices, and the phone models they conform to. A factory both *supports* phone models at one level of abstraction and *produces* the devices that instantiate those models one level below. Because strict modeling forbids relationships across level boundaries, the monolithic Melanee solution has to duplicate the *Factory* and *Company* concepts into two Clabjects representing the same real-world object on different levels. This “multiple-representations trick” introduces accidental complexity solely to satisfy the strict multi-level modeling flavor.

The trick can be dissolved by separating the two concerns into two overlapping deep models kept consistent by Vitruvius. The *FactoryAsProducer* model (Figure 1) captures a factory producing devices; the *FactoryAsSupporter* model (Figure 2) captures a factory supporting phone models. The two models fully overlap on

the phone-model hierarchy (blue) as well as on the *Factory* and *Company* concepts, so a change on one side, for example, adding a new device instance of an S400, must be mirrored on the other.

Crucially, the shared *Factory* and *Company* concepts sit on *different* ontological levels in the two models: *Factory*, for instance, resides on O_0 in *FactoryAsSupporter*, a metamodel-leaning type for factory instances, but on O_1 in *FactoryAsProducer*, where it is itself a model-leaning instance of a higher concept. Specifying these CPRs in shallow Reactions is cumbersome because the only available Ecore metamodel is the generic PLM, which is not ontological, forcing domain questions through string comparisons on infrastructure types rather than direct type references. Deep Reactions address this by exposing ontological types directly, though the metamodel-leaning/model-leaning duality of elements such as *Factory* also poses a challenge for their locking mechanism, as discussed in Section 4.3.

4 From Requirements to a Realized Language

This section presents the realized Deep Reactions language. Weber et al. [19] formulated three requirements for a deep extension: R1 (deepness), that the language be aware of the ontological structure of deep models; R2 (level awareness), that it supports explicit level-based scoping of changes; and R3 (reflection), that it provide rich reflective query capabilities. We first assess the fate of these requirements under implementation, then introduce the language constructs that emerged.

4.1 Requirements Revisited

R1 asks the language to be aware of the deepness of models. It is the backbone of the approach and is realized by the ontological perspective described below. R2 asks for level awareness. Its *intent* survives, but its *mechanism* changes: in the previous paper [19] the below level clause was attached to source and target model references, whereas associating locking with the model import statement leads naturally to `until level`, which specifies the stable region directly at the point of import (Section 4.3). R3 asks for reflective capabilities to write rich queries. We retain the *aspiration*, served ontologically by offspring-aware triggers, but reject the *mechanism*: exposing linguistic level structure works against the perspective shift, and R3’s stated motivation, placing new elements at the correct level, is unnecessary under strict modeling because the level of a new instance is determined by the level of its type and is set automatically (Section 4.5).

4.2 The Ontological Perspective

The shared goal of the constructs below is to shift the methodologist’s perspective from the linguistic to the ontological dimension. Consider the device S400_001 (Figure 1). Linguistically, it is an Entity on a Level, linked to its type through a Classification, with its memory held in an Attribute named “RAM”. Verifying that it is an S400 means navigating to its classifier and comparing a string; reading its RAM means `getAttributeByName("RAM").value`. Both encode domain questions as unchecked string operations on infrastructure. Ontologically, the same element is simply an S400: the type check reduces to a static type check, and the RAM read becomes `element.RAM`, validated against the definition of S400 at

Table 1: Fate of the requirements from Weber et al. [19].

Requirement	Fate	Realization / Rationale
R1 (deepness)	survived, fulfilled	Ontological type references alias: :T, implicit classification on new T, offspring-aware triggers(offspring of)
R2 (levels)	survived, reframed	below level replaced by until level (locking) and at level ranges (trigger scope)
R3 (reflection)	abandoned	Linguistic level access contradicts the ontological perspective; its use case is automated instead

write time. This is exactly the experience an Ecore-based reaction already provides on a domain-specific metamodel. Deep Reactions extend it to deep models, where the linguistic and ontological dimensions are formally distinct.

A related design point is ruled out by the same goal: exposing the linguistic and ontological facets of an element side by side, as Doreen does in its SUM, offers uniform access but forces the methodologist to keep the two apart at every access. Deep Reactions take the opposite stance and present an ontologically typed element through its ontological facet alone, so a single reaction stays in one perspective. Where richer linguistic access is needed, the methodologist falls back to shallow Reactions.

4.3 Establishing an Ontological Metamodel by Locking

Vitruvius assumes metamodels are stable, since compiled reactions cannot be adapted at runtime. In a deep model, the distinction between model and metamodel is blurred: a Clabject's type facet acts as a metamodel to its instances while its instance facet has a type itself. To make ontological types available in reactions, a part of the model must nevertheless be fixed. Otherwise, runtime changes could invalidate reactions. We therefore introduce a locking mechanism that designates a stable region as the ontological metamodel. On a deep import, the optional until level clause names the deepest level that still belongs to the fixed region; if omitted, all levels present at build time are locked:

```
1 import deep "FactoryAsProducer.lml" until level 1 as producer
2 import deep "FactoryAsSupporter.lml" until level 1 as supporter
```

Note that a deep import references the model file itself rather than a separate metamodel, reflecting the fact that the locked region of the deep model serves as the ontological metamodel.

The granularity of this lock is a design choice. Deep models typically contain elements that lean toward one facet: top-level Clabjects resemble metaclasses, bottom-level Clabjects resemble objects, while intermediate levels are ambiguous. Level-based locking faces two distinct limitations. First, a single level may contain elements that should be locked alongside elements that should remain mutable. Finer-grained locking at the element level would address this. Second, some elements are inherently ambiguous: S400 acts as a template for concrete devices (a metamodel role), yet one might also add new phone models at runtime (a model role), a tension that no locking mechanism can resolve, since it requires the same element to be simultaneously stable and mutable. Level-based locking is a pragmatic first cut that already covers many cases, but it cannot always separate the conceptual metamodel from the model, because a single level cut does not need to coincide with the conceptual boundary. Finer-grained locking at the level of individual types is

conceivable and is examined in Section 7. The coarse level cut is the direct cause of the expressiveness limitation reported in Section 6. For types on levels that cannot be locked, the methodologist either writes shallow Reactions on PLM directly, or defines a reaction on a locked ancestor and relies on deep awareness to reach its offspring on unlocked levels (Section 4.7).

The Factory–Company ownership association from the challenge domain (Figures 1–2) is a concrete instance of this tension and was, for that reason, excluded from the evaluation in Section 6. On the producer side the association attaches to Factory itself, the same ambiguous element discussed above: locking it would expose the association statically, but the domain also needs to admit new factory types as siblings of Factory, not merely new instances beneath it, so Factory cannot be locked without foreclosing that need. On the supporter side the association attaches to individual factory instances such as Fac124, one level below the locked Factory root, which is structurally no different from S400_001 beneath PhoneModel and could in principle be reached deeply through an offspring-aware trigger. Because a single association must agree on both sides, the producer-side obstruction is decisive: at best, the association is realizable only as a hybrid CPR, for the same reason the produces/supports CPR (Table 5, #9) is already hybrid in Section 6, rather than as a new failure mode of its own. Finer per-Clabject locking does not change this. The obstruction on the producer side is not a matter of separating a type facet from an instance facet on one Clabject, since Factory would need to admit an entirely new sibling type rather than merely new instances of itself, and no locking granularity separates a Clabject from its own siblings. This delineates the class of associations for which locking is faithful: those whose openness enters only as offspring beneath an already-fixed root, as with PhoneModel, and not as new siblings of an already-referenced type, as with Factory.

4.4 Ontological Types and Deep Triggers

For deep imports, the ontological types from the locked region become available with the ordinary type syntax, e.g., producer : S400, and their ontological features are accessible directly, e.g., the RAM attribute in the original challenge. Inheritance and classification are unified under a single subtype relation, so the offspring of modifier can traverse both in any combination. The implementation of this unification is described in Section 5.

Listing 1: Grammar rules for deep triggers (EBNF).

```
1 deep-change = 'deep'
2 ( Clabject-change | attribute-change | conn-end-change )
3 [ 'at' 'level' level-range ] [ 'with' precondition ]
4 Clabject-change = [ 'direct' ] ( 'offspring' 'of' | 'element' )
5 Clabject-type ( 'created' | 'deleted' )
6 attribute-change = 'attribute' 'replaced' 'at'
7 [ 'direct' ] ( 'offspring' 'of' | 'element' )
```

Table 2: The four deep trigger categories and their shallow counterparts (preconditions omitted).

Deep trigger	Shallow counterpart
after deep element T created	Classification inserted into Level[content]
after deep element T deleted	Classification removed from Level[content]
after deep attribute replaced at T[a]	Attribute[value] replaced
after deep element c[e] replaced	ConnectionEnd[destination] replaced

Table 3: Type-matching modes of deep triggers and deep match statements.

Modifier	Match condition
(default)	Element is an instance of the type or one of its subtypes
direct	Element is a direct instance of the type (exact type match)
offspring of	Ancestor reachable upward through classification and inheritance
direct offspring of	As above, along classification only

```

8  Clabject-type '[' attribute-name ']'
9  conn-end-change = 'element'
10 clabject-type '[' connection-end-name ']' 'replaced'
11 level-range = int [ '-' ( int | '*' ) ]

```

A deep trigger is introduced with `after deep` and comes in four categories: Clabject creation, Clabject deletion, ontological attribute change, and connection-end change (Table 2). Each category admits four type-matching modes that control which elements in the instantiation hierarchy match (Table 3). By default, a trigger matches direct instances of the type and of its subtypes; `direct` restricts to exact instances; `offspring of` matches the transitive closure of instantiation and inheritance, with at least one instantiation step; and `direct offspring of` follows classification only. Because classification is not transitive, offspring at deeper levels are not instances of the type but are still its offspring, which is what lets a single reaction reach arbitrarily deep. The optional `at level` clause bounds the matched levels, as a single number (`at level 2`), a bounded range (`at level 2-4`), or an open-ended range (`at level 2-*`). It is only available together with `offspring of`, since otherwise the level is already fixed by the type’s position. Listing 1 gives the resulting grammar.

Each deep trigger corresponds to a shallow trigger on a linguistic PLM element together with a generated precondition that recovers the ontological intent (right column of Table 2). Creation and deletion do not react to the Clabject itself but to the insertion or removal of its `Classification` into a level’s content list. In PLM, a Clabject is first created as a raw linguistic entity and only acquires its ontological type once the `Classification` link connecting it to its type is established. Triggering on the Clabject element itself would therefore fire before the ontological type is known, making

it impossible to check whether the element is, for instance, an `S400`. By triggering on the `Classification` insertion instead, the type is already available and can be checked in the generated precondition. Reclassifying a Clabject therefore appears as a deletion followed by a creation, which re-establishes the element in the target model, so retyping produces the expected effect without a dedicated construct. Note, however, that the change history in the target model records a deletion followed by a creation rather than a reclassification, so the intent of the retype operation is not preserved in the history, even though the resulting model state is correct.

4.5 Deep Match and Deep Create

Match blocks retrieve corresponding ontological elements with `retrieve` and require absence of statements, both carrying the same modifiers as triggers with identical semantics (Table 3). The `at level` clause is omitted for matches, since in the common case a `retrieve` yields at most one element on a single, well-determined level. An optional `at level` would have been feasible but was not included. Create blocks instantiate ontological types directly with `new supporter::S400`. Because the level of a new instance is determined by the level of its type under strict modeling, Deep Reactions place the new clabject on the level one below its type and create the linking `Classification` implicitly, so the methodologist never touches PLM infrastructure. When the target type is not statically known but only bounded by an ancestor, the statement `new offspring of supporter::T typed supType` allows the concrete type to be supplied at runtime from the match block. Listing 2 shows a complete deep reaction that mirrors a newly created `S400` device, Listing 3 shows the shallow realization of the *same* CPR for direct contrast, and Listing 4 demonstrates the attribute-change trigger on a related CPR (factory name propagation).

Listing 2: A deep reaction propagating a new S400 device.

```

1 reaction NewS400Inserted {
2   after deep direct element producer::S400 created
3   call insertNewS400(affectedEObject)
4 }
5 routine insertNewS400(producer::S400 prodS400) {
6   match {
7     require absence of supporter::S400 corresponding to prodS400
8   }
9   create { val supS400 = new supporter::S400 }
10  update {
11    supS400.name = prodS400.name
12    addCorrespondenceBetween(prodS400, supS400)
13  }
14 }

```

Listing 3: Shallow realization of the *same* creation CPR as Listing 2 (S400), operating on the linguistic PLM layer.

```

1 reaction NewS400Inserted {
2   after element producer::Classification inserted in producer::Level[content]
3   with affectedEObject.^level == 2 && newValue.type.name == "S400"
4   call { if (isProducerLevel(affectedEObject)) insertNewS400(newValue,
5     instance as Entity) }
6 }
7 routine insertNewS400(producer::Entity prodS400) {
8   match {
9     require absence of supporter::Entity corresponding to prodS400
10    with isSupporter(it) && it.directType.name == "S400"
11    val supType = retrieve supporter::Entity corresponding to prodS400,
12      directType
13  }
14  create { val supS400 = new supporter::Entity }

```

```

13 update {
14   val lvl = supporterDeepModel.getLevelAtIndex(2)
15   PLMPackage.eINSTANCE.PLMFactory.configureClObject(suppType, supS400, lvl,
16     true)
17   supS400.name = prodS400.name
18   addCorrespondenceBetween(prodS400, supS400)
19 }

```

Listing 4 shows an attribute-change reaction. It fires when a factory name is updated in one model and propagates the new value to the corresponding factory in the other. Compared to Listing 2, the trigger uses `attribute replaced at` instead of `element created`, and the update block reads the built-in `newValue` variable.

Listing 4: Deep attribute-change reaction propagating a factory name update.

```

1 reaction FactoryNameChanged {
2   after deep attribute replaced at
3     direct element producer::Factory[name]
4     call syncFactoryName(affectedEObject)
5 }
6 routine syncFactoryName(producer::Factory prodFac) {
7   match {
8     val supFac = retrieve supporter::Factory
9       corresponding to prodFac
10  }
11  update { supFac.name = newValue }
12 }

```

4.6 Soundness Conditions

The soundness conditions below derive from the implementation choice, detailed in Section 5, of representing ontological types as synthetic interfaces and mapping classification onto inheritance. This mapping makes every feature declared on an ancestor available on all of its offspring, and is sound only when the deep model satisfies a condition on durability and mutability: for every ontological feature, both must be at least the potency of the containing clobject. The potency bounds the deepest level at which a classification descendant can exist, whereas durability and mutability bound how deep the feature stays present and writable. If either bound is smaller than the potency, the synthetic type promises access to a feature that the actual clobject has locally dropped or frozen, so a statically valid access can fail at runtime. The condition is not enforced. It is best read as a precondition for guaranteed soundness, with the methodologist restricting a reaction to the safe level range (for instance via `at level`) on deep models that violate it. The underlying issue is that classification does not satisfy the Liskov substitution principle in the way the inheritance mapping assumes.

4.7 Pure Routines and Semi-Dynamic Reactions

The shallow Reactions produced by translation (Section 5) express both deep models through the same PLM metaclasses `Entity`, `Level`, `Classification`, and so on. When Vitruvius receives a change event, it sees only these linguistic types: a newly classified `Entity` could belong to either `FactoryAsProducer` or `FactoryAsSupporter`, and without further inspection, a reaction meant for one model would fire erroneously for the other. We introduce *pure routines*, routines restricted to a match block and therefore free of side effects. In this context, they navigate the containment hierarchy to determine which deep model an element belongs to and return a boolean (e.g. `isProducer`, `isSupporter`). Pure routines are not

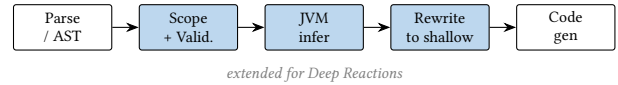


Figure 3: The Reactions compilation pipeline. Deep Reactions extend scoping, validation, and JVM model inference, and add a rewrite stage that lowers deep constructs to shallow ones before the unchanged code generator and runtime take over.

specific to Deep Reactions. The transformer inserts the disambiguation checks implicitly for Deep Reactions, while shallow Reactions must call them explicitly, as seen in Listing 3.

Locking freezes the referenced types, but the offspring of modifier extends a reaction’s reach to instances/offsprings created after the reaction was written. We call this *semi-dynamic*: a reaction on the locked ancestor `Mobile_Phone_Model`, triggered by offspring of ... created at level 1–2, reacts both to new phone models on level 1 and to their device instances on level 2. When a developer later adds an unforeseen model such as `S500` and its devices, the same reaction propagates them without any change to their source, provided the new offspring introduce no features beyond those already defined on the locked ancestor. This directly resolves a pain point of the previous prototype [19], which required restarting Vitruvius to react to a newly introduced phone model.

5 Realization by Translation

Deep Reactions are not executed by a new runtime. Instead, the extended grammar, scoping, validation, and JVM model inference produce Deep Reactions that are *translated to conventional shallow Reactions before code generation*, as sketched in Figure 3. The Reactions language is an Xtext-based DSL. Its JVM model inference, implemented using the Xbase API, defines the developer-specified mapping from DSL constructs to Java classes and serves two purposes. At editing time, the inferred model drives content assist and type checking against ontological types. At build time, it is serialized into the Java classes that implement the reactions. This creates a tension, since synthetic ontological types must be present during editing but must not reach the generated code, which only understands linguistic PLM types.

The implementation resolves the tension by rewriting deep constructs into their shallow equivalents before code generation, so the existing code generator and the Vitruvius runtime are reused essentially unchanged. The methodologist’s Xbase blocks become shallow once ontological feature accesses are rewritten to PLM navigation. The inferred JVM types become shallow as well, by re-inferring them from the rewritten AST rather than by editing the JVM model directly. A dedicated deep code generator was rejected because there is no one-to-one correspondence between deep constructs and code artifacts. A single deep entity creation, for instance, expands into an entity, a classification, and their configuration. The rewrite, therefore, expresses these expansions once, on the named constructs of the Reactions model. Among its stages are trigger transformation, Clobject disambiguation, stable-level guards, and create-statement transformation. Trigger transformation maps each

Table 4: GQM mapping: metrics assigned to questions.

	Metric	Q1	Q2	Q3	Q4
M1	Metamodel references	•			
M2a	Logical lines of code	•	•		
M2b	Number of tokens	•	•		
M3	Linguistic PLM references		•		
M4	Additional vocabulary		•		
M5	Unvalidated type refs			•	
M6	Fully-deep CPRs				•

deep trigger to the shallow counterpart of Table 2 with a generated precondition. Clabject disambiguation inserts pure-routine guards to ensure each reaction fires only for elements from the intended model. Stable-level guards reject runtime changes to elements on locked levels, enforcing that the locked region remains immutable for the lifetime of the V-SUM. Create-statement transformation materializes the implicit classification when a new Clabject is instantiated.

6 Evaluation

This section addresses RQ2 by quantifying the benefit Deep Reactions provide to methodologists writing consistency preservation rules over deep models. We apply the Goal-Question-Metric (GQM) approach to derive objective metrics from a qualitative goal, collect them on two realizations of the same scenario, and discuss the results together with threats to validity.

6.1 GQM Plan

The expected benefit of Deep Reactions, namely lower complexity, better understandability, and reduced error potential, is difficult to assess directly: complexity and understandability are inherently subjective, and error potential can only be approximated by objective proxies without a controlled user study. We evaluate it with the Goal-Question-Metric approach [6, 7], which derives objective metrics from a qualitative goal. The goal, derived from RQ2, is to *improve the usability of consistency preservation rules for methodologists working with multi-level, multi-view models*. It decomposes into four questions: whether Deep Reactions lower structural complexity (Q1), improve understandability (Q2), reduce error potential (Q3), and cover the expressiveness needed for ontological consistency (Q4). The associated metrics are: referenced metamodel elements (M1); logical lines of code (M2a) and number of tokens (M2b); linguistic PLM references (M3); additional vocabulary a methodologist must learn (M4); statically unvalidated type references (M5); and the proportion of fully-deep CPRs (M6). A single metric may serve several questions. Table 4 shows the full mapping.

This mapping already carries a construct-validity assumption worth stating up front: M1–M2b stand in for structural complexity and M3–M4 for understandability, but understandability itself is approximated here by size and reference-count metrics rather than measured through a comprehension task. We treat this as an assumption of the plan rather than an established fact, and revisit it as a threat to validity in Section 6.4.

Table 5: The nine ontological consistency rules of the scenario.

#	Consistency rule (abbreviated)
1–2	Each Factory / Model instance has a corresponding instance on the other side
3–5	Corresponding Factory/Model instances agree on name and RAM
6–8	Each Model offspring on O_2 has a corresponding offspring; they agree on name and IMEI
9	Each produces connection has a corresponding supports connection

6.2 Scenario and Realization

The metrics are collected on two realizations of the *same* set of CPRs for the mobile-phone scenario. Nine ontological consistency rules between *FactoryAsProducer* and *FactoryAsSupporter* (Table 5) decompose into fourteen CPRs, one per change kind that can violate a rule. Existence invariants split into an insertion and a deletion CPR, attribute invariants into a single attribute-change CPR, and the produces/supports invariant into an insertion CPR and one connection-end-change CPR per endpoint.

The shallow realization uses only the existing Reactions DSL and pure routines on PLM. The deep realization uses Deep Reactions wherever the relevant types can be locked, and falls back to *hybrid* reactions elsewhere, using deep references on the side whose types are locked and shallow PLM access on the other. Reactions that only mirror PLM infrastructure (Domain, DeepModel, Level) are excluded because they express no ontological consistency rules and thus fall outside the scope of Deep Reactions by design. It is incidental that they are also identical in both versions. To keep the comparison fair, the shallow version uses the same pure routines and the same PLM convenience helpers (`configureClabject`, `deleteClabject`) available to any methodologist, making the shallow baseline more concise and representative than the hand-inlined code of the previous examples, which did not use these helpers. A shared test suite verifies that the two versions produce identical and correct observable behavior.

For the size metrics, one statement of the Reactions DSL counts as one logical line of code (M2a): a trigger with its with clause, a retrieve with its corresponding to and with clauses, and the source/target metamodel declaration each count once, even when split across physical lines. M2b counts lexical tokens via the Reactions lexer and is the only automatically collected metric. The remaining metrics are counted by manual inspection.

6.3 Results

Table 6 summarizes the measurements.

Q1 (structural complexity): positive. Referenced metamodel elements (M1) nearly halve, from 214 to 108, as long PLM navigation chains give way to direct ontological references. Logical lines (M2a) drop modestly (−12%). The one place the deep version needs *more* code is that a fully deep creation trigger must name the level or endpoint statically, so a shallow reaction that serves several levels with different handling must split into several deep ones.

Table 6: Metric values for the shallow and deep realizations of the fourteen CPRs. Lower is better for M1–M5. M6 is omitted as it applies only to the deep version and is discussed separately under Q4.

	Metric	Shallow	Deep	Change
M1	Metamodel references	214	108	–49%
M2a	Logical lines of code	110	97	–12%
M2b	Number of tokens	1603	1005	–37%
M3	Linguistic PLM references	169	55	–67%
M4	Additional vocabulary	22	23	+5%
M5	Unvalidated type references	42	7	–83%

Table 7: Additional vocabulary (M4) by category.

Category	Shallow	Deep	Diff.
PLM EClasses	8	7	–1
PLM attributes and operations	14	9	–5
Deep Reactions keywords	0	7	+7
Total	22	23	+1

Token count (M2b) drops more visibly (–37%): the surviving statements are shorter, property access replaces method-call chains, and many trigger preconditions vanish because the ontological trigger type encodes them. The per-statement complexity NOT/LLOC (derived from M2b/M2a) falls from 14.6 to 10.4, confirming that per-statement compaction is the larger effect.

Q2 (understandability): partially positive. Linguistic PLM references (M3) fall by two thirds, from 169 to 55. The residue comes from the hybrid reactions and from the `at` level/until level clauses, which are a standardized reference to the PLM Level concept, the one linguistic concept a methodologist must still know. The additional vocabulary (M4), however, edges *up* from 22 to 23. Table 7 breaks this down: the deep version removes five PLM attributes and operations but adds seven new Deep Reactions keywords, which are net additions on top of the PLM vocabulary rather than replacements, because as long as some reactions remain hybrid, the methodologist must still know the full PLM infrastructure. Understandability, therefore, improves for the individual reaction but not for the vocabulary that the methodologist must master overall.

Q3 (error potential): positive. Statically unvalidated type references (M5) drop from 42 to 7. All seven remaining references live in the six hybrid CPRs, where a shallow trigger still forces a string-literal type filter. In the eight fully-deep CPRs, a typo in a type name is rejected by the compiler rather than silently disabling a reaction at runtime.

Q4 (expressiveness): negative for the current prototype. Of the fourteen CPRs, only eight are fully deep; six remain hybrid. The shared cause is the locking granularity: producer-side `Huawei_MP_Factory` and the produces connection reside on level O_1 , which the scenario must keep open for runtime additions, so a trigger on that level cannot be deep. The supporter side references only types on the locked level O_0 and is fully deep. The remaining six stay hybrid,

not because the language cannot express them but because the two stable types happen to share a level with mutable instances.

Goal assessment. Two of four questions are clearly positive (Q1, Q3), one is partially positive (Q2), and one is negative (Q4), not because the language constructs lack expressiveness but because the current level-based locking granularity is too coarse to separate stable from mutable types. A finer-grained locking mechanism, one that locks individual Clabjects rather than whole levels, would lift the six hybrid CPRs into the fully-deep category and push every metric further in the intended direction: M5 would reach zero, M6 would reach 100%, and the residual PLM vocabulary behind M4 would disappear. The measured benefits are therefore a lower bound on what the integration can deliver.

6.4 Threats to Validity

Conclusion validity. Six of the seven metrics are counted by hand; only M2b is computed automatically. Effect sizes are large (49%, 37%, 83%) so plausible manual counting errors on the order of a few units do not alter the qualitative direction. We cross-checked a random sample of five CPRs and found all hand counts consistent across two independent passes.

Construct validity. Usability is a subjective construct; the metrics are objective proxies whose link to a methodologist’s cognitive experience is not directly established. Labeling M1–M2b as *structural complexity* and M3–M4 as *understandability* does not prove these metrics track those constructs. A controlled experiment with comprehension tasks and error counts would be needed to establish the link more rigorously.

Internal validity. The author of both realizations is also the language designer, which risks bias toward the deep version. We mitigate this by equipping the shallow baseline with the same pure routines and PLM helpers available to any methodologist, making the baseline as concise as possible rather than artificially verbose. An independent re-implementation of the shallow version would further reduce this threat.

External validity. A single scenario with fourteen CPRs was used. It exercises all four trigger categories, creation, deletion, attribute change, and connection-end change, and covers both fully-deep and hybrid cases, but generalizability to other domains, other rule shapes, and larger CPR sets remains open.

7 Discussion

Three limitations shape what methodologists can express and both point to concrete future work.

Locking granularity. Level-based locking is, in general, too coarse to follow the conceptual contour of an ontological metamodel. In the mobile-phone example, the Phone and Factory hierarchies have different depths, so the concepts that conceptually belong to the metamodel sit at different levels in each model, and no single level cut covers both intended sets. Concepts below the cut are then inaccessible as ontological types even though they never change at runtime, while concepts above it cannot evolve even when the domain would allow it. This is exactly what forced six CPRs into the hybrid category in Section 6. Two finer-grained mechanisms would close the gap. The first lets the methodologist lock individual

types alongside whole levels, at the cost of extra specification effort. The second locks types automatically: every Clabject referenced as an ontological type, together with its ancestors, joins the locked region, so no explicit region has to be specified at all, though the locked set is then scattered across the reactions.

Soundness conditions. As Section 4.6 explains, mapping classification onto inheritance in the synthetic types was chosen to reach offspring of through the existing Xbase machinery, but it makes static type safety conditional on the durability and mutability conditions rather than absolute. A sound replacement is an open design question. One direction is to generate one synthetic interface per Clabject and level, restricted to the features still available at that level, so that a level-pinned trigger is typed against the corresponding interface. Level ranges, unbounded offspring matching, and infinite potency would each need their own treatment.

Toward fully dynamic Reactions. Semi-dynamic reactions already react to instances of types unknown at write time, but they are still written against the features of the locked ancestor. Fully dynamic reactions would let the methodologist add or change reactions and address features that only appear after deployment, while the V-SUM runs. The hurdle is that reactions compile to Java, so this requires runtime compilation and class loading. The benefit is most pronounced for deep models, whose appeal lies precisely in not fixing the type-instance split that static reactions impose. Independently, two constructs introduced here, the `direct` modifier and the typed create clause, would also benefit shallow Reactions, where subtype-precise triggering is currently awkward and runtime-typed creation is not currently supported.

Incremental translation. The translation from Deep to shallow Reactions (Section 5) currently runs once, over the whole reactions file, at build time: growing the locked region, for instance, to lift a hybrid CPR into the fully-deep category requires re-translating and redeploying every reaction, not just the ones affected by the change. An incremental version of the translation step, one that recomputes only the shallow code reachable from a changed deep construct, would let the locked region grow over the lifetime of a project without full regeneration, and is a natural companion to fully dynamic Reactions rather than a substitute for them: the former targets the build-time translation, the latter the runtime loading of reactions.

8 Related Work

Kühne [14] combats the accidental complexity of strict, level-adjutant MLM by separating orthogonal ontological dimensions, allowing Clabjects from different dimensions to connect regardless of level. Deep Reactions instead keep the strict discipline within each model and dissolve the same accidental complexity by splitting concerns across two consistent models. Orthographic Software

Modeling [4] is a deep, view-based approach built on a redundancy-free SUM. Because it too rests on strict MLM, its solution to the challenge would still require the multiple-representations trick (Section 3) inside the SUM. The V-SUM approach used here differs by tolerating redundancy and integrating multiple pre-existing meta-models via explicit consistency specifications [16]. The Doreen approach [5] exposes linguistic and ontological facets side by side

in a single SUM element, whereas Deep Reactions take the opposite stance and present an element through its ontological facet alone, so a fully-deep reaction stays in one perspective; hybrid reactions still mix the two where locking constraints require it.

Stevens [18] frames multi-model consistency as a build problem over megamodels, determining which models a change affects and performing the minimum necessary transformations. This is complementary to Deep Reactions at two levels. First, Stevens' approach addresses the scheduling problem, which models in a network are stale after a given change, while Deep Reactions addresses the execution problem—how to propagate a specific change between two overlapping deep models once the need for propagation has been established. Second, in a multi-level setting, a change at one ontological level may affect consistency at other levels in ways that depend on the model network topology; Stevens' megamodel perspective could inform which deep models need to be reconciled, while Deep Reactions specify the actual transformation rules.

The other solutions to the Multi Collaborative Comparison Challenge [17] offer further contrast. The Melanee/DLM solution [15] keeps everything in a single deep model but splits concerns into parallel hierarchy branches, introducing the multiple-representations trick that Deep Reactions dissolve. OpenFlexo [10] achieves a superficially similar result through FML virtual models and federation, but its multi-level structure lives inside FML rather than in a dedicated MLM discipline, so the two approaches rest on different notions of what a level is. Taken together, the challenge solutions illustrate a spectrum from monolithic approaches that avoid redundancy by encoding all concerns in a single model, to V-SUM approaches [12, 16] that tolerate controlled redundancy and make consistency explicit through separate specifications—at the cost of writing CPRs for each area of overlap, which Deep Reactions make substantially cheaper.

9 Conclusion

We realized and evaluated the Deep Reactions vision sketched by Weber et al. [19]. Answering RQ1, the integration rests on two pillars: a locking mechanism that carves a stable ontological meta-model out of a deep model, and a feature set composed of, ontological type references, four offspring-aware trigger categories, implicit classification, pure routines, and semi-dynamic reactions (offspring-aware). Our extension lets methodologists write reactions on deep models much as on Ecore metamodels. Separation into stable and changeable parts is possible in the same qualified sense as freezing a metamodel in the classical two-level setting. Answering RQ2, the GQM evaluation shows substantial improvements in structural complexity and error potential, a partial gain in understandability, and an expressiveness gap caused by level-based locking rather than by the language. Translating Deep Reactions to shallow ones keeps the Vitruvius runtime unchanged and makes the extension deployable today.

The next step is finer-grained locking at the granularity of individual Clabjects, which the evaluation identifies as the single change that would lift every metric to its ideal. Beyond that, we plan a user study to connect the objective proxies to methodologists' experience, and further multi-level scenarios to strengthen external validity. A deep Vitruvius would let Melanee's deep models and the

pragmatic V-SUM approach reinforce each other. Methodologists gain multi-level flexibility for existing V-SUMs, and existing deep models with overlapping information gain automatic consistency preservation.

More broadly, this work demonstrates that the type-instance boundary, long treated as a fixed axiom in view-based frameworks, is in fact a configuration parameter. By choosing how much of a deep model to lock, a methodologist decides where the ontological metamodel ends and where the evolving model begins. This flexibility opens the door to a class of scenarios currently out of reach for tools like Vitruvius, including multi-step product lines and living ontologies whose domain concepts evolve alongside the instances they type. The semi-dynamic reactions introduced here already address part of this space by letting a single reaction reach instances of types not yet known at the time of writing them. Fully dynamic reactions, in which new rules can be loaded into a running V-SUM, would cover the rest and are a natural next step for making the entire Deep Reactions approach as dynamic as MLM itself. The artifact, including both realizations and the shared test suite, is available at [11].

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – CRC 1608 – 501798263 and supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF), and by KASTEL Security Research Labs.

References

- [1] Colin Atkinson and Ralph Gerbig. 2012. Melanee: multi-level modeling and ontology engineering environment. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*. 1–2.
- [2] Colin Atkinson and Thomas Kühne. 2001. The Essence of Multilevel Metamodeling. In *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 19–33.
- [3] Colin Atkinson and Thomas Kühne. 2001. Processes and Products in a Multi-Level Metamodeling Architecture. *International Journal of Software Engineering and Knowledge Engineering* 11, 6 (2001), 761–783. doi:10.1142/S0218194001000724
- [4] 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 (Communications in Computer and Information Science, Vol. 69)*. Springer, Berlin, Heidelberg, 206–219. doi:10.1007/978-3-642-14819-4_15
- [5] Colin Atkinson, Christian Tunjic, and Torben Möller. 2015. Fundamental Realization Strategies for Multi-view Specification Environments. In *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. IEEE, 40–49. doi:10.1109/EDOC.2015.17
- [6] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. 1994. The Goal Question Metric approach. In *Encyclopedia of Software Engineering*. Wiley.
- [7] Victor R Basili and David M Weiss. 1984. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering* 6, 728–738.
- [8] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.
- [9] Istvan David, Hans Vangheluwe, and Eugene Syriani. 2023. Model consistency as a heuristic for eventual correctness. *Journal of Computer Languages* 76 (2023), 101223.
- [10] Sylvain Guerin, Joel Champeau, Jean-Christophe Bach, Antoine Beugnard, Fabien Dagnat, and Salvador Pérez. 2022. Multi-Level Modeling with Openflexo/FML: A Contribution to the Multi-Level Process Challenge. (2022).
- [11] Annika Kienle, Arne Lange, and Erik Burger. 2026. *Deep Reactions*. doi:10.5281/zenodo.21327188
- [12] Heiko Klare, Max E 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 (2021), 110815.
- [13] Max Emanuel Kramer. 2017. *Specification Languages for Preserving Consistency between Models of Different Languages*. Ph.D. Dissertation. Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. doi:10.5445/IR/1000069284
- [14] Thomas Kühne. 2022. Multi-dimensional multi-level modeling. *Software and Systems Modeling* 21 (2022), 543–559. doi:10.1007/s10270-021-00951-5
- [15] Thomas Kühne and Arne Lange. 2022. Melanee and DLM: a contribution to the multi collaborative comparison challenge. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '22)*. ACM, Montreal, Quebec, Canada, 434–443. doi:10.1145/3550356.3561571
- [16] Johannes Meier, Christopher Werner, Heiko Klare, Christian Tunjic, Uwe Aßmann, Colin Atkinson, Erik Burger, Ralf Reussner, and Andreas Winter. 2020. Classifying approaches for constructing single underlying models. In *Model-Driven Engineering and Software Development: 7th International Conference, MOD-ELSWARD 2019, Revised Selected Papers*. Springer, 350–375.
- [17] Gergely Mezei, Thomas Kühne, Victorio Carvalho, and Bernd Neumayr. 2021. The multi collaborative comparison challenge. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 495–496.
- [18] Perdita Stevens. 2020. Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels. *Software and Systems Modeling* 19, 4 (2020), 935–958. doi:10.1007/s10270-020-00788-4
- [19] Thomas Weber, Monalisha Ojha, Mohammad Sadeghi, Lars König, Martin Armbruster, Arne Lange, Erik Burger, and Colin Atkinson. 2024. Towards Deep Reactions in Multi-Level, Multi-View Modeling. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24)*. ACM, New York, NY, USA. doi:10.1145/3652620.3688208