# Towards Examining the Complexity of Consistency

Romain Pascual*†, Arne Lange†, Thomas Weber†, Lars König†, Michael Kirsten‡†, Terru Stübinger†
*MICS University Paris-Saclay, Paris, France
romain.pascual@centralesupelec.fr
†KASTEL Security Research Labs, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{firstname.lastname}@kit.edu
‡Ludwig-Maximilians-Universität (LMU) Munich, Munich, Germany
michael.kirsten@ifi.lmu.de

***Abstract*—Modeling and model-driven processes offer abstraction as a means to cope with the increasing complexity of systems. As systems become more complex, additional stakeholders with diverse expertise contribute, leading to heterogeneous and federated models, each capturing a different perspective and abstraction. Since these models describe overlapping aspects of the same system, some information is shared, and thus redundancy is introduced. Maintaining consistency of such information across models is crucial to ensure that they collectively provide a coherent system representation. In fact, inconsistencies can lead to errors in system development, making consistency necessary for system correctness. In addition, when the system is critical to safety, the correctness must be established with the highest level of guarantee, for example, achieved by formal verification. In this context, understanding which aspects of the consistency's complexity influence the complexity of verification may allow for more efficient verification techniques. In this paper, we examine the complexity of consistency for managing and mitigating verification efforts, to ultimately systematically reduce unnecessary complexity while ensuring the required consistency.**

***Index Terms*—Model Consistency, Complexity, Formal Proofs**

## I. Introduction

Modern software and systems engineering mitigates complexity by structuring development around pragmatic abstractions called models [1]. These models capture relevant information about the domain in which the system will be used. As the information becomes more detailed, the models evolve, that is, they are refined over time. System development is typically a collaborative effort that involves multiple stakeholders, teams, or organizations. Since models are related, overlapping, and aim to describe different views of the same system, it is essential to ensure that they remain *consistent* to maintain the integrity and reliability of engineering processes [2].

Each model captures a specific concern, which ensures separation of responsibilities during development. Hence, the models only collectively fully describe the intended system. As such, models typically specify properties that overlap in their meaning or interact in their behavior. Therefore, model *consistency* becomes a necessary condition for *joint realizability* [3]. Generally, any two models are consistent if they do not contradict each other, which would entail that their conjunction cannot be simultaneously realized.

Inconsistencies hinder realization and may lead to systems that cannot be implemented. In contrast, a consistent collection of models reduces ambiguity and prevents contradictions while establishing confidence that the design will result in a correct and (semantically) feasible system. Consistency checking is therefore both an integration prerequisite and a lightweight verification [2], similar to early formal validation [4].

However, consistency management of practical systems is challenging. Existing approaches range from lightweight checks to full-fledged formal verification, offering little insight into the cost of checking consistency. This cost depends on the *complexity of consistency*: What makes some consistency relations more difficult to verify than others? What features of the models or constraints drive that complexity? Such questions create the need to understand what this complexity is and what information it carries.

In this paper, we introduce an exploratory method for characterizing the complexity of consistency. We observe that many consistency constraints between modeling artifacts can be (abstractly) captured in fragments of the Object Constraint Language (OCL) [5]. Using Featherweight OCL [6], a shallow embedding of OCL into higher-order logic, we transcribe these constraints into the theorem prover Isabelle/HOL [7]. This encoding makes the verification process explicit and allows us to *measure* the effort of consistency checking by analyzing the generated proof obligations and the structure of their proofs.

Our abstract perspective contributes to the broader goals of verification and validation by offering a formal lens on the effort required to ensure model consistency. To illustrate this idea, we use a collaborative automotive design example and highlight several key dimensions of complexity inspired by structural software metrics [8]. While not exhaustive, the selected dimensions are intrinsic to multi-model development and can guide future efforts toward better tool support and methodology.

We seek to explore both, (1) how the complexity of consistency constraints can be assessed, and (2) which structural or logical properties of models and constraints affect this complexity the most.

This work was funded by the Topic Engineering Secure Systems of the Helmholtz Association (HGF), by KASTEL Security Research Labs, Karlsruhe, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263, and supported by the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF). This paper has been edited by our textician Daniel Shea. We used artificial intelligence (AI) only for checking the spelling and grammar.

Our contributions are as follows:

- We turn the notion of consistency complexity into a *measurable* property through a rigid formalization, namely the encoding into Isabelle/HOL via Featherweight OCL
- We identify and analyze key dimensions of consistency complexity using a working example, focusing on proof obligations and logical structure.
- We reflect on consistency management through formal verification and highlight limitations and the potential of proof complexity to guide modeling.

Our contribution is a proof of concept that lays the groundwork for a systematic approach to consistency analysis grounded in formal reasoning. It shows how early formalization (even on small examples) can yield actionable insight into the feasibility and cost of consistency checking. Such analyses can help guide modeling decisions by functional adequacy and the formal tractability of the constraints they induce.

## II. Foundations

### A. Model Consistency

Model consistency is a fundamental concern in model-driven and model-based engineering. A common view treats consistency as a relation: models are either consistent or not [9]. Consistency refers to the absence of contradictions between two or more models that are related, for example, through refinement, projection, or cross-domain mappings. Two models are said to be *consistent* if their combined statements do not contradict each other, and *inconsistent* otherwise. Specifying consistency relations in model-based engineering is done in various ways [3], [10]–[13], yet consistency checking is typically based on formal specifications, expressed, for example, in the Object Constraint Language (OCL) [5] or in model transformation languages that encode consistency preservation rules [14]. The choice of language influences the expressiveness and complexity of the consistency conditions. In this work, we use OCL for our examples as it offers a concise and standard way to express model constraints.

Although more refined notions of consistency, for instance describing gradual or temporal consistency, exist in the literature (see Sect. VI), we focus on a simple setting for clear detection and analysis of consistency violations.

### B. Complexity in Modeling

To analyze the complexity of consistency specifications, we need a measurable notion of complexity for models and their relationships. In software engineering, one widely used structural indicator is *size*. Albeit a coarse metric, size correlates with understandability and maintainability [15]. This principle carries over to modeling, where size (e.g., number of elements, expressions, or constraints) is often a proxy for complexity.

A key distinction is between *essential* and *accidental* complexity [16], [17]. Essential complexity stems from the inherent difficulty of the domain or specification task. Accidental complexity arises from limitations of modeling tools, languages, or processes. In our setting, consistency specifications may include both complexity that is justified by the semantics of the domains being connected and complexity that could be reduced through better abstractions or tooling.

While more sophisticated metrics exist (e.g., McCabe's cyclomatic complexity [18] and Halstead's effort metrics [19]), we focus on size-based measures as a practical and implementation-independent first approximation. This enables general reasoning about the difficulty of understanding and maintaining consistency specifications.

## III. Dimensions for the Complexity of Consistency

Developing systems requires understanding the domain in which the system will be used, often through the means of appropriate abstraction. Modeling allows for the construction of such abstractions, typically by refining models over time. When multiple organizations collaborate on a system, each defines and refines their own metamodels in order to describe their own excerpt from the domain of the developed system. These metamodels define structural elements — metamodel elements — representing the system's concepts. When some of these concepts overlap, the associated metamodel elements are duplicated across organizations.

In this section, we introduce an example of a car developed collaboratively by two organizations: Car manufacturer and Supplier. The two organizations model a message bus for communication between the car's components, but their representations may vary in the levels of detail. The message bus is an example of a metamodel element. The overlap between the two metamodels needs to be managed and kept consistent via explicit consistency specifications. Through our example here, we want to illustrate such model refinements and the resulting overlaps, with the stated aim of discussing and assessing the overlaps' associated complexity.

Before we proceed, we define two terms that are needed to fully describe consistency specifications: *correspondence* and *coextension*. Correspondence, on the one hand, is a relationship on metamodel elements, namely a 1-to-1 relationship. If an instance is created, then another instance of the corresponding model element must also be created. Coextension, on the other hand, builds on these correspondences between model elements and enables the retrieval of the corresponding model elements. We denote coextension by the $\sim$ symbol. Coextension works both on individual model elements and on their collections. The coextension operator is a function that uses the defined correspondences.

In the following subsections, we use the running example of a collaboratively developed car to explore the key dimensions of our intuitive notion of complexity in the context of consistency. It is not our aim to cover all possible dimensions of complexity of consistency, since many of these actually emerge from the complexity of the domain. Instead, we lay our focus on those dimensions which we regard as intrinsic to developing a system with multiple metamodels.

### A. Arity of the Consistency Specification

The Car manufacturer and the Supplier start with the most trivial model of a car, which consists only of the car itself,

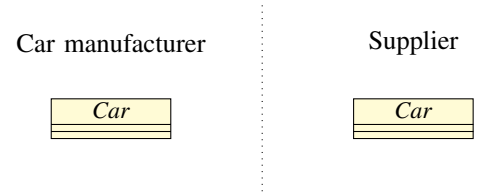


Fig. 1. The Car manufacturer and Supplier both want to build the same car, thus their modeling starts with the most basic abstraction, i.e., a *Car*.

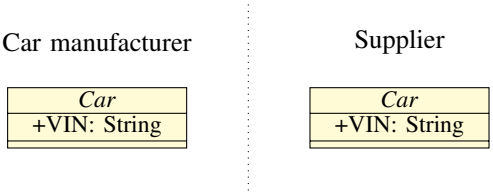


Fig. 2. *Car*s with structural features, like *VIN*.

not divided further, as illustrated in Figure 1. The notion of consistency introduced by this state of the example is the identity matching of metamodel elements in both models.

When structural features, such as attributes, are used to identify model elements, we can explicitly encode the coextension operation as the equality of these features. For Figure 2, we can use the attribute *VIN* to distinguish cars from each other. We can also use this attribute to determine the correspondence. If two model elements, instances of cars, from each side of the models, correspond with each other, they have the same value for the *VIN* attribute, hence we refine the equality of the coextension operator. This notion of identity matching can be extended to fields and meta-references, as illustrated in Figure 3 with the *messageBus* as String, and in Figure 4 with an explicit class *MessageBus*. Both representations of the message bus need to be consistent. For example, they have to share the protocol used, because both the components of Car manufacturer and Supplier use the same physical message bus and cannot communicate with each other if they do not use the same protocol.

While Car manufacturer handles the development of the whole car, Supplier only considers the parts of the car which it supplies to Car manufacturer. In our simplified example in Figure 4, Car manufacturer provides two components using the message bus, while Supplier provides only one component.

Car manufacturer might not even need to model the *ECUs*, if only Supplier has access to the *MessageBus* and if the relevant

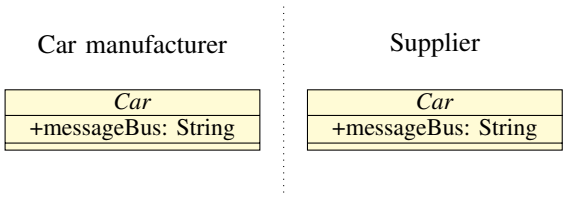


Fig. 3. The *Car* will have components communicating over a *messageBus*.

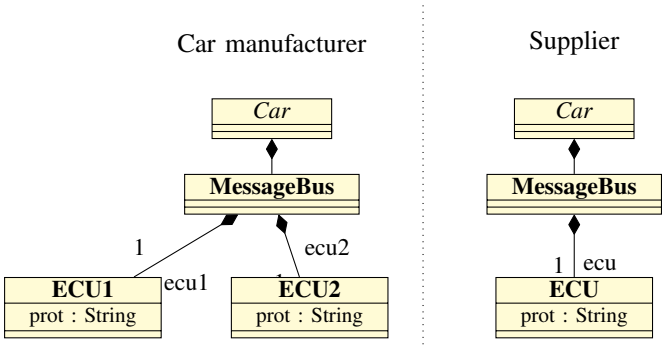


Fig. 4. Multiple types of Electronic Control Units (n-1 on meta).

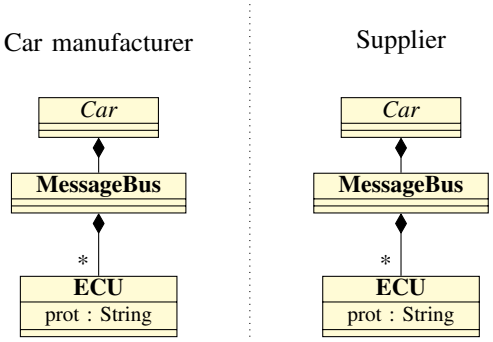


Fig. 5. Multiple Electronic Control Units (n-m on model).

aspects to be modeled are limited to the message bus and its communication protocol, for example to provide the correct voltage to the message bus. This scenario introduces our first dimension of the complexity of consistency: the arity of the mapping induced by a consistency specification. We already introduced a 1-to-1 mapping between the two *Car* classes, and a 1-to-$n$ mapping from *ECU* to the classes *ECU1* and *ECU2* in Figure 4. The two remaining members of this dimension are an $n$-to-1 and an $n$-to-$m$ mapping. The $n$-to-1 mapping is needed to preserve consistency for changes made by Supplier, as the reverse of changes made by Car manufacturer, which require a 1-to-$n$ mapping. The remaining $n$-to-$m$ mapping occurs, e.g., if the abstraction levels on both sides differ, as illustrated in

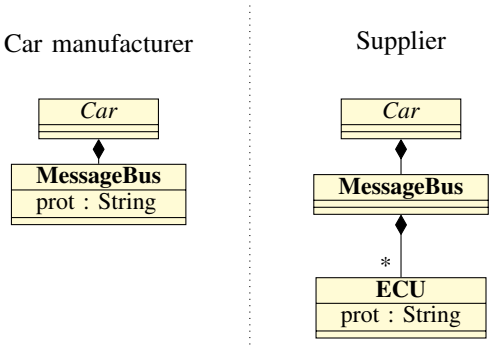


Fig. 6. The *MessageBus* is modeled explicitly, as it is too complex to be described by a simple type like String (1-n on model).

Figure 5. Therein, the protocols of all *ECU*s must be the same, so that the components can communicate with each other.

These mapping arities live at the metamodel level. Figure 6 introduces a 1-to-1 on the metamodel level, i.e., the mapping connects the *MessageBus* on the side of Car manufacturer with the metaclass of the *ECU* on the side of Supplier. However, this induces a 1-to-$n$ mapping at the model level, which does not change the complexity of the consistency specification itself, but rather the computation required to determine consistency on specific instances.

In summary, consistency specifications can involve different numbers of model elements, which leads to the following dimensions of arity complexity:

- 1-to-1 mapping
- $n$-to-1 mapping
- 1-to-$n$ mapping
- $n$-to-$m$ mapping

While the concrete effect on complexity is hard to assess, e.g., a 2-to-2 mapping usually brings less complexity than a 1000-to-1 mapping. The $n$-to-1 mapping is the inverse mapping of the 1-to-$n$ mapping. The underlying idea of that heuristic is that the more model elements are involved in the specification, the more complex it is.

### B. Extended OCL Consistency Specification

Having described consistency specifications as relations or mappings between metamodel elements, a more precise formulation is needed toward a formal description. We propose to use OCL extended with a coextension operator, written "$\sim$". This operator is not part of the OCL specification, but it enables consistency specifications via constraints. This operator acts as a function that returns a Boolean value capturing whether two elements are in correspondence with each other, that is, whether they need to be kept consistent.

The "select($\sim$)" operator hence describes the set of coextended model elements. The resulting set might be empty if no other model elements coextend with the root element. The resulting set might also contain one or more elements, depending on how many model elements coextend with the root element. Coextension only happens if the elements overlap in a shared semantic space and for the purpose of consistency between these coextending elements. This abstraction allows us to focus on dimensions independently from the concrete consistency rule and the involved elements. We discuss that influence later in Sect. III-C. In our example from Figure 1, we can use the OCL constraint in Listing 1 to specify a consistency relation between models.

```
context Car manufacturer::Car
inv: Supplier::Car.allInstances()
    -> select(c|c ~ self)
    -> size() = 1
```

Listing 1. 1-to-1 nominal consistency specification

The context is the *Car* class in the car manufacturer model. The first step in the consistency specification is to query for all instances of the *Car* class in the supplier's model. Then, the specification requires that the size of the corresponding elements in that collection is exactly one. This means that there must not be an instance of a *Car* in either model that corresponds with more or less than one other instance of the other model. If the *Car* classes in both models have an attribute, for instance, the VIN (Vehicle Identification Number), we can refine our consistency specification to the concrete equality expression in Listing 2.

```
context Car manufacturer::Car
inv: Supplier::Car.allInstances()
    -> select(c|c.VIN = self.VIN)
    -> size() = 1
```

Listing 2. 1-to-1 computed consistency specification

Looking at Figure 6, we immediately see that the consistency specification becomes more complex when more elements are involved. Indeed, the message bus protocol must be kept consistent with possibly more than one *ECU* instance. We can formulate another OCL expression to specify this consistency (Listing 3), with the precondition that all cars and message buses have a unique correspondence.

```
context Car manufacturer::MessageBus
inv: let mb:Supplier::MessageBus
    = self.select(~) in mb.ecu
    -> forAll(e|e.prot = self.prot)
```

Listing 3. 1-to-$n$ consistency specification

In some situations (Listing 4), we have to rely on nominal correspondence instead of relying on structural or behavioral observations. The following constraint states that each ECU on a message bus must have a corresponding ECU on the opposite side of the model. In this example, the ECU instances do not have an attribute *prot* to infer their consistency relation.

```
context Car manufacturer::MessageBus
inv: self.ecu
    -> forAll(e|self.select(~).ecu
    -> exists(f|f ~ e)
```

Listing 4. $n$-to-$m$ nominal consistency specification

It is also possible that we want to keep instances of multiple metaclasses consistent, as shown in Figure 4. An example of such a consistency specification is provided by the OCL constraint in Listing 5.

```
context Car manufacturer::MessageBus
inv: self.ecu1.prot = self.ecu2.prot and
     self.ecu1.prot = self.select(~).ecu.prot
```

Listing 5. Metaclass consistency specification for the example in Figure 4

### C. Complexity of the Computation

So far, we have introduced computationally rather simple mappings of String or object identity for our coextension operator that serve to concretize correspondences if needed. Going beyond that assumption, the consistency specification may further include some computations written in a Turing-complete language or be expressed with a logical formulae.

Then, we additionally gain a notion of complexity based on the computation that is needed to express the consistency specification. This computation includes the computation of the values that have to be consistent, e.g., in the case of *MessageBus*, the consistency specification may also include a simulation for assessing whether it can handle the components or might get overloaded. This computation is part of the consistency specification and does not influence the complexity of the coextension operator, but it is needed in addition to its complexity. Similarly, the computation of a value may include multiple other values, where the boundary to the arity of the mapping becomes a bit blurry. On the one hand, the arity of the mapping has an influence, but on the other hand, the function used to combine the values also has an influence on the complexity of the computation. This is illustrated in Listing 6 with the method "simulate", which takes a set of *ECU*s, both from the message bus itself and from corresponding message buses, as input and returns a boolean value that indicates whether the *ECU*s can use the message bus or overload it. This value is then compared against the boolean field *MessageBus::overloaded*, which indicates whether the message bus is overloaded. Depending on the use case, it might be acceptable to have an overloaded message bus, where it is up to the developer's decision on how to react in the case when an inconsistency between the field value and the simulation result occurs. The complexity of the computation is, in this example (see Listing 6), the computation of the simulation and the coextension. In general, the complexity of the computation is connected to the complexity of the constructs of the language, e.g., the complexity of OCL [20].

```
context Car manufacturer::MessageBus
inv: simulate(self.ecu, self.select(~).ecu)
    <> self.overloaded
```

Listing 6. Non-breakable consistency specification with external computation

Therefore, the overall complexity of the consistency specification also depends on the complexity of the computation as part of the consistency specification. Concerning exclusively the coextension operators, we can determine their computational complexity in the $\mathcal{O}(n)$ notation. In the first case of the 1-to-1 mapping, the complexity is $\mathcal{O}(1)$, as we only need to check for the existence of two elements, one in the Car manufacturer model and one in the Supplier model. While the nominal correspondence yields a constant algorithmic complexity, the structural 1-to-1 mapping yields a linear algorithmic complexity of $\mathcal{O}(n)$ where $n$ is the number of elements in the set of Supplier model instances. The second case of the 1-to-$n$ mapping is $\mathcal{O}(n)$, as we have to check for the existence of one element in the Car manufacturer model and $n$ elements in the Supplier model. The third case is the $n$-to-$m$ mapping with a complexity of $\mathcal{O}(n \cdot m)$, as we have to check for the existence of $n$ elements in the Car manufacturer model and $m$ elements in the Supplier model. In this case, the computational complexity is approximately $\mathcal{O}(n^2)$ if we assume that the number of elements is the same in both models.

### D. Compositional Complexity

The arity and computation complexity are properties of a given consistency specification. However, a consistency specification might also be decomposed into several simpler consistency specifications. In the example with the message bus scenario, the consistency specification stating that all connected components must use the same message bus protocol can be broken down into individual consistency specifications, each ensuring that a single component conforms to the protocol. From an arity perspective, this transformation replaces one 1-to-$n$ to $n$ 1-to-1 mappings. A similar principle applies to the complexity of computation. Independent parts of the computation might allow for breaking down the specification such that each one compares sub-aggregations.

Still, not all consistency specifications can be decomposed in this way. Some constraints are inherently non-decomposable because breaking them down would alter their semantics and lead to incorrect verification results. For instance, consider the consistency specification based on the simulation of all the *ECU*s to ensure that they do not overload the message bus when operating simultaneously. If this specification is split such that each *ECU* is simulated in isolation, it would fail to capture the cumulative effect of multiple ECUs using the message bus at the same time. This type of non-breakable consistency specification occurs when the specification depends on global properties that cannot be meaningfully divided into independent subproblems. In this example, combining the simulation results for each *ECU* separately does not yield the result of the simulation with all *ECU*s.

Lastly, breaking down a consistency specification does not necessarily reduce its overall complexity. Whereas usually fewer metamodel elements are involved, we might lose easily accessible information that is costly to recompute for the consistency specification. In summary, to enable the decomposition of a consistency specification, we need a decomposition operation that inherently includes its composed state, e.g., splitting an equation into equal sub-equations also results in the equality of the whole equation.

## IV. Influence of Complexity Dimensions on Complexity with Formal Proofs

We now want to illustrate how the consistency specifications from the previous section can be employed in formal proofs which — if the specification holds — ensure that the system can indeed be realized. In order to showcase the associated proof complexity, we outline proofs for illustrative examples, but do not limit our expressiveness. As discussed in Sect. I, we use Featherweight OCL with the interactive theorem prover Isabelle/HOL. This allows for rigorous and structured proofs in a strong logic with as few assumptions as possible. Nonetheless, the following observations are not tied to the specific logic but, instead, are of a general and structural nature based on the metamodel elements and consistency notions at hand. As such, our observations are also expected to hold similarly, e.g, for simpler embeddings in Isabelle/HOL [21] or in the Rocq prover [22].

Formally verifying that a consistency specification holds means demonstrating, via rigorous proof, that a set of models adheres to a formally stated consistency requirement. The burden of proof entails providing a complete mathematical argument that the specification is met by the model instances. Once the formal proof is derived, it comprises a chain of instructions in a dedicated proof language, such that a theorem prover can check whether the consecutive application terminates as a complete proof. Many formal proof frameworks bear strong similarities with source code written in a programming language with internal structure and logical dependencies [23]. Indeed, similar to object-oriented code being split into classes with methods and possible inheritance, formal proof frameworks are structured in theory modules with definitions, theorems, and possibly theory imports. While it is hard to make precise statements about the real complexity of a proof, applying metrics such as lines of codes (LoCs), depth of function calls, or cyclomatic complexity to proof frameworks already allows approximating their complexity.

We can leverage these proof-based metrics to assess and compare the complexity of different consistency specifications. More precisely, we describe how the complexity of the proof relates to the various dimensions of complexity presented in Sect. III. First, we discuss how to formalize consistency on models on the example of the proof framework Featherweight OCL within the theorem prover Isabelle/HOL. Second, we show how this transfers to the structure of the respective formal proofs. Third, we discuss the resulting proof complexity.

### A. Formalizing Consistency via Coextension on Models

To formally prove consistency between models, we first derive a formal consistency relation that spans across various models, but can also be rigorously evaluated between every element of each metamodel. In the simplest case, checking consistency between two models might reduce to comparing primitive type values, e.g., String or Integer, that correspond to a model element. More coarse-grained consistency notions may simply check that the numbers of elements for a specific type correspond. Moreover, when addressing abstract models, not every model element may be represented by a primitive value (e.g., when their structure is too complex or when they are only described by informal domain knowledge). In such cases, natural consistency evaluation might not apply. Hence, we assume that the consistency for each non-primitive model element is captured by an a priori mapping of correspondences that tell us, e.g., that two specific feature-less cars are indeed consistent or inconsistent.

In our formal proof framework, we use the coextension relation (denoted $\sim$) to build on these correspondences and relate elements from two different metamodels. Thus, the first and second arguments of $\sim$ belong to distinct metamodels. As with OCL, Featherweight OCL only talks about a single metamodel, so we must define two disjoint halves from a single metamodel that we can treat as two distinct metamodels. Practically, this can be achieved by having two versions of each model and hence omitting the usual single `OclAny` class, which normally serves as the top of the class hierarchy to combine the models within their metamodel. This model duplication and artificial separation enables two fully disjoint universes of model elements within the same metamodel. Elements from the first universe can only be mapped to elements from the second universe, either via their (primitive) values or by evaluating $\sim$ on their correspondence map. By extending all nonprimitive model elements by such correspondence maps, we may use the coextension relation as a basis for our general formal notion of consistency.

In Figure 1, $\sim$ uses only object identities and the correspondence map, since there are no primitive values. To analyze the invariant given in Listing 1, we resort to a correspondence map, e.g., as follows:

```
definition correspondences_car where
  "correspondences_car ≡ Set{
      Pair{x_car1_1, x_car2_1},
      Pair{x_car1_2, x_car2_2}}"

definition coext_car (infixl "∼_car" 100) where
  "a ∼_car b ≡ correspondences_car
                 ->includes_Set(Pair{a, b})"
```

Moreover, the invariant of Listing 1 in Featherweight OCL using an Isabelle/HOL definition reads as follows:

```
definition One_to_One_inv :: "Car2 ⇒ Boolean" where
  "One_to_One_inv (self) ≡ Car1
    .allInstances()->select_Set(c | c ∼ self)
          ->size_Set() ≜ 1"
```

On the surface, only little has changed, i.e., the type annotations are now explicit for operators on collections and OCL's equality relation is written $\triangleq$ to distinguish it from Isabelle/HOL's general equality relation. The Isabelle proof assistant parses and typechecks this definition and thereby guarantees that it is syntactically correct and that the query semantically agrees with the metamodels in Figure 3.

As in Listing 2, we can also use structural features such as the VINs of Figure 2 to concretize the coextension operator to a simple check for equality.

```
definition One_to_One_Refined_inv :: "Car2 ⇒
                                      Boolean"
where
  "One_to_One_Refined_inv (self) ≡
     Car1 .allInstances()
      ->select_Set(c | c .vin_Car1 ≜ self
.vin_Car2)
      ->size_Set() ≜ 1"
```

To express invariants for larger models, such as in Figure 6, we need to generalize the coextension relation further, in order to work with correspondences on more than just one component. In the example, we can introduce the two distinct concrete operators $\sim_{\text{Car}}$ and $\sim_{\text{MB}}$ which consider the correspondence of cars and message buses, respectively. Hence, the bare $\sim$ can be used as a polymorphic operator to abstract away from the individual components and thus be available for use in place of either of the concrete operators.

We can now add the invariant on message buses in Listing 5, while still incorporating the invariant from Listing 1 on cars, as follows:

```
definition One_to_N_inv :: "MessageBus1 ⇒ Boolean"
where
"One_to_N_inv (self) ≡
  let mb = MessageBus2 .allInstances()
             ->select_Set(m| self ~ m)
             ->asSequence_Set()->first_Seq()
  in (mb .ecu_MessageBus2 ->forAll_Set(e|
    e .prot_ECU2 ≜ self .prot_MessageBus1))"
```

In order to keep the OCL extensions to a minimum, where Listing 3 uses `.select(~)`, we do the same with standard OCL operators plus the $\sim$ coextension.

Overall, we see that only little change is necessary regarding the invariants, in order to handle the meaning of our coextension operator in Featherweight OCL.

### B. Reasoning about Consistency with Isabelle/HOL

Having defined the invariants, we can now consider what it takes to prove that they hold in a given model, that is, in a given instance of the metamodel of Figure 1.

Inside Featherweight OCL, such an instance is called a 'heap state,' which we write $\tau$; and the relation $\tau \models e$ expresses that under the heap state $\tau$, the OCL expression $e$ will evaluate to `True`. A heap state captures a specific instantiation of metamodel components and associations. Since Featherweight OCL is a shallow embedding of OCL into Isabelle/HOL, we can freely mingle OCL notation with Isabelle's own metalogic, without invalidating our semantics. Using the relation $\models$, an Isabelle proof can directly talk about OCL expressions and prove whether they are satisfied for a given metamodel instance. Hence, we must prove that invariants are satisfied by specific heap states. Specifically, every instance of a metamodel element must satisfy the appropriate invariant, i.e., we must prove that for an arbitrary instance of the metamodel component $a$ and an invariant $\alpha_{\text{inv}}$ on $a$, the statement $\tau \models \alpha_{\text{inv}}(a)$ holds.

### C. Complexity of Consistency in Formal Proofs

In the following, we give *proof outlines* for our consistency theorems. Even though these are not full proofs, they already convey a sense of the dimension of complexity carried over from an informal semantics as presented in Sect. III into a formal semantics that is required for formal verification. As a first step, we consider again the invariant in Listing 2:

```
lemma example_2:
  fixes a :: "Car2"
  shows "τ ⊨ One_to_One_Refined_inv a"
proof −
 obtain b :: "Car1" where "{b} =
   {x. τ ⊨ Car1 .allInstances()->includes_Set(x)
         ∧ τ ⊨ x ~_Car a}"
 hence "τ ⊨ b ~ a"
 hence "τ ⊨ b .vin_Car1 ≜ a .vin_Car2"
 hence "τ ⊨ Car1 .allInstances()
          ->select_Set(c| c .vin_Car1 ≜ a
.vin_Car2)
          ≐ Set{b}"
 moreover have "τ ⊨ Set{b}->size_Set() ≜ 1"
 ultimately have
  "τ ⊨ Car1 .allInstances()
       ->select_Set(c| c .vin_Car1 ≜ a .vin_Car2)
       ->size_Set() ≜ 1"
 thus ?thesis unfolding One_to_One_Refined_inv_def
by simp
qed
```

This proof is somewhat lengthy, but still straightforward to read, as it closely follows the structure of the 1-to-1 mapping. We first obtain a value $b$ corresponding to $a$ via the coextension operator on cars. Then, we prove that this $b$ is unique. Hence, we may deduce that the VINs for $a$ and $b$ are the same, and additionally that `Car.allInstances().select(c| c.VIN ~ a.VIN)` has indeed size one. Thus, we prove that the invariant is satisfied.

We can contrast the above proof with a similar proof for the considerably more complex invariant in Listing 5:

```
lemma example_3:
  fixes a :: "MessageBus1"
  shows "τ ⊨ One_to_N_inv a"
proof −
 obtain b :: MessageBus2
  where "τ ⊨ MessageBus2 .allInstances()
               ->includes_Set(b)"
  and   "τ ⊨ a ~_mb b"
 from ‹τ ⊨ a ~_mb b› have "τ ⊨ a ~ b"
  by (simp add: squiggle_mb)
 moreover {
  have "∀e. τ ⊨ b .ecu_MessageBus2->includes_Set(e)
    ⟶ τ ⊨ e .prot_ECU2 ≜ a .prot_MessageBus1"
  hence "τ ⊨ b .ecu_MessageBus2->forAll_Set(e|
           e .prot_ECU2 ≜ a .prot_MessageBus1)"
 }
  ultimately show ?thesisqed
```

Here, we see that the mapping's arity shapes the proof. This outline is structurally similar to the previous one, and the coextension operator allows to obtain a value $b$ that corresponds to $a$; yet, in the second half, we have an additional universal quantifier. In order to prove the validity of `ecu->forAll(e| e.prot = a.prot)`, we lift it to Isabelle's metalogic, and we obtain the additional proof obligation which quantifies over `ecu` components on the heap state: $\forall e.\ \tau \models$ `b.ecu->includes(e)` $\rightarrow \tau \models$ `e.prot = a.prot`.

Likewise, we have seen the influence of compositional complexity on the proof. As in Sect. III-D, the 1-to-$n$ mapping can be replaced by $n$ 1-to-1 mappings, which yields simpler (but more) proof obligations. However, such a restructuring has little influence on the overall proof, merely making complexity more visible. Therefore, by replacing the universal quantifier in the second half of the proof, we gain another proof obligation over the additional 1-to-1 mapping. Hence, we have an implicit universal quantifier that handles arbitrary instances of the message bus component.

Finally, the complexity of computations appears in two different aspects of our formalization. First, it appears in the definition of operators used inside Featherweight OCL

itself; and second, the computational complexity appears in the definitions of custom functions with which we extend OCL in Listing 6 with `simulate()`. This second aspect lies largely outside the scope of the proofs; instead, it is part of the effort required for the larger formalization work. The computational complexity in the definition is more concrete: if we substitute a more complex computation for the simple condition of equality on protocols in invariant Listing 5, then it needs to be discharged in the corresponding proof.

## V. Discussion

We explored the idea that the complexity of consistency specifications is reflected in the structure of their formal proofs. By translating OCL-like constraints into Isabelle/HOL, we examined how dimensions such as arity, aggregation, and computational content impact the size and shape of resulting proof obligations. These preliminary results suggest that formal verification tools can also be used to analyze and quantify the modeling effort associated with managing consistency.

The proposed approach provides a formal lens to reason about consistency beyond informal or tool-specific interpretations. By grounding consistency rules in Isabelle/HOL, we obtain rigid, unambiguous, machine-checkable specifications. This ensures syntactic well-formedness, reveals implicit assumptions, and opens the door to proof-based analysis of specification structure. We further illustrated how complexity evolves during a staged modeling process in Sect. III, providing an early demonstration of how proof complexity can reflect modeling choices.

Several limitations constrain the generality of our results. First, the models and consistency specifications are intentionally simplified to keep the encoding tractable. As such, our findings should be understood rather exploratory than conclusive. Second, the observed complexity is influenced not only by the intrinsic properties of the models but also by artifacts of the encoding (e.g., use of Featherweight OCL) and by Isabelle's logic itself. Distinguishing essential complexity from accidental overhead remains an open challenge. We also encountered limitations in the current tool support. The translation of UML and OCL to Isabelle/HOL is largely manual, and existing translation approaches into the Rocq prover [22] are not actively maintained. Automating this translation pipeline can reduce the encoding effort and also enable an application of proof-based consistency analysis in practice.

Despite these challenges, we believe that formalization brings significant value. Beyond correctness, formalization opens opportunities for quantitative metrics. Measures such as proof size, depth, and dependency structure [23] may serve as proxies for complexity and highlight regions in a model that require simplification or refinement. Empirical validation of the approach will require additional models to derive and validate meaningful metrics, and the application to real-world systems is a promising direction for future work. Our results illustrate the potential of combining formal verification with model-driven engineering to reason about the structure and maintainability of consistency specifications. While challenges remain in scalability, automation, and distinguishing essential from accidental complexity, our work points toward a richer understanding of consistency management as a modeling activity that integrates formal verification.

## VI. Related work

A fine-grained examination of complexity requires proper metrics. Object-oriented software design metrics have been explored [24]–[26], while size-related modeling metrics have also been studied [15]. Object-oriented software measures [27] have also influenced the evaluation of formal proof complexity [23]. More broadly, the complexity of formal reasoning systems has been studied through the lens of proof complexity by Cook and Reckhow [28], [29]. Complementary work, such as that by Heijstek et al. [30], evaluates the complexity of distributed modeling processes by aggregating metrics per model type. Software complexity also links to cognitive weight [31]. Consistency in model-driven engineering is often related to model synchronization [32]–[34]. Model transformations, especially bidirectional transformations (BX), enable a consistent propagation of changes between models. A *lens* is an asymmetric BX where one model (the view) is derived from another (the source) and changes to the source are reflected in the view [35]. BX approaches provide formal specifications for consistency and repair between metamodel pairs [36].

Intra-model consistency and well-formedness can be expressed and checked using OCL. Early work by Chiorean et al. [37] introduced OCL-based consistency specifications. Moreover, Bodeveix et al. proposed extensions for verifying UML model consistency [38]. Mapping OCL constraints onto graph conditions enables automated verification using attributed typed graph rewriting [39]. Other approaches have also been proposed to integrate heterogeneous models. Triple-graph grammars provide a systematic framework for maintaining consistency between multiple related models and have been applied to incremental consistency management [32], [40]. Similarly, comprehensive systems aim to provide integrated solutions for managing multiple interrelated heterogeneous models in a coherent manner [41], [42].

In view-based development, consistency is closely linked to how information is distributed across views, which can be done with a projective or synthetic approach [43]. Projective methods derive (user-requested) views on demand from a centralized *Single Underlying Model (SUM)* [44], which is redundancy free and internally consistent by design. Synthetic approaches encode the full system across overlapping views, requiring explicit pairwise consistency relations. Consistency is then preserved via model-to-model transformations, such as bidirectional transformations. The *Virtual Single Underlying Model (V-SUM)* [10] blends both paradigms and presents itself as projective, but internally consists of overlapping and redundant models. The internal models must be actively kept consistent.

We considered consistency as a relation, such that models are either consistent or not. In most frameworks, consistency is viewed as a syntactic property, but it can also rely on

semantics [9]. While this qualitative perspective enables formal repair and enforcement, temporary inconsistencies may be tolerated to avoid information loss [45]. Consistency can also be considered quantitatively, for instance, by counting constraint violations [46], [47], allowing graded analysis and change tracking. Such a notion of quantitative consistency can be linked to the gradual notion proposed by Stevens [48] where the degree of agreement between models is captured by a value in a partially ordered set or lattice.

## VII. Conclusion

We analyzed the complexity of consistency specifications using OCL-like specifications and translated them into formal proof obligations to assess their proof complexity in formal verification. We showed that complex consistency constraints can be decomposed into simpler, more manageable parts. We also discussed some limitations of OCL that prevent the direct specification of consistency across (meta-)models, which underscores the relevance of more expressive approaches. The core of our work concerns heterogeneous models with overlapping information that must be kept consistent. In this context, a proper analysis of the complexity of consistency should consider both the consistency specification and the collection of the involved models. In fact, understanding this particular notion of complexity is relevant for all practitioners who face the challenges of consistency in system development. Our work is a first step toward a deeper understanding of consistency specifications and their formal verification.

We aim to validate our complexity assessment through a real-world case study, which is underway in the context of an acknowledged nationally funded interdisciplinary research project. Therein, we want to empirically show which dimensions contribute to the complexity of consistency, either accidentally or essentially. Such a case study would enable an analysis of the effects of modeling techniques and paradigms on the complexity of consistency, thus offering guidance on more manageable consistency specifications. On the formal side, we plan to further investigate the formalization of consistency as supported by the *Vitruv* platform. This necessitates the currently missing tooling that automates reasoning and verification techniques on consistency specifications between models. By combining theoretical and empirical perspectives, our research opens the way for more effective and scalable consistency management in model-driven engineering and, potentially, cyberphysical systems.

## References


[1] H. Stachowiak, *Allgemeine Modelltheorie*. Wien; New York: Springer, 1973.

[2] I. David, H. Vangheluwe, and E. Syriani, "Model consistency as a heuristic for eventual correctness," *Journal of Computer Languages*, vol. 76, p. 101223, 2023.

[3] H. Bowman, M. Steen, E. Boiten, and J. Derrick, "A formal framework for viewpoint consistency," *Formal Methods in System Design*, vol. 21, no. 2, pp. 111–166, 2002.

[4] J. Cederbladh, A. Cicchetti, and J. Suryadevara, "Early validation and verification of system behaviour in model-based systems engineering: A systematic literature review," *ACM Transactions on Software Engineering Methodology*, vol. 33, no. 3, pp. 81:1–81:67, 2024.

[5] Object Management Group, Inc. (OMG), "Object constraint language (OCL)," Object Management Group, Inc. (OMG), Tech. Rep., 2014. [Online]. Available: https://www.omg.org/spec/OCL/2.4

[6] A. Brucker, F. Tuong, and B. Wolff, "Featherweight OCL: a proposal for a machine-checked formal semantics for OCL 2.5," *Archive of Formal Proofs*, January 2014, https://isa-afp.org/entries/Featherweight_OCL.html, Formal proof development.

[7] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[8] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[9] R. Pascual, B. Beckert, M. Ulbrich, M. Kirsten, and W. Pfeifer, "Formal foundations of consistency in model-driven development," in *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2024). Specification and Verification*, T. Margaria and B. Steffen, Eds. Springer Nature Switzerland, 2025, pp. 178–200.

[10] H. Klare, M. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner, "Enabling consistency in view-based system development — the Vitruvius approach," *Journal of Systems and Software*, vol. 171, no. 110815, 2021.

[11] A. Finkelstein, "A foolish consistency: Technical challenges in consistency management," in *Database and Expert Systems Applications*. Springer, 2000.

[12] F. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Information and Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009.

[13] P. Stevens, "Maintaining consistency in networks of models: bidirectional transformations in the large," *Software and Systems Modeling*, vol. 19, pp. 39–65, 2020.

[14] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[15] C. Lange, "Model size matters," in *International Conference on Models in Software Engineering, Workshops and Symposia at MoDELS*, ser. Lecture Notes in Computer Science, T. Kühne, Ed., vol. 4364. Springer, 2006, pp. 211–216.

[16] F. Brooks, "No silver bullet – Essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[17] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," *Software & Systems Modeling*, vol. 7, no. 3, pp. 345–359, 2008.

[18] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[19] M. H. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.

[20] E. Franconi, A. Mosca, X. Oriol, G. Rull, and E. Teniente, "$\text{Ocl}_{\text{fo}}$: first-order expressive OCL constraints for efficient integrity checking," *Software and Systems Modeling*, vol. 18, no. 4, pp. 2655–2678, 2019.

[21] T. Ali, M. Nauman, and M. Alam, "An accessible formal specification of the UML and OCL meta-model in Isabelle/HOL," in *11th IEEE International Multitopic Conference (INMIC 2007)*, J. Zaidi and A. Chughtai, Eds. IEEE, 2007.

[22] F. Sheng, H. Zhu, and Z. Yang, "Towards the mechanized semantics and refinement of UML class diagrams," in *26th Asia-Pacific Software Engineering Conference (APSEC 2019)*. IEEE, 2019, pp. 47–54.

[23] D. Aspinall and C. Kaliszyk, "Towards formal proof metrics," in *19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016), held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2016)*, ser. Lecture Notes in Computer Science, P. Stevens and A. Wasowski, Eds., vol. 9633. Springer, 2016, pp. 325–341.

[24] H. Ma, W. Shao, L. Zhang, Z. Ma, and Y. Jiang, "Applying OO metrics to assess UML meta-models," in *7th International Conference on the Unified Modelling Language: Modelling Languages and Applications (UML 2004)*, ser. Lecture Notes in Computer Science, vol. 3273. Springer, 2004, pp. 12–26.

[25] M. Lorenz and J. Kidd, *Object-oriented software metrics - a practical guide*. Prentice Hall, Inc., 1994.

[26] S. Purao and V. Vaishnavi, "Product metrics for object-oriented systems," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 191–221, 2003.

[27] C. Archer and M. Stinson, "Object-oriented software measures," Defense Technical Information Center (DTIC), Tech. Rep., apr 1995.

[28] S. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71, M. Harrison, R. Banerji, and J. Ullman, Eds. Association for Computing Machinery, 1971, pp. 151–158.

[29] S. Cook and R. Reckhow, "The relative efficiency of propositional proof systems," *The Journal of Symbolic Logic*, vol. 44, no. 1, pp. 36–50, 1979.

[30] W. Heijstek and M. R. Chaudron, "Empirical investigations of model size, complexity and effort in a large scale, distributed model driven development process," in *35th Euromicro Conference on Software Engineering and Advanced Applications*, 2009, pp. 113–120.

[31] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," *Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, pp. 69–74, 2003.

[32] H. Giese and R. Wagner, "Incremental model synchronization with triple graph grammars," in *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, ser. Lecture Notes in Computer Science, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., vol. 4199. Springer, 2006, pp. 543–557.

[33] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei, "Towards automatic model synchronization from model transformations," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. Association for Computing Machinery, 2007, pp. 164–173.

[34] H. Giese, S. Hildebrandt, and S. Neumann, "Model synchronization at work: Keeping SysML and AUTOSAR models consistent," in *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, Eds. Springer, 2010, pp. 555–579.

[35] A. Bohannon, J. Foster, B. Pierce, A. Pilkiewicz, and A. Schmitt, "Boomerang: Resourceful lenses for string data," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '08. Association for Computing Machinery, 2008, pp. 407–419.

[36] P. Stevens, "Bidirectional model transformations in QVT: Semantic issues and open questions," *Software & Systems Modeling*, vol. 9, no. 1, pp. 7–20, 2010.

[37] D. Chiorean, M. Pasca, A. Cârcu, C. Botiza, and S. Moldovan, "Ensuring UML models consistency using the OCL environment," *Electronic Notes in Theoretical Computer Science*, vol. 102, pp. 99–110, 2004.

[38] J. Bodeveix, T. Millan, C. Percebois, C. Le Camus, P. Bazex, and L. Feraud, "Extending OCL for verifying UML models consistency," Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Tech. Rep. 2002:06, 2002.

[39] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer, "Consistency checking and visualization of OCL constraints," in *UML 2000 – Third International Conference on the Unified Modeling Language: Advancing the Standard*, ser. Lecture Notes in Computer Science, A. Evans, S. Kent, and B. Selic, Eds., vol. 1939. Springer, 2000, pp. 294–308.

[40] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr, "Efficient model synchronization with view triple graph grammars," in *10th European Conference on Modelling Foundations and Applications (ECMFA@STAF 2014)*, ser. Lecture Notes in Computer Science, J. Cabot and J. Rubin, Eds., vol. 8569. Springer, 2014, pp. 1–17.

[41] P. Stünkel, H. König, Y. Lamo, and A. Rutle, "Comprehensive systems: A formal foundation for multi-model consistency management," *Formal Aspects of Computing*, vol. 33, no. 6, pp. 1067–1114, 2021.

[42] F. Golra, A. Beugnard, F. Dagnat, S. Guérin, and C. Guychard, "Addressing modularity for heterogeneous multi-model systems using model federation," in *Companion Proceedings of the 15th International Conference on Modularity (Modularity '16)*, L. Fuentes, D. Batory, and K. Czarnecki, Eds. Association for Computing Machinery, 2016, pp. 206–211.

[43] C. Atkinson, C. Tunjic, and T. Moller, "Fundamental realization strategies for multi-view specification environments," in *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. IEEE Computer Society, 2015, pp. 40–49.

[44] C. Atkinson, D. Stoll, and P. Bostan, "Orthographic software modeling: A practical approach to view-based development," in *Evaluation of Novel Approaches to Software Engineering*, L. Maciaszek, C. González-Pérez, and S. Jablonski, Eds. Springer, 2010.

[45] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multiperspective specifications," *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 569–578, 1994.

[46] K. Kegel, S. Götz, R. Marx, and U. Aßmann, "A variance-based drift metric for inconsistency estimation in model variant sets," in *20th European Conference on Modelling Foundations and Applications*. JOT, 2024.

[47] J. Kosiol, D. Strüber, G. Taentzer, and S. Zschaler, "Sustaining and improving graduated graph consistency: A static analysis of graph transformations," *Science of Computer Programming*, vol. 214, p. 102729, 2022.

[48] P. Stevens, "Bidirectionally tolerating inconsistency: Partial transformations," in *Fundamental Approaches to Software Engineering*, S. Gnesi and A. Rensink, Eds. Springer, 2014, pp. 32–46.