MULTI Vector Industry Challenge

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

Clemens Reichmann
clemens.reichmann@vector.com
Vector Informatik GmbH
Karlsruhe, Germany

Thomas Kühne
thomas.kuehne@ecs.vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Pierre Maier
pierre.maier@uni-due.de
University of Duisburg-Essen
Essen, Germany

Thomas Weber
thomas.weber@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

The "MULTI" workshop series has set a number of multi-level modeling challenges, each designed to allow competing multi-level modeling approaches to demonstrate their capabilities and/or to tease out their limitations. The challenges, therefore, have been serving a three-fold purpose: First, they have allowed technologies to demonstrate their abilities. Second, they have pointed out where technologies still fall short of providing optimal modeling support. Third, they have provided a basis for comparing competing technologies, often revealing the trade-offs implied by certain design choices. The MULTI Vector Industry Challenge described in this paper is the fifth installment in this series. It is unique among the existing challenges in that it has been developed in collaboration with the industry partner Vector[1] and is directly inspired by a respective real world modeling challenge.

CCS Concepts

• **Software and its engineering → Software design engineering**; • **Computing methodologies → Modeling methodologies**.

Keywords

Multi-level modeling, challenge

ACM Reference Format:

Arne Lange, Clemens Reichmann, Thomas Kühne, Pierre Maier, and Thomas Weber. 2026. MULTI Vector Industry Challenge. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '26), October 4–9, 2026, Malaga, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/TBD

1 Introduction

Modern vehicles feature complex electrical systems, often involving many electronic control units (ECUs) interconnected via wiring harnesses comprising cables and connectors. Managing the design and engineering of these electrical systems is a significant challenge, as a single vehicle may require thousands of electrical connections.

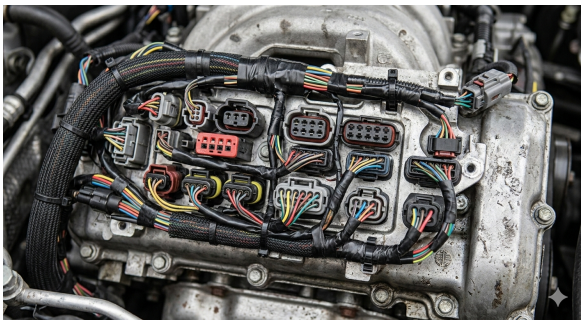

Figure 1: Wiring Connectors[2]

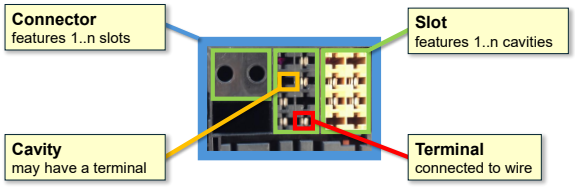

Figure 2: Terminology

A connector that is part of a wiring harness typically comprises a hierarchical assembly of components (see Fig. 2), i.e., a connector housing contains one or more slots and each slot in turn aggregates one or more cavities, which themselves contain wire terminals. When two connectors are mated – for example, a plug (female socket) and a header (male pin housing) – their terminals must match precisely, both logically and physically, to ensure correct electrical connectivity.

In model-based systems engineering, the aforementioned physical structures are often represented at multiple levels of abstraction. The "RFLP" approach captures these levels by featuring so-called Requirements, Functional, Logical, and Physical *domains* [3]. With respect to our example, the functional domain specifies what a connector must achieve in terms of signal routing; the logical domain defines how cavities and terminals relate to one another; while the physical domain captures the concrete geometry and mating constraints of the hardware.

[1] https://www.vector.com/int/en/

[2] generated with Gemini Nano Banana 2

MODELS Companion '26, Malaga, Spain

ACM ISBN TBD
https://doi.org/TBD

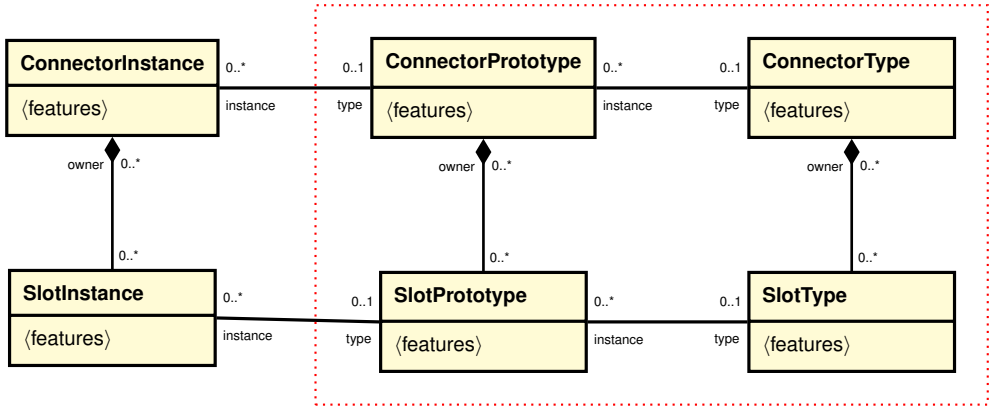

Figure 3: Simplified Metamodel, featuring two "instance Square" applications

2 Challenge description

The Vector company offers a model-based development tool named "PREEvision" and this challenge is about investigating whether the use of multi-level modeling concepts may be advantageous when representing the concepts underpinning the tool. In the following, we present a narrowly scoped and simplified conceptualization but retain Vector's internally used terminology. The latter should not be interpreted to reference specific definitions used in certain technical modeling contexts; challenge solutions may choose alternative terms, as long as the same functionality is supported.

While standards such as Vehicle Electrical Container (VEC) only recognize connector specifications and connectors, Vector models three concepts (cf. Figure 3):

A **ConnectorType** represents the specification of a connector family (e.g., single slot, 3-cavity housing).

A **ConnectorPrototype** represents a specific usage of a connector type within a particular (sub-)system context.

A **ConnectorInstance** represents a physical connector in a specific installation context.

Vector uses a so-called *"structured typeOf"* (STO) relationship between connector types and connector prototypes, and connector prototypes and connector instances respectively (see the associations with "instance" and "type" roles in Figure 3). What makes these STO relationships "structured" is that such connected concepts may have parts (cf. the composition relationships in Figure 3) which then must conform to the parts of their types (see the following section for details). Respective composition hierarchies can be arbitrarily deep; Figure 4 shows three levels (connectors, slots, and cavities). The challenge consists of supporting the above described modeling approach, while enforcing certain integrity constraints.

2.1 Structural Foundation

Vector refers to the model shown in Figure 4, in accordance with common parlance, as a "metamodel" since the model plays the role of a linguistic type model [4]. In other words, the model is conceptually located at a so-called "M_2" level [1] and serves as a grammar for models to be created at the "M_1" level below, which then actually represent the domain.

Figure 3 shows a version of the "metamodel" in Figure 4 that has been reduced to its essentials to support focusing on the main relationships. The "⟨features⟩" section in each class is a placeholder for a collection of features that are not elaborated for simplicity. Figure 4 is provided as a "metamodel" extract that uses structural principles elaborated below. Type names using variant indicators "A", "B", or "C", are placeholders for typically rather cryptic part concept identifiers. Note that the "*Instance" types on the left-hand side of Figure 4 all represent some kind of connection type. An example for a "PluginLocationInstance" would be an object that represents a concrete relay socket, e.g. as part of an ECU.

An instance of ConnectorType can be linked to an arbitrary number of ConnectorPrototype instances, meaning such an instance plays the role of the type for the respective ConnectorPrototype instances. This design follows the "Type Object" pattern [2], which uses links between objects – here between objects of type ConnectorType and objects of type ConnectorPrototype – to emulate classification relationships between an entity playing the role of a type and entities playing the role of its instances. This design is repeated for the pair ConnectorPrototype and ConnectorInstance.

In Figure 3, ConnectorInstance maintains a composition relationship to SlotInstance, since slots are considered to be essential parts of connectors (see Sec. 1). In accordance with the principle of modeling a system at different levels of abstraction (see Sec. 2), the same composition relationship is also modeled at the prototype and type levels. Note that the owned elements (cf. the bottom elements in Figure 3) form two applications of the "Type Object" pattern themselves. In combination with the composition relationships, the red rectangle in Figure 3 therefore marks an application of the "Type-Square" pattern [5, 6], since the slot concepts can be regarded as "properties" of the connector concepts. The same Type-Square structure is repeated between ConnectorPrototype and ConnectorInstance and the respectively associated slot concepts.

The repeated Type-Square structure shown in Figure 3 may be extended indefinitely to the right, introducing more whole-part pairs, and can also be extended vertically downwards, i.e., to increase composition depth. In the chosen example domain, the composition depth continues beyond what is shown in Figs. 3 & 4 and includes terminal concepts below cavity concepts.

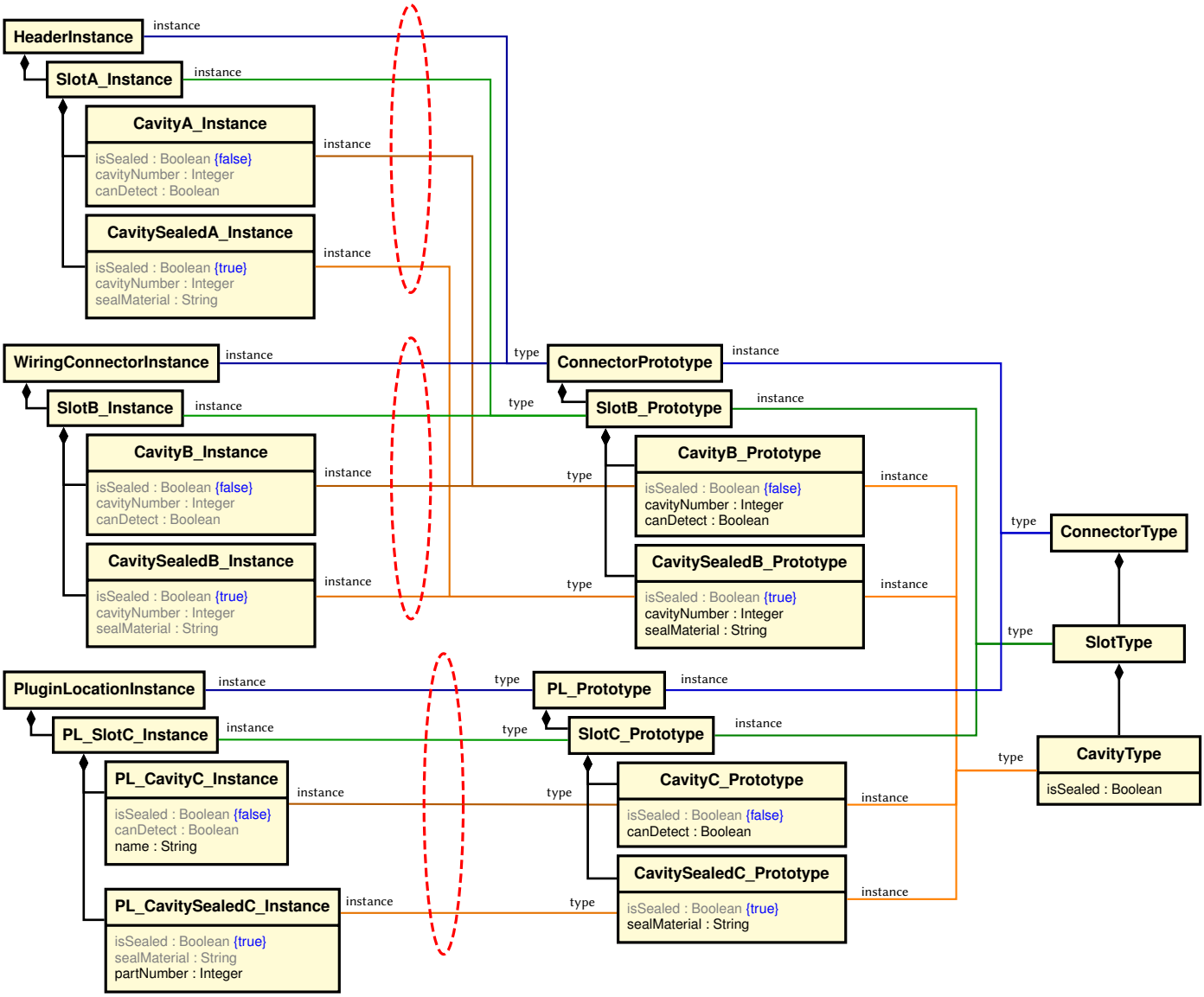

Figure 4: Metamodel

2.1.1 Feature projection. Attributes shown in gray in Figure 4 define so-called "projected features". For example, the "isSealed" value of a CavityType instance, is projected to instances of CavityB_Prototype and, further, to instances of CavityB_Instance. The two mentioned types define the "isSealed" attribute to equip their instances with the respective property, but the attributes are shown in gray because a typical characteristic of such properties within STO chains is that all values are constrained to be identical to the one introduced at the highest level. Note that for "canDetect", the highest level is the prototype level, rather than the type level as is the case for "isSealed". Solution submitters are encouraged to additionally allow refinement of values along STO chains, but value projection as described above should always be possible and remain effective after any optional refinement to a value has been made.

Particular feature values, e.g., "isSealed = true", can dictate which choice of M_2-level types (here, e.g. CavitySealedB_Prototype) are admissible for instances in the respective STO chain and which are not (here, e.g. CavityB_Prototype). This impact of feature values on M_2-level type selection is denoted by the use of blue "{true}"/"{false}" constraints in Figure 4.

For the sake of a concise presentation, in the following, we focus on the substructure within the red frame of Figure 3, i.e., we ignore both the horizontal extension of the frame (ConnectorInstance & SlotInstance) as well as the vertical continuation of the composition hierarchy (cavity & terminal concepts). Note, however, that the integrity constraints described below do apply analogously for all elements shown in Figure 4.

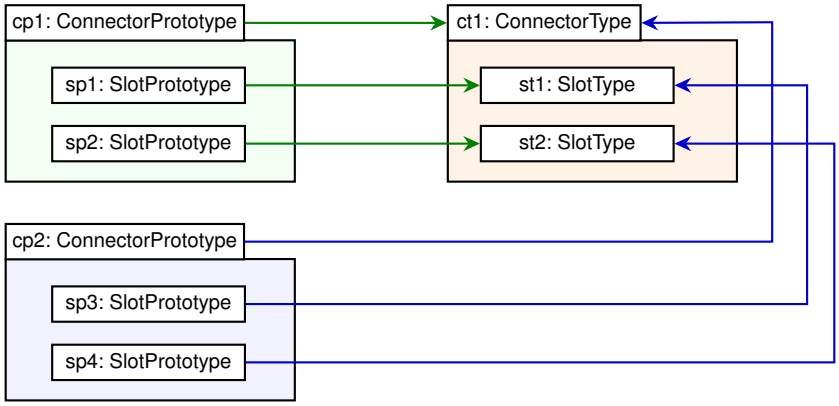

Figure 5: Congruent owner and parts typing

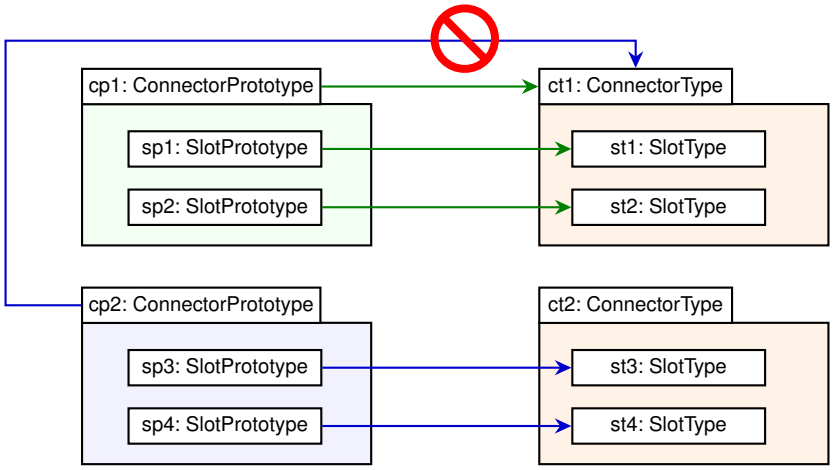

Figure 6: Incongruent typing relationship

2.2 Integrity constraints

Scenarios conforming to the structure shown in Figures 3 & 4 must satisfy two integrity constraints that ensure the integrity of classification links:

1. typing relationships must be congruent with respect to the composition structure.
2. part-typing relationships must form a bijection in the context of their owning elements.

Figure 4 selectively shows three examples of sets of relationships that are constrained in this manner as circled in red. Further sets are not circled to reduce diagram complexity.

2.2.1 Typing relationship congruence. In Figure 5, we use a visual nesting approach to obviate the need to draw composition relationships between owners (e.g., ct1) and their parts (here, st1 & st2). We furthermore omit "type" and "instance" role names; instead we use arrowheads to denote "type" role ends. The top part of Figure 5 illustrates a well-formed scenario in which the typing relationships between the parts (here, between sp1 & st1 and sp2 & st2 respectively) are congruent with the typing relationship of their owners (cp1 & ct1), i.e., the typing relationships between the parts conform to the context defined by the owners in the sense that they do not cross between ownership contexts.

The bottom part of Figure 5 reinforces that the congruence context is defined by the owning elements (here cp2 & ct1); in particular, there is no crossing over of relationships between entities. N.B., the different colors of the typing relationships do not carry any semantics, we merely use them to differentiate between owner contexts.

In contrast, Figure 6 shows an invalid crossing of the typing relationship between cp2 and ct1. Of course, it could also be argued that this crossing relationship is the intended one and the two blue typing relationships at the bottom of Figure 6 are the offending elements, and should be connecting to st1 and st2, respectively. However, unless one ascribes higher priority to the relationship between the owners, there is no way to assert blame; one may only detect an inconsistent (as in "incongruent ownership") structure.

In Figure 7, the typing relationships between the owners are assumed to be the intended ones and therefore the two crossing (incongruent) typing relationships between parts are considered the offending ones.

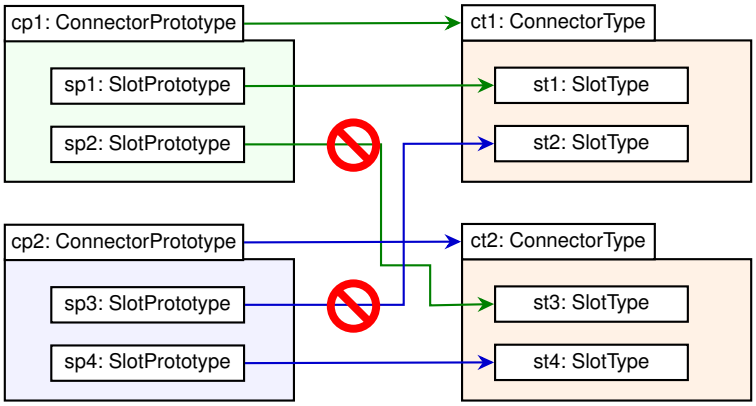

Figure 7: Incongruent part typing relationship

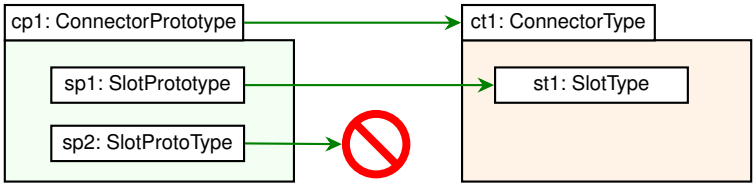

Figure 8: Invalid lack of a part type

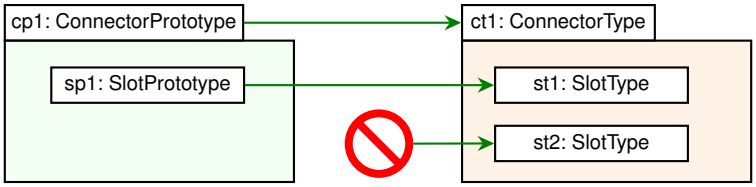

Figure 9: Invalid lack of a part instance

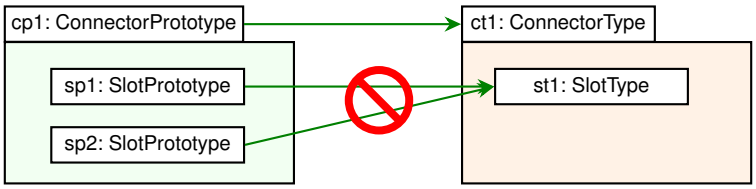

Figure 10: Invalid sharing of a part type

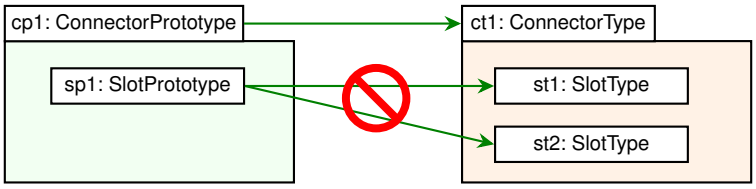

Figure 11: Invalid sharing of a part instance

2.2.2 Bijection requirement. Parts must maintain a one-to-one mapping with parts they are typed by and parts they are types for. Formally, the relation formed by all typing relationships between parts (within an owner context) on different adjacent abstraction levels must be a bijection, i.e., the relation must be both injective and surjective. In other words, within a single owner context, all parts in an instance role must have exactly one type, and all parts in a typing role must have exactly one instance. Figures 8–11 depict cases in which the typing relation between the parts of owning elements is not a bijection.

In case the parts of a single owner may have different M_2-types (cf. Figure 4), the bijection requirement applies to relationships between corresponding M_2-types (e.g., CavityB_Instance and CavityB_Prototype), i.e., it would not be sufficient to just have a global owner-context bijection that is agnostic of M_2-level part types.

Note that in Figures 6 & 7 the bijection requirements are satisfied, but the typing relationships are not confined by the ownership context, i.e., violate the typing relationship congruence integrity condition.

2.3 Change management

In addition to enforcing the aforementioned integrity conditions, a solution must also manage changes to owner/part structures. For instance, if an M_1-level modeler attempts to remove a part (say, a slot prototype), this must either be prohibited or the consequences must be managed accordingly. Any affected elements must be adjusted accordingly, so that the overall structure remains consistent, i.e., satisfies the integrity conditions. Changes are expected to propagate bi-directionally, i.e., there is no predominance of types over their instances, for example.

Likewise, value changes to properties like "isSealed" must be propagated to projected features along STO chains, and any potentially repercussions with respect to M_2-level type selections must be flagged (cf. Section 2.1.1).

3 Solution submission requirements

Solutions must feature:

1. a very brief explanation of the technology used (e.g., language, tool, properties, etc.).
2. a brief solution description, including a diagram and a list of salient properties.
3. any significant well-formedness constraint(s).
4. a trade-off analysis.
5. assumptions made, if applicable.

The diagram may focus on essential parts, i.e., it is not necessary to include all concepts shown in Figure 4.

3.1 Mandatory aspects to cover

a) Which fundamental conceptual relationships (e.g., classification, concretization, refinement) support the solution?
b) What are the main well-formedness principles underlying the solution (e.g., strict vs. non-strict)?
c) Which language constructs or modeling patterns are vital to the solution?
d) Which integrity conditions (well-formedness constraints) are evaluated/enforced and when?

References

[1] Colin Atkinson and Thomas Kühne. 2005. Concepts for Comparing Modeling Tool Architectures. In *Proceedings of the ACM/IEEE 8th MODELS conference*. 398–413.
[2] Ralph Johnson and Bobby Woolf. 1997. Type Object. In *Pattern Languages of Program Design 3*, Robert C. Martin, Dirk Riehle, and Frank Buschmann (Eds.). Addison-Wesley, USA, 47–65.
[3] Sven Kleiner and Christoph Kramer. 2013. Model Based Design with Systems Engineering Based on RFLP Using V6. In *Smart Product Engineering*, Michael Abramovici and Rainer Stark (Eds.). Springer Berlin Heidelberg, 93–102.
[4] Thomas Kühne. 2006. Matters of (Meta-) Modeling. *Software and System Modeling* 5, 4 (2006), 369–385. doi:10.1007/s10270-006-0017-9
[5] Joseph W Yoder, Federico Balaguer, and Ralph Johnson. 2001. Architecture and design of adaptive object-models. *ACM Sigplan Notices* 36, 12 (2001), 50–60.
[6] Joseph W. Yoder and Ralph E. Johnson. 2002. The Adaptive Object-Model Architectural Style. In *Proceedings of the 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*. Kluwer, NLD, 3–27.