A Formal Embedding for Assessing the Complexity of Model Consistency

Romain Pascual
romain.pascual@centralesupelec.fr
University of Paris-Saclay
Arne Lange
Karlsruhe Institute of Technology
Michael Kirsten
Ludwig-Maximilians-Universität München
Terru Stübinger
Karlsruhe Institute of Technology
Lars König
Karlsruhe Institute of Technology
Thomas Weber
Karlsruhe Institute of Technology

Research Article

Keywords: Model Consistency, Complexity, Formal Proofs, Formal Semantics

Posted Date: March 20th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-8978072/v1

Additional Declarations: No competing interests reported.

A Formal Embedding for Assessing the Complexity of Model Consistency

Romain Pascual[1*], Arne Lange[2], Michael Kirsten[3,2], Terru Stübinger[2], Lars König[2], Thomas Weber[2]

[1]MICS, CentraleSupélec, Université Paris-Saclay, 9 rue Joliot Curie, Gif-sur-Yvette, 91190, France.
[2]KASTEL Security Research Labs, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, Karlsruhe, 76131, Germany.
[3]Institute for Informatics, Ludwig-Maximilians-Universität München, Oettingenstraße 67, Munich, 80538, Germany.

*Corresponding author(s). E-mail(s): romain.pascual@centralesupelec.fr; Contributing authors: arne.lange@kit.edu; michael.kirsten@ifi.lmu.de; stuebinger@kit.edu; lars.koenig@kit.edu; thomas.weber@kit.edu;

Abstract

Modeling and model-driven processes offer abstraction to cope with the increasing complexity of systems. Since federated models describe overlapping aspects of the same system, some information is shared, introducing redundancy. Maintaining consistency of such information is crucial to ensure a coherent system representation. In safety-critical domains, such consistency requirements often require formal verification to ensure strong correctness guarantees. However, the verification effort is influenced not only by the models themselves, but also by the structure and expressiveness of the consistency specifications. In this article, we examine the complexity of consistency from a formal perspective. We embed OCL-based consistency constraints into higher-order logic using the theorem prover Isabelle/HOL and analyze the resulting proof obligations. By identifying key dimensions that influence the verification effort, we aim to understand how the design of consistency specifications affects the formal reasoning required to assess them.We illustrate the approach via a case study of a car braking system, for which we construct a mechanized formalization of realistic metamodels and consistency constraints, and discuss metrics for their proof complexity. Understanding which structural aspects of consistency influence the verification effort provides a foundation for ultimately reducing unnecessary complexity while ensuring the required consistency constraint.

Keywords: Model Consistency, Complexity, Formal Proofs, Formal Semantics

1 Introduction

Modern software and systems engineering mitigates complexity by structuring development around pragmatic abstractions as models (Stachowiak 1973). Models capture relevant information about the domain in which the system operates and are progressively refined in the engineering process as additional details become available. Since system development is an inherently collaborative effort, multiple stakeholders, developer teams, or organizations construct related and often overlapping models that describe different views of a system. Ensuring that these models remain *consistent* is therefore essential to maintain the integrity and reliability of an engineering process (David et al. 2023). Each model captures a specific concern to ensure a separation of responsibilities during development. Hence, only the entire collection of models provides a complete description of the intended system. Models typically specify properties that overlap in their meaning or interact in their behavior. Therefore, model *consistency* becomes a prerequisite for their *joint realizability* (Bowman et al. 2002). Generally, any two models are consistent if they do not contradict each other, and otherwise, 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 results in a correct and semantically feasible system. Consistency checking is therefore both an integration prerequisite and a lightweight verification (David et al. 2023), similar to early formal validation (Cederbladh et al. 2024). Yet, the management of consistency for practical systems is challenging. Existing approaches range from lightweight checks to full-fledged formal verification, and offer only 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? Which 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 article, we introduce an exploratory method for characterizing the complexity of consistency. We observe that many consistency constraints between modeling artifacts can be abstractly captured by fragments of the Object Constraint Language (OCL) (Object Management Group, Inc. (OMG) 2014). We employ the formal framework Featherweight OCL (Brucker et al. 2014), a shallow embedding of OCL into higher-order logic. Therein, we transcribe the consistency constraints into the theorem prover Isabelle/HOL (Nipkow et al. 2002). This encoding makes the verification process explicit and allows us to integrate and *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 of a system 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 (Chidamber and Kemerer 1994). 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 properties of models and constraints affect complexity the most.

Our contributions are as follows:

- We turn consistency complexity into a *measurable* property through rigid formalization, by an encoding into Isabelle/HOL via the Featherweight OCL framework.
- We identify and analyze key dimensions of consistency complexity using a working example, focusing on proof obligations and logical structure.
- We detail the formal encoding into Isabelle/HOL via Featherweight OCL and the analysis in a case study about a realistic car braking system.
- 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 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 that they induce.

2 Foundations

2.1 Model Consistency

Model consistency is a fundamental concern in model-driven and model-based engineering. At an abstract level, consistency boils down to a relation: models are either consistent or not (Pascual et al. 2025a). More practically, *consistency* means that two or more related models, e.g., connected through refinement, projection, or cross-domain mappings, do not contradict each other and can be realized together. There are various approaches in model-based engineering to specify consistency relations (Klare et al. 2021; Bowman et al. 2002; Finkelstein 2000; Lucas et al. 2009; Stevens 2020). However, practical encoding of consistency often relies on formal specifications in an appropriate language, for example, in the Object Constraint Language (OCL) (Object Management Group, Inc. (OMG) 2014) or in model transformation languages via consistency-preservation rules (Czarnecki and Helsen 2006). 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 standardized way to express model constraints.

In the literature, there are more refined notions of consistency, for instance, describing gradual or temporal consistency (see Sect. 7). However, we focus on a simple setting for clear detection and analysis of consistency violations.

2.2 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 (Lange 2006). This principle carries over to modeling, where size (e.g., amount of elements, expressions, or constraints) is often a proxy for complexity.

A key distinction lies between *essential* and *accidental* complexity (Brooks 1987; Atkinson and Kühne 2008). 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 connected domains and complexity that could be reduced by better abstractions or tooling.

More sophisticated metrics are, e.g., McCabe's cyclomatic complexity (McCabe 1976) or Halstead's effort metrics (Halstead 1977). However, we focus on size-based measures as a practical and implementation-independent first approximation that enables general reasoning about the difficulty of understanding and maintaining consistency specifications.

2.3 Isabelle/HOL and Featherweight OCL

To analyze requirements using formal verification, they must be expressed as precise logical specifications. These logical formulas are meant to encode informally stated requirements. In our case, the requirement is model consistency. Therefore, we need to formally encode the description of models and their consistency relations over a given collection of models. The question here is not only how to check consistency, but also how to formalize it in the first place, such that verification reflects the modeling semantics rather than the artifacts of the encoding.

Since we express the consistency specifications in OCL, we need a translation of OCL into some logic that is suitable for formal verification. Many approaches translate OCL into first-order logic (FOL) (Beckert et al. 2002; Clavel et al. 2009; Demuth and Wilke 2009), see Rajic and Sruk (2024) for a complete survey. These encodings are typically optimized for specific tasks, such as deductive software verification with KeY (Ahrendt et al. 2016), or unsatisfiability checking or extraction of SQL statements to exploit the semi-decidability of first-order logic (Clavel et al. 2009). In contrast, *Featherweight OCL* (Brucker et al. 2014) provides a shallow embedding of OCL into the theorem prover Isabelle's native Higher-Order Logic (HOL) (Nipkow et al. 2002) instead of formalizing and rebuilding OCL from the ground up (Steimann et al. 2023). Isabelle/HOL provides a generic infrastructure for implementing deductive systems in higher-order logics (HOL) and supports structured, human-readable, and machine-checked correctness proofs. HOL supports definitions of very expressive, rigorous, and general theorems that are re-checked and confirmed by Isabelle through a small and well-established trusted logical foundation. Featherweight OCL builds on top of Isabelle by providing a shallow embedding of a substantial fragment of the OCL standard into HOL. Rather than translating OCL into a different logical paradigm, it preserves OCL's semantic structure within Isabelle's logic. This design choice by Featherweight OCL, originally designed to clarify ambiguities in the OCL standard (Brucker et al. 2006) allows for a native formalization and verification of UML models and OCL constraints that closely mirrors their original form and makes them explicit in theorems and proofs.

Having formalized consistency specification, the realization of such properties on a concrete set of models can be verified using various techniques: from model checking (Clarke et al. 2018) and abstract interpretation (Cousot and Cousot 1977) to contract-based reasoning (Meyer 1992), or interactive theorem proving (Paulson 1989). Since Featherweight OCL is deeply linked to Isabelle/HOL, we propose to employ interactive theorem proving, e.g., using Isabelle/HOL for the verification task and provide exemplary proof sketches. This choice directly supports our objective: the OCL constraints introduced in Sect. 3 translate closely into their formal counterparts presented in Sect. 4. The rigidness of Featherweight OCL binds us to the official standard of OCL, where more liberal interpretations are often used in software engineering. Moreover, the formal mechanization of Isabelle requires type annotations for operations that are overloaded across OCL collection types but correspond to distinct operators in Featherweight OCL.

3 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 organization defines and refines their own metamodels in order to describe their own excerpt of 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 that is collaboratively developed 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. The process of establishing the consistency specifications is illustrated in Figure 1. Our example illustrates such model refinements and the resulting overlaps, with the stated aim of discussing and assessing the overlaps' associated complexity.

We use these examples to address the first challenge: assessing consistency and the factors or dimensions that contribute to its complexity. We restrict our study to the technical space of the consistency specifications, leaving out the cognitive difficulty of creating models and the intrinsic complexity of the modeling process. Before we proceed, we define two notions required to describe consistency specifications: *correspondence* and *coextension*. Correspondence is a 1-to-1 relationship between model elements that belong to different metamodels. It expresses that these elements represent the same, or parts of the same, conceptual entity within a shared semantic space for the purpose of checking their consistency. Given a correspondence relation, we use the

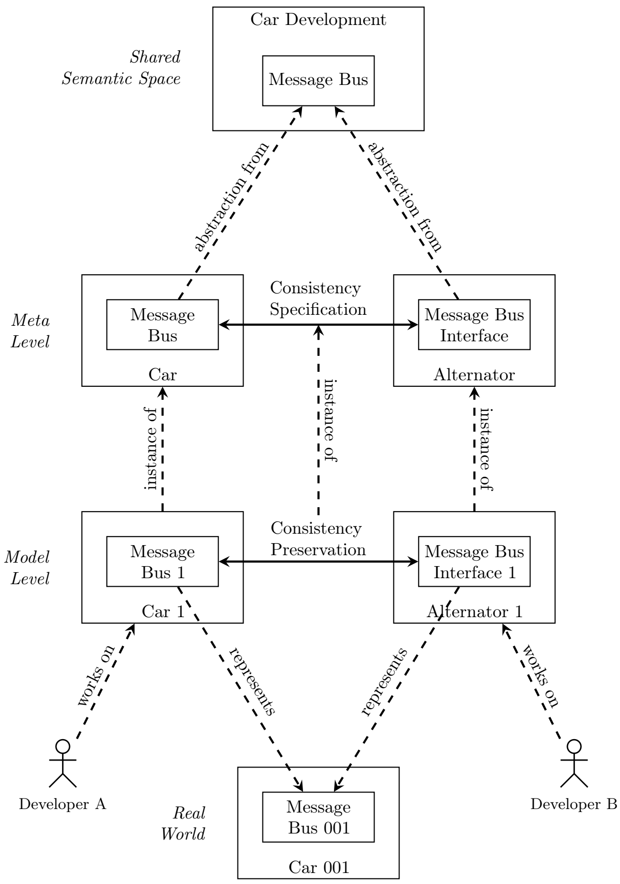

Figure 1 Process of collaboratively developing a car with two organizations, Car manufacturer and Supplier, each having their own metamodels and models, while sharing a common semantic space for consistency. The metamodel elements *Abstract Message Bus* and *Message Bus Interface* are abstracted to the shared semantic element *Message Bus* that is used to define a consistency specification between the two metamodel elements. The metamodels are then instantiated to create specific models, *Car 1* and *Light Machine 1*, each containing instances of the message bus concepts.

term *coextension* to denote the induced relation on metamodel elements, i.e., the set of instances that correspond to a given element with respect to consistency. We denote coextension by the $\sim$ symbol. We also use $\sim$ to denote the query operator that, for a given model element, returns its coextending elements. Coextension means that metamodel elements overlap semantically, i.e., share a semantic space where instances represent the same, or parts of the same, original (Stachowiak 1973). They produce instances that are not changeable in isolation, but instead require changing another instance as well. We view this operator as part of the specification of cross-model consistency. This abstraction deliberately hides the concrete implementation of correspondences, such as the use of identifiers or trace links, in order to reduce specification complexity. For our purposes, the coextension operator can be understood as a binary function defined over the given correspondence relation.

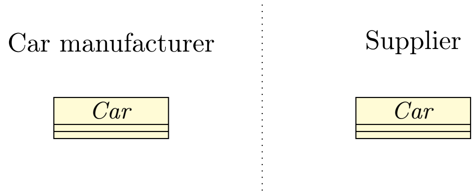

Figure 2 The Car manufacturer and Supplier both want to build the same car, thus their modeling starts with the most basic abstraction, i.e., a *Car*.

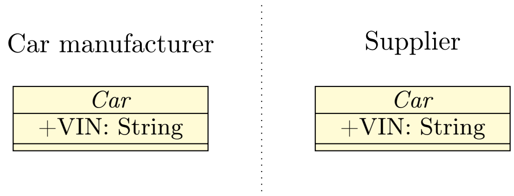

Figure 3 *Car*s with structural features, like *VIN*.

In the running example of Figure 1, the metamodel elements *Abstract Message Bus* (on the side of Car manufacturer) and *Message Bus Interface* (on the side of Supplier) are in a 1-to-1 correspondence, as both abstract the same conceptual message bus in the shared semantic space. At the model level, however, this correspondence induces a coextension relation between concrete instances. For instance, the message bus instance *Message Bus 1* in model *Car 1* may coextend with one or more message bus interface instances in *Light Machine 1*, depending on how the supplier refines their model. The coextension operator $\sim$ abstracts over these realizations and allows consistency specifications to range over all such corresponding instances without committing to a particular implementation of the correspondence.

In the following, 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.

3.1 Arity of the Consistency Specification

The Car manufacturer and the Supplier start with a trivial model of a car that consists only of the car itself, with no further elements, as illustrated in Figure 2. 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 3, 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

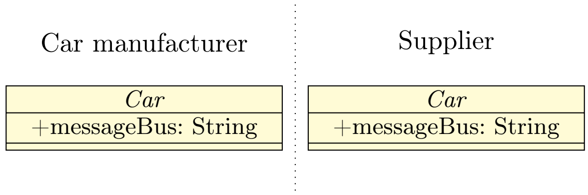

Figure 4 The *Car* will have components communicating over a *messageBus*.

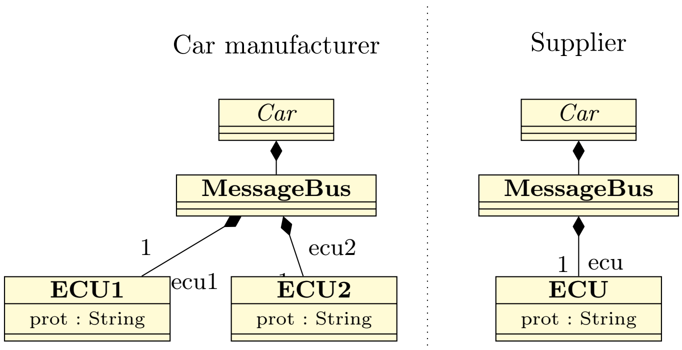

Figure 5 Multiple types of Electronic Control Units (n-1 on meta).

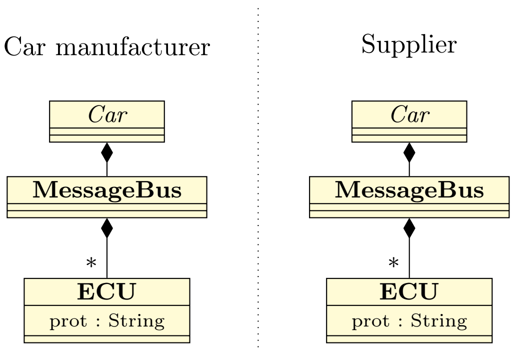

Figure 6 Multiple Electronic Control Units (n-m on model).

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 4 with the *messageBus* as String, and in Figure 5 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 that it supplies to Car manufacturer. In our simplified example from Figure 5, Car manufacturer provides two components that use the message bus, while Supplier provides only one component.

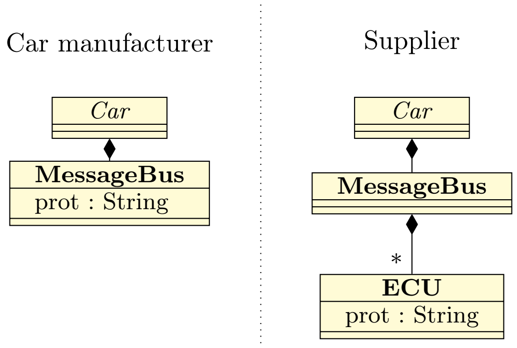

Figure 7 The *MessageBus* is modeled explicitly, as it is too complex to be described by a simple type like String (1-n on model).

Car manufacturer might not even need to model the *ECUs*, if only Supplier has access to the *MessageBus* and if the relevant aspects to be modeled are limited to the message bus and its communication protocol, e.g., 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 5. 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, or 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 Figure 6. 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 7 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.

3.2 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, in the following denoted as "∼". 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 which captures whether two elements are in correspondence with each other, that is, whether they need to be kept consistent.

In essence, the "select(∼)" operator hence describes the set of coextended pairs of 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 maintaining consistency between these coextending elements. This abstraction allows us to focus on dimensions independently of the concrete consistency rule and the involved elements. We discuss that influence later in Sect. 3.3. In our example from Figure 2, 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 7, 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 5. 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 5

3.3 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 by a logical formula. 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., for *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 more 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" that 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 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 the example from 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 (Franconi et al. 2019).

```
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.

3.4 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 ECUs.

Lastly, breaking down a consistency specification does not necessarily reduce 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.

4 Influence of Complexity Dimensions on 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. 1, 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 (Ali et al. 2007) or in the Rocq prover (Sheng et al. 2019).

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 (Aspinall and Kaliszyk 2016). 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. 3. 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.

4.1 Formalizing Consistency via Coextension on Models

To formally establish consistency between models, we first need to define a formal consistency relation that spans multiple metamodels, but that can also be evaluated at the level of individual elements. The difficulty lies in the granularity of the relation. In the simplest case, checking consistency reduces to comparing primitive type values (e.g., `String` or `Integer`) of the corresponding elements. However, coarser consistency notions may simply require agreement on the number of instances of a given type. Besides, no intrinsic comparison may exist, e.g., for abstract model elements that are not represented by a primitive value. Typical cases involve elements whose structure is too complex or that are only described by informal domain knowledge. In such cases, consistency must rely on an a priori correspondence that indicates which elements represent the same conceptual entity.

Therefore, we assume a predefined correspondence relation between model elements. As in Sect. 3, we write the induced coextension operation as $\sim$ in order to query the correspondence relations between model elements across different models and metamodels. In particular, a and b in the expression $a \sim b$ belong to different metamodels. This immediately raises a technical issue since standard OCL expressions only range over a single model and cannot directly express cross-model properties.

Our solution is to interpret the various models as disjoint parts of a single larger universe. Moreover, we ensure that elements from different metamodels remain type-separated by omitting the shared `OclAny` supertype from the standard top type of the OCL type hierarchy. The coextension operator then acts as the only bridge between these universes in order to provide controlled means to express cross (meta)model relations such as consistency. In Isabelle/HOL, we implement the coextension operator $\sim$ as an overloaded constant that can be used with any expression of the overloading type that is associated with the different metamodels. For example, two different metamodels that represent cars may relate their `Car` class instances via an a priori lookup-up table that encodes the following correspondence in Isabelle:

```
definition correspondences_car :: "(𝒯_Car1 × 𝒯_Car2) set"
        ("∼_car" 100) where
  "correspondences_car ≡ {(car1_1, car2_1), (car1_2, car2_2)}"

definition coext_car :: "[Car1, Car2] ⇒ Boolean"  (infixl "∼_car" 100) where
  [code_unfold]: "a ∼_car b ≡
      (λ τ :: 𝔄 st. ⌊⌊ (⌈⌈a τ⌉⌉, ⌈⌈b τ⌉⌉) ∈ correspondences_car ⌋⌋)"
```

In Figure 2, $\sim$ uses only object identities and the correspondence set, since there are no primitive values. For analyzing the invariant given in Listing 1, we rely on the correspondence and coextension introduced above, which provide the set of related `Car` instances across the two metamodels. Moreover, we translate the invariant of Listing 1 to the following Isabelle/HOL definition using Featherweight OCL:

```
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 ≜ to distinguish it from Isabelle/HOL's general equality relation. The Isabelle proof assistant parses and type-checks this definition, and thereby guarantees that it both is syntactically correct in terms of Featherweight OCL's formalization and that it semantically agrees with the metamodels in Figure 4.

As in Listing 2, we can also use structural features such as the VINs of Figure 3 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 7, we need to generalize the coextension relation further, which enables us to work with correspondences on more than just one component. In the example, we can introduce the two distinct concrete operators $\sim_{\mathrm{Car}}$ and $\sim_{\mathrm{MB}}$ which consider the correspondence of cars with message buses, respectively. Hence, the bare $\sim$ operator 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.

Before introducing Listing 3, we clarify the semantics of the operation `.select(~)`. Rather than testing whether two elements are related by a correspondence, it *collects* all elements that are in coextension with the elements on which it is called by performing a relational navigation across the models. A naive encoding can be obtained from a combination of `.select()` and `.allInstances()` that enumerates all instances and filters those that are related by $\sim$:

```
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)))"
```

This example reveals that `.select(~)` is not a simple syntactic abbreviation. The construction requires knowledge of the target class name `MessageBus2`, since the operator must know over which universe it quantifies. Without this information, the expression would be ill-defined: the correspondence relation alone does not determine which model elements should be considered. Therefore, we treat the coextension as an intrinsically typed operation and introduce a family of concrete `.select(~`$_{class}$`)` operations parameterized by the target class. This makes the quantification domain explicit and ensures well-defined cross-model navigation.

4.2 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 2.

Inside Featherweight OCL, such an instance is called a *heap state*, which we denote as τ. The relation $\tau \models e$ expresses that the OCL expression e will evaluate to `True` under the heap state τ. 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 α_{inv} on a, the statement $\tau \models \alpha_{\text{inv}}(a)$ holds.

4.3 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 in Sect. 3 into a formal semantics that is required for formal verification. As a first step, we consider again the invariant in Listing 2:

```
lemma one_to_one_refined_lemma:
  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 a bit 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 now contrast the above proof with a similar proof for the more complex invariant in Listing 5:

```
lemma one_to_n_{lemma}:
  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.

For proving 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. 3.4, 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, but merely makes its 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 that are used inside Featherweight OCL itself. Second, the computational complexity appears in the definitions of custom functions with which we extend OCL in Listing 6 by `simulate()`. This aspect lies largely outside the scope of the proofs, and is instead part of the effort 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.

The whole formalization without the proof outlines and constraints comprises about 670 lines of formalization code for the fully abstract metamodel, 768 for the refined metamodel with `VINs`, and 1689 for the metamodel that contains the `MessageBus`. For each, the constraints and proof outlines are around 100 lines of formalization in Isabelle.

5 Application to the Brake Case Study

In this section, we apply our workflow to an automotive brake case study derived from (Hagel et al. 2025), in order to investigate more precisely how the structure of consistency specification influences formal verification effort. The case study consists of two models: a brake component model and CAD model that describe parametric and spatial aspects of the same brake system. The goal is to formalize their consistency and analyze the resulting proofs obligations.

5.1 Case Study Overview

The CAD metamodel (Figure 8) expresses the spatial structure of the brake, using *Namespace* elements and associated *Parameter* objects all linked to the *CAD* class as root element. The brake component metamodel (Figure 9) describes structural entities, called *BrakeComponents*, such as *BrakeDisc*, *BrakeCaliper*, *BrakePad*, and *ABSSensors*, all parts of a *BrakeSystem*, providing the root of the models.

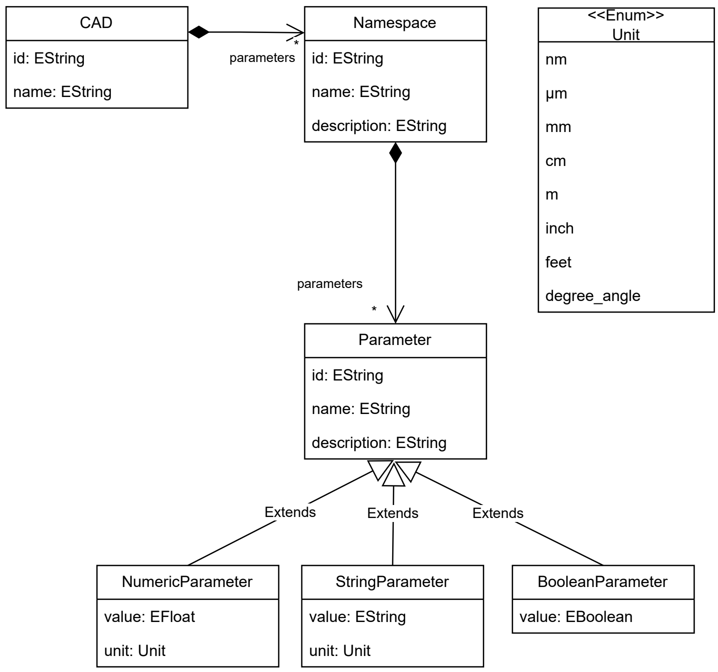

Figure 8 CAD Meta Model

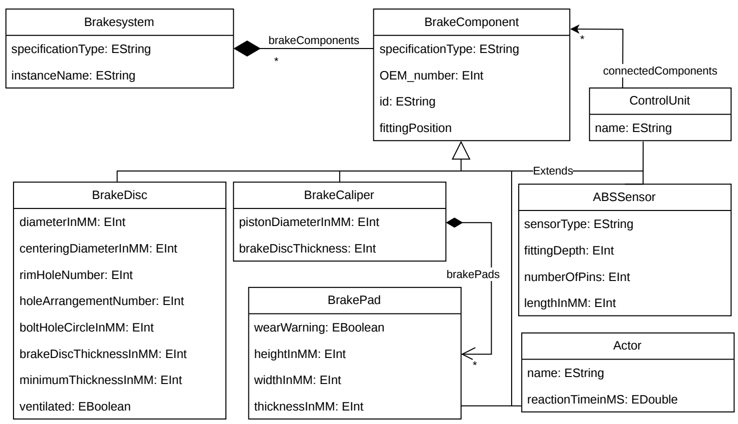

Figure 9 Brake Component Meta Model

The concrete instances of both metamodels are respectively shown in Figure 10 and Figure 11. These models overlap: individual brake components correspond to CAD namespaces, and some selected attributes must coincide across representations. Consistency between these models was originally specified using the *Reactions* language (Klare et al. 2021). We first translate these rules into OCL constraints over the combined multi-model system, before embedding them into higher-order logic.

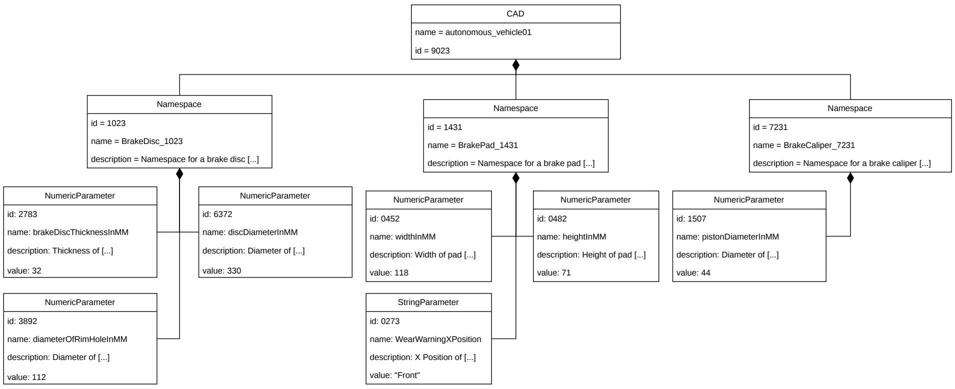

Figure 10 CAD Model

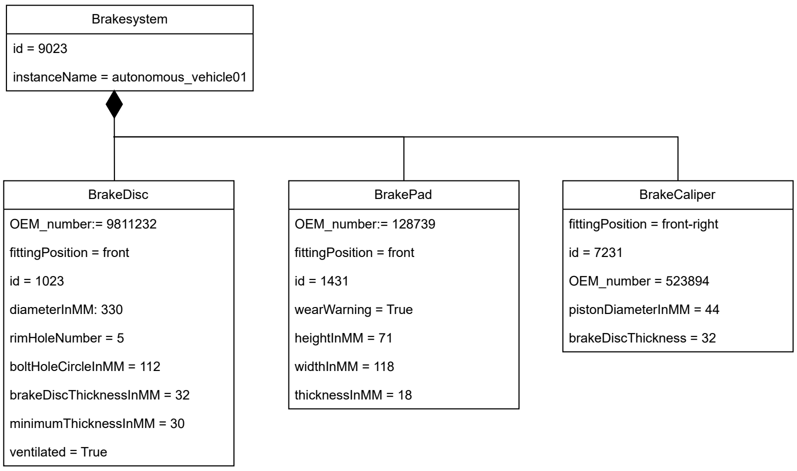

Figure 11 Brake Component Model

5.2 OCL Consistency Constraints

From the original case study (Hagel et al. 2025), we consider six consistency constraints, and discuss three representative examples here. Additional details and the other three constraints can be found in the supplementary material.

Unique Correspondence and Identifier Equality.

Each *BrakeComponent* from the component model must correspond to exactly one *Namespace* in the CAD model, and their identifiers must coincide. The OCL specification is given in Listing 7 and combines: a cardinality requirement, type filtering via `oclIsTypeOf`, and attribute comparison. Note that the comparison of the *id* with the one corresponding element, we have to invoke the `first()` collection operation because the `select($\sim$)` operation yields a collection.

```
context brakesystem::BrakeComponent
inv: self.select(~) ->
select(c|c.oclIsTypeOf(CAD::Namespace)) -> size() = 1
and self.id = self.select(~) ->
select(c|c.oclIsTypeOf(CAD::Namespace)) -> first().id
```

Listing 7 Brake Component corresponding element has to have the same id

Unique Correspondence and Identifier Equality.

The value of the attribute *OEM_ Number* in the *BrakeComponent* must be equal with the *NumericParameter*, named *OEM Number*, that is connected to the corresponding *Namespace*. The OCL specification is given in Listing 7, assuming that the constraint in Listing 7 already holds. The constraint introduces multi-step navigation, subtype reasoning anf explicit type casts. Note that there should be exactly one *Parameter* that has the name *OEM Number* and its value should be distinct.

```
context brakesystem::BrakeComponent
inv: self.select(~).parameters ->
select(p|p.name = "OEM Number").value =
self.OEM_Number
```

Listing 8 The OEM number of any Brake Component has to be the same in any corresponding modeling element

Specification Type Consistency

The value of the attribute *Specification_ Type* must coincide with the *StringParameter* of the corresponding *Specification Type*. The OCL specification in Listing 9 mirrors the previous one but omits numeric type casting.

```
context brakesystem::BrakeComponent
inv: self.select(~).parameters ->
select(p|p.name = "Specification Type").value =
self.Specification_Type
```

Listing 9 All specification type values of brake components should have the same value in the corresponding model elements

These constraints differ in structural complexity, particularly in navigation depth and type refinement, which later affects proof effort.

5.3 Formally Embedding the Models in Isabelle/HOL

The metamodels are shallowly embedded into Isabelle/HOL using Featherweight OCL. Since the Featherweight OCL framework already provides extensive machinery, we can directly start expressing the elements of our metamodel. More precisely, each UML class is represented by a HOL datatype within an object universe $\mathfrak{A}$. In a standard UML object universe, we would further include the type `OclAny`, but by its omission, we ensure that the logic does not allow for unification or any other relation of different types without us providing explicit rules that allow for it. Object identifiers (`oid`) provide unique references, and accessor functions implement the structural relations.

```
datatype 𝔄 = in_BrakeSystem 𝒯_BrakeSystem |
        in_BrakeComponent 𝒯_BrakeComponent |
        in_ABSSensor 𝒯_ABSSensor | in_BrakeCaliper 𝒯_BrakeCaliper |
        in_BrakeHose 𝒯_BrakeHose | in_BrakeDisk 𝒯_BrakeDisk |
        in_BrakePad 𝒯_BrakePad | in_CAD_Model 𝒯_CAD_Model |
        in_Namespace 𝒯_Namespace | in_Parameter 𝒯_Parameter |
        in_NumericParameter 𝒯_NumericParameter |
        in_StringParameter 𝒯_StringParameter |
        in_BooleanParameter 𝒯_BooleanParameter
```

For instance, considering the formalization of the `BrakeComponent` element, we can directly translate the names and attributes based on the HOL types `string` for a list of characters, `int` for an Integer, and `option` for the optional type.

```
datatype 𝒯_BrakeComponent = mk_BrakeComponent (brakecomponent_oid: oid)
  (brakecomponent_specificationType: "string option")
  (brakecomponent_OEM_number: "int option")
  (brakecomponent_fittingPosition: "string option")
  (brakecomponent_id: "string option")
```

On top of the above type definitions, we must further define and give basic constants and lemmas for various instantiations of Featherweight OCL's generic machinery to express strict equality, queries for `OclAsType`, `OclIsTypeOf`, `OclIsKindOf`, `OclAllInstances`, as well as specific OCL selectors and accessors. As of now, this is a very tedious manual effort that consists of around 5200 lines of Isabelle formalization, but it could potentially be largely automated and can hence be neglected by the consistency engineer. Some care needs to be taken in formalizing the inheritance relations and cardinalities of containment accessors, however.

Taking the above groundwork, we are able to construct a concrete heap state σ of a few objects and associations for brake components, brake systems, string parameters, numeric parameters, namespaces, and CAD models, on which we will exemplify our formalizations and proofs for consistency verification. This heap state is a record of one map `heap` that assigns concrete object identifiers to model elements, and one map `assocs` that assigns concrete object identifiers to a list of mappings from source to target lists that associate lists of source object identifiers to lists of target object identifiers. By design, Featherweight OCL also allows formalizing operation contracts to conjecture that some operation that is executed on a pre-state that satisfies a given precondition afterward ends in a post-state that satisfies a given post-condition. However, our case study only considers a static metamodel, and we hence use the same heap state as both pre- and post-state, yielding the state transition $\tau \equiv (\sigma, \sigma)$.

5.4 Correspondences and Coextension in Isabelle/HOL

Before encoding the consistency specifications, we clarify the formalization of the correspondences and coextension. Assuming our exemplary heap state contains the four model elements `brakecomponent_1`, `brakecomponent_2`, `namespace_1`, and `namespace_2` of the obvious types, we also define elements of the corresponding OCL types, e.g., as follows:

```
definition x_brakecomponent_1 :: "BrakeComponent" where
  "x_brakecomponent_1 ≡ λ _. ⌊⌊brakecomponent_1⌋⌋"
```

From this, we construct the following two sets of correspondence pairs between `BrakeComponent` and `Namespace`:

```
definition correspondences_bcns :: "(T_BrakeComponent × T_Namespace) set"
  where "correspondences_bcns ≡ {(brakecomponent_1, namespace_1),
                                 (brakecomponent_2, namespace_2)}"

definition correspondences_nsbc :: "(T_Namespace × T_BrakeComponent) set"
  where "correspondences_nsbc ≡ {(namespace_1, brakecomponent_1),
                                 (namespace_2, brakecomponent_2)}"
```

Now, we can defining our co-extension operators and resort againt to a static model where both elements are evaluated in the same state τ. The enclosing $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ catch evaluations on invalid or non-existing states, but we will, in the following, only employ our previously defined concrete heap state anyway.

```
definition coext_bcns :: "[BrakeComponent, Namespace] ⇒ Boolean"
    (infixl "∼_bcns" 100) where "a ∼_bcns b ≡
        (λ τ :: 𝔄 st. ⌊⌊ (⌈⌈a τ⌉⌉, ⌈⌈b τ⌉⌉) ∈ correspondences_bcns ⌋⌋)"

definition coext_nsbc :: "[Namespace, BrakeComponent] ⇒ Boolean"
    (infixl "∼_nsbc" 100) where "a ∼_nsbc b ≡
        (λ τ :: 𝔄 st. ⌊⌊ (⌈⌈a τ⌉⌉, ⌈⌈b τ⌉⌉) ∈ correspondences_nsbc ⌋⌋)"
```

With these two correspondence sets, we formalize a constant type-generic $\sim$ operator (which we internally name *squiggle*) that we overload by the above two specific co-extension operators for relating `BrakeComponent` and `Namespace`:

```
consts squiggle :: "[(𝔄, 'α option option) val,(𝔄, 'β option option) val]
                    ⇒ Boolean" (infixl "∼" 100)

overloading squiggle ≡ "squiggle :: [Namespace, BrakeComponent] ⇒ Boolean"
begin
definition squiggle_bcns[code_unfold]:
    "(a::BrakeComponent) ∼ (b::Namespace) ≡ a ∼_bcns b"
end
definition squiggle_nsbc[code_unfold]:
```

```
    "(a::Namespace) ∼ (b::BrakeComponent) ≡ a ∼nsbc b"
end
```

To show that $\sim$ behaves as intended, we can use Featherweight OCL's native **Assert** environment to evaluate for every heap state $\sigma\prime$ as both pre- and post-state under the type universe $\mathfrak{A}$ that $x_{\text{brakecomponent\_1}}$ co-extends with $x_{\text{namespace\_1}}$ as our correspondence set requires.

```
Assert "⋀ σ'::𝔄 state. (σ', σ') ⊨ x_brakecomponent_1 ∼ x_namespace_1"
```

With this complete machinery, the assertion successfully evaluates to true, and we can further, e.g., inversely validate that the following negated assertion evaluates to true as well for instances that should not co-extend, and we are now ready for actual consistency constraints.

```
Assert "⋀ σ'::𝔄 state. (σ', σ') ⊨ not (x_brakecomponent_1 ∼ x_namespace_2)"
```

5.5 Consistency Constraints in Isabelle/HOL

Each consistency rule is encoded as a mapping from `BrakeComponent` to the OCL `Boolean` type.

The first constraint (unique correspondence and identifier equality) from Listing 7 requires enumerating all `Namespace` instances via `allInstances()`, selecting those co-extending the argument `self`, proving uniqueness (cardinality is 1) via a `size` query and finally extracting the single element and comparing identifiers.

```
definition Id_IDs_inv :: "BrakeComponent ⇒ Boolean" where
  "Id_IDs_inv (self) ≡
     Namespace .allInstances()->select_Set(m | self ∼ m)->size_Set() ≜ 1 and
       (Namespace .allInstances()->select_Set(m | self ∼ m)->asSequence_Set()
         ->first_Seq().id_Namespace ≜ (self .id_BrakeComponent))"
```

The OEM constraint from Listing 8 asks that for every instance of `Namespace` that co-extends the given `BrakeComponent` instance, the name of the associated parameter that has the name "OEM Number" has the same value as the `OEM number` when typed to an Integer. Thus, the constraint additionally requires navigation over associations, subtype filtering of the queried instance as `NumericParameter` via `oclAsType` and explicit casting of the numeric value from *Real* to *Int*.

```
definition Id_OEMs_inv :: "BrakeComponent ⇒ Boolean" where
  "Id_OEMs_inv (self) ≡
    ((Namespace .allInstances()->select_Set(m|self ∼ m)
        ->asSequence_Set()->first_Seq().parameters
        ->select_Set(c | c .name_Parameter ≐ oem number)
        ->asSequence_Set()->first_Seq()
        .oclAsType(NumericParameter) .value_NumericParameter))
      ≜ ((self .OEM_number_BrakeComponent)->oclAsType_Int(Real))"
```

The last constraint (Listing 9) follows the same structure but without explicit typecast operation.

```
definition Id_Specs_inv :: "BrakeComponent ⇒ Boolean" where
  "Id_Specs_inv (self) ≡
    (Namespace .allInstances()->select_Set(m|self ∼ m)
      ->asSequence_Set()->first_Seq().parameters
        ->select_Set(c | c .name_Parameter ≐ specification type)
          ->asSequence_Set()->first_Seq().oclAsType(StringParameter)
          .value_StringParameter) ≜ (self .specificationType_BrakeComponent)"
```

5.6 Proof Characteristics

The proof `id_ids`$_{lemma}$ of the constraint `Id_IDs`$_{inv}$ is lengthier than the ones from our example on car components, as after obtaining an explicit set of co-extending instances of type `Namespace`, we obtain the unique instance as the co-extension is uniquely defined in our setting. Moreover, for showing that the string identifiers encode the same string, we again obtain an explicit set of co-extending instances of type `Namespace` of size one, and then show that the encoded string for both identifiers is the same one. Note that the occurrence of **by** *presburger* indicates that the respective proof step was found automatically using presburger logic. All the other proof steps either further decompose with another **proof** command or are not proved here, but only outlined.

```
lemma id_ids_lemma:
— Skeleton for proving the invariant (3) on arbitrary BrakeComponents
  fixes a :: "BrakeComponent"
  shows "τ ⊨ Id_IDs_inv a"
proof (unfold Id_IDs_inv_def)
  have single:
    "τ ⊨ (Namespace .allInstances()->select_Set(m | a ∼ m)->size_Set() ≜ 1)"
  proof -
    obtain ms :: "Namespace set" where "ms =
      {x. (τ ⊨ (Namespace .allInstances()->includes_Set(x)))
                ∧ (τ ⊨ a ∼ x)}"
      by presburger
    with σ_def have "card ms = 1"
    then obtain m :: "Namespace" where
      "τ ⊨ (Namespace .allInstances()->select_Set(c | c ∼ a) ≐ Set{m})"
      unfolding UML_Set.OclSelect_def
    moreover with ‹card ms = 1›
    have "τ ⊨ (Set{m}->size_Set() ≜ 1)"
    ultimately show
     "τ ⊨ (Namespace .allInstances()->select_Set(m | a ∼ m)->size_Set() ≜ 1)"
      using UML_Set.cp_OclSize
  qed
  moreover have "τ ⊨ (Namespace .allInstances()->select_Set(m|a ∼ m)
      ->asSequence_Set()->first_Seq().id_Namespace ≜ (a .id_BrakeComponent))"
  proof -
    from single obtain m :: "Namespace" where
```

```
    "τ ⊨ (Namespace .allInstances()->select_Set(m | a ∼ m) ≜ Set{m})"
  moreover have "τ ⊨ (m .id_Namespace ≜ (a .id_BrakeComponent))"
  ultimately show
    "τ ⊨ (Namespace .allInstances()->select_Set(m | a ∼ m)->asSequence_Set()
            ->first_Seq().id_Namespace ≜ (a .id_BrakeComponent))"
qed
thus "τ ⊨ (Namespace .allInstances()->select_Set(m | a ∼ m)->size_Set() ≜ 1
        and (Namespace .allInstances()->select_Set(m|a ∼ m)
            ->asSequence_Set()->first_Seq()
          .id_Namespace ≜ (a .id_BrakeComponent)))"qed
```

The proof $\texttt{id\_oems}_{lemma}$ of the constraint $\texttt{Id\_OEMs}_{inv}$ is shorter than the one above for $\texttt{Id\_IDs}_{inv}$, largely due to the fact that we make explicit use of $\texttt{Id\_IDs}_{inv}$ in the first step for showing that we have exactly one queried instance of type `Namespace` that co-extends. From that, we can automatically perform the proof for obtaining this instance of type `Namespace`. Note that the occurrences of **by** *metis* indicate that the respective proof steps were found automatically using an automated theorem prover for first-order logic. The remaining steps are obtaining the respective `Parameter` instance via the field accessor, proving that its `OEM_number` encodes the same string value, and finally deducing the same property for the top-level statement that does not use our explicitly obtained instances. These three steps are only outlined and not proved in this article.

```
lemma id_oems_lemma:
— Skeleton for proving the invariant (4) on arbitrary BrakeComponents
  fixes a :: "BrakeComponent"
  shows "τ ⊨ Id_OEMs_inv a"
proof (unfold Id_OEMs_inv_def)
  from id_ids_lemma foundation10'
  have "τ ⊨ (Namespace .allInstances()->select_Set(m|a ∼ m)->size_Set() ≜ 1)"
    unfolding Id_IDs_inv_def
    by metis
  then obtain m :: "Namespace" where
    "τ ⊨ (Namespace .allInstances()->select_Set(m | a ∼ m)
          ->asSequence_Set()->first_Seq() ≜ m)"
    using StrongEq_L_sym StrongEq_L_trans StrongEq_refl
    by metis
  then obtain p :: "Parameter" where
     "τ ⊨ (m .parameters
        ->select_Set(c | c .name_Parameter ≐ oem number) ≜ Set{p})"
  moreover have "τ ⊨ (p .oclAsType(NumericParameter).value_NumericParameter
                        ≜ (a .OEM_number_BrakeComponent->oclAsType_Int(Real)))"
  ultimately show
    "τ ⊨ (Namespace .allInstances()->select_Set(m|a ∼ m)
          ->asSequence_Set()->first_Seq().parameters
            ->select_Set(c|c .name_Parameter ≐ oem number)
          ->asSequence_Set()->first_Seq().oclAsType(NumericParameter)
      .value_NumericParameter ≜ ((a .OEM_number_BrakeComponent)
                                 ->oclAsType_Int(Real)))"qed
```

We do not additionally show our proof outline for `id_specs`$_{lemma}$ here, as it is structurally equivalent, and the two additional typecasts are not reflected in the structure of its proof sketch.

5.7 Metrics of Complexity for Model Consistency

We have reported on the formalization of a realistic metamodel, respective consistency constraints, and outlines of their proof attempt. Regarding the formalization effort, we had 5700 lines of Isabelle formalization plus 380 lines for the consistency definition, the constraints, and the proof outlines. For the car component example from Sect. 4, we had 670 lines for the fully abstract car metamodel, 770 for the refined car metamodel with `VINs`, and 1690 for the car component model that contains a message bus. For the consistency definition, constraints, and proof outlines, each of the car component examples comprises around 100 lines where the brake system model consists of 380 lines. Moreover, we have seen consistency specifications with varying complexity of navigation depth, namely 0 for the abstract car metamodel, 1 for the car model with navigation to the field `VINs`, and 2 for the message bus example, where we navigate to the `ECU` and then `prot`. For the brake system model, we navigate 1 step to the field `id` for the first constraint. For the second and third constraint, we navigate 4 steps to the association `parameters`, the field `name`, the respective subclass `NumericParameter`, and the field `value` for the second constraint. Depending on the granularity of our formalization, we could further count the additional typecast to Integer as a 5$^{\text{th}}$ step for the second constraint. The effects of the different navigation depths can be seen for the proofs from Sect. 4.3 to compare depth 0 with depth 1. Furthermore, the differences in depth of 1 and 4 in the proofs from Sect. 5.6 can be conjectured when considering that the proof of the first constraint is re-used in the proofs for the second and third constraint. The comparison of the total amount of Isabelle lines shown above also correlates with the difference in navigation depths.

6 Discussion

We explored the idea that 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. 3, 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 tedious, and existing translation approaches into the Rocq prover (Sheng et al. 2019) are not actively maintained. Automating this translation pipeline can reduce the encoding effort and also enable applying 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 (Aspinall and Kaliszyk 2016) 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.

7 Related Work

A fine-grained examination of complexity requires proper metrics. Object-oriented software design metrics have been explored (Ma et al. 2004; Lorenz and Kidd 1994; Purao and Vaishnavi 2003), while size-related modeling metrics have also been studied (Lange 2006). Object-oriented software measures (Archer and Stinson 1995) have also influenced the evaluation of formal proof complexity (Aspinall and Kaliszyk 2016). More broadly, the complexity of formal reasoning systems has been studied through the lens of proof complexity by Cook (1971); Cook and Reckhow (1979). Complementary work, such as that by Heijstek and Chaudron (2009), evaluates the complexity of distributed modeling processes by aggregating metrics per model type. Software complexity also links to cognitive weight (Shao and Wang 2003). Consistency in model-driven engineering is often related to model synchronization (Giese and Wagner 2006; Xiong et al. 2007; Giese et al. 2010). 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 (Bohannon et al. 2008). BX approaches provide formal specifications for consistency and repair between metamodel pairs (Stevens 2010).

Intra-model consistency and well-formedness can be expressed and checked using OCL. Early work by Chiorean et al. (2004) introduced OCL-based consistency specifications. Moreover, Bodeveix et al. (2002) proposed extensions for verifying UML model consistency. Mapping OCL constraints onto graph conditions enables automated verification using attributed typed graph rewriting (Bottoni et al. 2000). 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 (Giese and Wagner 2006; Anjorin et al. 2014). Similarly, comprehensive systems aim to provide integrated solutions for managing multiple interrelated heterogeneous models in a coherent manner (Stünkel et al. 2021; Golra et al. 2016). 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 (Atkinson et al. 2015). Projective methods derive (user-requested) views on demand from a centralized *Single Underlying Model (SUM)* (Atkinson et al. 2010), 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)* (Klare et al. 2021) blends both paradigms and presents itself as projective, but internally consists of overlapping and redundant models. The internal models must be actively kept consistent.

Managing inconsistencies (Dávid et al. 2017, 2016a,b; Vanherpen et al. 2016) is an important step towards successful collaborative engineering processes. These approaches also use constraints to detect inconsistencies, but typically do not rely on explicitly modeled correspondences between model elements. Instead, inconsistencies are identified directly at the level of global constraints spanning several models. Dávid et al. (2016b) introduce a metric called *bounded consistency*, which they use to describe scenarios where inconsistent states of the models eventually become consistent.

We considered consistency as a relation, where models are either consistent or not. In most frameworks, consistency is viewed as a syntactic property, but it can also rely on semantics (Pascual et al. 2025a). While this qualitative perspective enables formal repair and enforcement, temporary inconsistencies may be tolerated to avoid information loss (Finkelstein et al. 1994). Consistency can also be considered quantitatively, e.g., by counting constraint violations (Kegel et al. 2024; Kosiol et al. 2022), allowing graded analysis and change tracking. Such a quantitative consistency notion can be linked to the gradual notion proposed by Stevens (2014) who captures the degree of agreement between models by a value in a partially ordered set or lattice.

8 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.

In order to express consistency specification declaratively, we need a language other than the *Reactions* language. We aim to implement an OCL like language that can operate in a multi-model environment, much like Vitruvius and will be called Vitruv-OCL. We will implement all non-standard OCL language features that we presented in this paper, such as the coextension operator. This language will be used to express consistency without having an automated repair mechanism in mind and will not be used to repair inconsistent states of the system, at least not in the early stages.

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.

Statements and Declarations

Funding

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).

Competing interests

All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Author contributions

All authors contributed to the conceptualization and methodology developed in this manuscript. Porting the consistency specifications in the case study from its formalization in Vitruv to OCL was done by Lars König and Arne Lange. The formalization and analysis in Isabelle/HOL based on FeatherweightOCL was done by Michael Kirsten and Terru Stübinger. The first draft of the manuscript was written by Arne Lange, Romain Pascual, Thomas Weber and Michael Kirsten. All authors commented and updated the various versions of the manuscript. All authors read and approved the final manuscript.

Acknowledgments

This article is an extended version of a proceedings paper with the title "Towards Examining the Complexity of Consistency" that appeared at MODELS-C 2025 with DOI `10.1109/MODELS-C68889.2025.00091` (Pascual et al. 2025b). This paper has been edited by our textician Daniel Shea. We used generative artificial intelligence (AI) only for checking the spelling and grammar.

References

Ahrendt W, Beckert B, Bubel R, et al (eds) (2016) Deductive Software Verification - The KeY Book: From Theory to Practice, Lecture Notes in Computer Science, vol 10001. Springer International Publishing, https://doi.org/10.1007/978-3-319-49812-6

Ali T, Nauman M, Alam M (2007) An accessible formal specification of the UML and OCL meta-model in Isabelle/HOL. In: Zaidi J, Chughtai A (eds) 11th IEEE International Multitopic Conference (INMIC 2007). IEEE, https://doi.org/10.1109/INMIC.2007.4557693

Anjorin A, Rose S, Deckwerth F, et al (2014) Efficient model synchronization with view triple graph grammars. In: Cabot J, Rubin J (eds) 10th European Conference on Modelling Foundations and Applications (ECMFA@STAF 2014), Lecture Notes in Computer Science, vol 8569. Springer, pp 1–17, https://doi.org/10.1007/978-3-319-09195-2_1

Archer C, Stinson M (1995) Object-oriented software measures. Tech. rep., Defense Technical Information Center (DTIC), https://doi.org/10.21236/ADA294737

Aspinall D, Kaliszyk C (2016) Towards formal proof metrics. In: Stevens P, Wasowski A (eds) 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), Lecture Notes in Computer Science, vol 9633. Springer, pp 325–341, https://doi.org/10.1007/978-3-662-49665-7_19

Atkinson C, Kühne T (2008) Reducing accidental complexity in domain models. Software & Systems Modeling 7(3):345–359. https://doi.org/10.1007/s10270-007-0061-0

Atkinson C, Stoll D, Bostan P (2010) Orthographic software modeling: A practical approach to view-based development. In: Maciaszek L, González-Pérez C, Jablonski S (eds) Evaluation of Novel Approaches to Software Engineering. Springer, https://doi.org/10.1007/978-3-642-14819-4_15

Atkinson C, Tunjic C, Moller T (2015) Fundamental realization strategies for multi-view specification environments. In: 2015 IEEE 19th International Enterprise Distributed Object Computing Conference. IEEE Computer Society, pp 40–49, https://doi.org/10.1109/EDOC.2015.17

Beckert B, Keller U, Schmitt P (2002) Translating the object constraint language into first-order predicate logic. In: Autexier S, Mantel H (eds) The Second Verification Workshop: VERIFY affiliated with the 18th Conference on Automated Deduction (CADE) at FLoC'02, pp 02–07

Bodeveix J, Millan T, Percebois C, et al (2002) Extending OCL for verifying UML models consistency. Tech. Rep. 2002:06, Department of Software Engineering and Computer Science, Blekinge Institute of Technology

Bohannon A, Foster J, Pierce B, et al (2008) Boomerang: Resourceful lenses for string data. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Association for Computing Machinery, POPL '08, pp 407–419, https://doi.org/10.1145/1328438.1328487

Bottoni P, Koch M, Parisi-Presicce F, et al (2000) Consistency checking and visualization of OCL constraints. In: Evans A, Kent S, Selic B (eds) UML 2000 – Third International Conference on the Unified Modeling Language: Advancing the Standard, Lecture Notes in Computer Science, vol 1939. Springer, pp 294–308, https://doi.org/10.1007/3-540-40011-7_21

Bowman H, Steen M, Boiten E, et al (2002) A formal framework for viewpoint consistency. Formal Methods in System Design 21(2):111–166. https://doi.org/10.1023/A:1016000201864

Brooks F (1987) No silver bullet – Essence and accidents of software engineering. Computer 20(4):10–19. https://doi.org/10.1109/MC.1987.1663532

Brucker A, Doser J, Wolff B (2006) Semantic issues of OCL: Past, present, and future. Electronic Communications of the EASST 5. https://doi.org/10.14279/tuj.eceasst.5.46

Brucker A, Tuong F, Wolff B (2014) Featherweight OCL: a proposal for a machine-checked formal semantics for OCL 2.5. Archive of Formal Proofs https://isa-afp.org/entries/Featherweight_OCL.html, Formal proof development

Cederbladh J, Cicchetti A, Suryadevara J (2024) Early validation and verification of system behaviour in model-based systems engineering: A systematic literature review. ACM Transactions on Software Engineering Methodology 33(3):81:1–81:67. https://doi.org/10.1145/3631976

Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6):476–493. https://doi.org/10.1109/32.

295895

Chiorean D, Pasca M, Cârcu A, et al (2004) Ensuring UML models consistency using the OCL environment. Electronic Notes in Theoretical Computer Science 102:99–110. https://doi.org/10.1016/j.entcs.2003.09.005

Clarke E, Henzinger T, Veith H, et al (eds) (2018) Handbook of Model Checking. Springer, https://doi.org/10.1007/978-3-319-10575-8

Clavel M, Egea M, de Dios M (2009) Checking unsatisfiability for OCL constraints. Electronic Communications of the EASST 24. https://doi.org/10.14279/tuj.eceasst.24.334

Cook S (1971) The complexity of theorem-proving procedures. In: Harrison M, Banerji R, Ullman J (eds) Proceedings of the Third Annual ACM Symposium on Theory of Computing. Association for Computing Machinery, STOC '71, pp 151–158, https://doi.org/10.1145/800157.805047

Cook S, Reckhow R (1979) The relative efficiency of propositional proof systems. The Journal of Symbolic Logic 44(1):36–50. https://doi.org/10.2307/2273702

Cousot P, Cousot R (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham R, Harrison M, Sethi R (eds) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages. Association for Computing Machinery, pp 238–252, https://doi.org/10.1145/512950.512973

Czarnecki K, Helsen S (2006) Feature-based survey of model transformation approaches. IBM Systems Journal 45(3):621–645. https://doi.org/10.1147/sj.453.0621

Dávid I, Denil J, Gadeyne K, et al (2016a) Engineering process transformation to manage (in) consistency. In: Muccini H, Malavolta I, Gerard S, et al (eds) 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016) co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), pp 7–16

Dávid I, Syriani E, Verbrugge C, et al (2016b) Towards inconsistency tolerance by quantification of semantic inconsistencies. In: Muccini H, Malavolta I, Gerard S, et al (eds) 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016) co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), pp 35–44

Dávid I, Meyers B, Vanherpen K, et al (2017) Modeling and enactment support for early detection of inconsistencies in engineering processes. In: Burgueño L (ed) MoDELS Satellite Events, pp 145–154

David I, Vangheluwe H, Syriani E (2023) Model consistency as a heuristic for eventual correctness. Journal of Computer Languages 76:101223. https://doi.org/10.1016/J.COLA.2023.101223

Demuth B, Wilke C (2009) Model and object verification by using Dresden OCL. In: The 2nd Russian-German Workshop “Innovation Information Technologies: Theory and Practice”, pp 687–690

Finkelstein A (2000) A foolish consistency: Technical challenges in consistency management. In: Database and Expert Systems Applications. Springer

Finkelstein A, Gabbay D, Hunter A, et al (1994) Inconsistency handling in multiperspective specifications. IEEE Transactions on Software Engineering 20(8):569–578. https://doi.org/10.1109/32.310667

Franconi E, Mosca A, Oriol X, et al (2019) $\mathrm{Ocl}_{\mathrm{fo}}$: first-order expressive OCL constraints for efficient integrity checking. Software and Systems Modeling 18(4):2655–2678. https://doi.org/10.1007/S10270-018-0688-Z

Giese H, Wagner R (2006) Incremental model synchronization with triple graph grammars. In: Nierstrasz O, Whittle J, Harel D, et al (eds) 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), Lecture Notes in Computer Science, vol 4199. Springer, pp 543–557, https://doi.org/10.1007/11880240_38

Giese H, Hildebrandt S, Neumann S (2010) Model synchronization at work: Keeping SysML and AUTOSAR models consistent. In: Engels G, Lewerentz C, Schäfer W, et al (eds) Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday. Springer, p 555–579, https://doi.org/10.1007/978-3-642-17322-6_24

Golra F, Beugnard A, Dagnat F, et al (2016) Addressing modularity for heterogeneous multi-model systems using model federation. In: Fuentes L, Batory D, Czarnecki K (eds) Companion Proceedings of the 15th International Conference on Modularity (Modularity ’16). Association for Computing Machinery, pp 206–211, https://doi.org/10.1145/2892664.2892701

Hagel N, Mäkelburg J, Hammann C, et al (2025) Explainability in automated cross-domain model-driven brake system development. In: Proceedings of the 1st International Workshop on Explainable Automated Software Engineering @ASE2025, pre-print

Halstead MH (1977) Elements of Software Science. Elsevier, New York

Heijstek W, Chaudron MR (2009) 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, pp

113–120, https://doi.org/10.1109/SEAA.2009.70

Kegel K, Götz S, Marx R, et al (2024) A variance-based drift metric for inconsistency estimation in model variant sets. In: 20th European Conference on Modelling Foundations and Applications. JOT

Klare H, Kramer M, Langhammer M, et al (2021) Enabling consistency in view-based system development — the Vitruvius approach. Journal of Systems and Software 171(110815). https://doi.org/10.1016/j.jss.2020.110815

Kosiol J, Strüber D, Taentzer G, et al (2022) Sustaining and improving graduated graph consistency: A static analysis of graph transformations. Science of Computer Programming 214:102729

Lange C (2006) Model size matters. In: Kühne T (ed) International Conference on Models in Software Engineering, Workshops and Symposia at MoDELS, Lecture Notes in Computer Science, vol 4364. Springer, pp 211–216, https://doi.org/10.1007/978-3-540-69489-2_26

Lorenz M, Kidd J (1994) Object-oriented software metrics - a practical guide. Prentice Hall, Inc.

Lucas F, Molina F, Toval A (2009) A systematic review of UML model consistency management. Information and Software Technology 51(12):1631–1645. https://doi.org/10.1016/j.infsof.2009.04.009

Ma H, Shao W, Zhang L, et al (2004) Applying OO metrics to assess UML meta-models. In: 7th International Conference on the Unified Modelling Language: Modelling Languages and Applications (UML 2004), Springer, pp 12–26, https://doi.org/10.1007/978-3-540-30187-5_2

McCabe T (1976) A complexity measure. IEEE Transactions on Software Engineering SE-2(4):308–320. https://doi.org/10.1109/TSE.1976.233837

Meyer B (1992) Applying 'design by contract'. Computer 25(10):40–51. https://doi.org/10.1109/2.161279

Nipkow T, Paulson L, Wenzel M (2002) Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol 2283. Springer, https://doi.org/10.1007/3-540-45949-9

Object Management Group, Inc. (OMG) (2014) Object constraint language (OCL). Tech. rep., Object Management Group, Inc. (OMG), URL https://www.omg.org/spec/OCL/2.4

Pascual R, Beckert B, Ulbrich M, et al (2025a) Formal foundations of consistency in model-driven development. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2024). Specification and Verification. Springer Nature Switzerland, pp 178–200, https://doi.org/10.1007/978-3-031-75380-0_11

Pascual R, Lange A, Weber T, et al (2025b) Towards examining the complexity of consistency. In: Kessentini M, Ali S, Sahraoui H (eds) ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C 2025). IEEE Computer Society, pp 663–672, https://doi.org/10.1109/MODELS-C68889.2025.00091

Paulson L (1989) The foundation of a generic theorem prover. Journal of Automated Reasoning 5(3):363–397. https://doi.org/10.1007/BF00248324

Purao S, Vaishnavi V (2003) Product metrics for object-oriented systems. ACM Computing Surveys (CSUR) 35(2):191–221. https://doi.org/10.1145/857076.857090

Rajic G, Sruk V (2024) Definitions and computational properties of OCL: A systematic review. IEEE Access 12:99704–99738. https://doi.org/10.1109/ACCESS.2024.3428865

Shao J, Wang Y (2003) A new measure of software complexity based on cognitive weights. Canadian Journal of Electrical and Computer Engineering 28(2):69–74. https://doi.org/10.1109/CJECE.2003.1532511

Sheng F, Zhu H, Yang Z (2019) Towards the mechanized semantics and refinement of UML class diagrams. In: 26th Asia-Pacific Software Engineering Conference (APSEC 2019). IEEE, pp 47–54, https://doi.org/10.1109/APSEC48747.2019.00016

Stachowiak H (1973) Allgemeine Modelltheorie. Springer, Wien; New York

Steimann F, Clarisó R, Gogolla M (2023) OCL rebuilt, from the ground up. In: 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023. IEEE, pp 194–205, https://doi.org/10.1109/MODELS58315.2023.00010

Stevens P (2010) Bidirectional model transformations in QVT: Semantic issues and open questions. Software & Systems Modeling 9(1):7–20. https://doi.org/10.1007/s10270-008-0109-9

Stevens P (2014) Bidirectionally tolerating inconsistency: Partial transformations. In: Gnesi S, Rensink A (eds) Fundamental Approaches to Software Engineering. Springer, pp 32–46

Stevens P (2020) Maintaining consistency in networks of models: bidirectional transformations in the large. Software and Systems Modeling 19:39–65. https://doi.org/10.1007/s10270-019-00736-x

Stünkel P, König H, Lamo Y, et al (2021) Comprehensive systems: A formal foundation for multi-model consistency management. Formal Aspects of Computing 33(6):1067–1114. https://doi.org/10.1007/s00165-021-00555-2

Vanherpen K, Denil J, David I, et al (2016) Ontological reasoning for consistency in the design of cyber-physical systems. In: 1st International Workshop on Cyber-Physical Production Systems (CPPS). IEEE, pp 1–8, https://doi.org/10.1109/CPPS.2016.7483922

Xiong Y, Liu D, Hu Z, et al (2007) Towards automatic model synchronization from model transformations. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering. Association for Computing Machinery, ASE '07, pp 164–173, https://doi.org/10.1145/1321631.1321657

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- supplementarymaterial.zip