Software and Systems Modeling (2026) 25:635–637
https://doi.org/10.1007/s10270-026-01384-8

EDITORIAL

How UML models and ontologies can complement each other

Marsha Chechik[1] · Benoit Combemale[2] · Jeff Gray[3] · Bernhard Rumpe[4]

Published online: 7 May 2026

A common understanding of ontologies is that an ontology is a structured framework used to organize information and knowledge in a specific domain. It defines concepts, categories, properties, and relationships in a way that both humans and computers can understand. Typical ontological components include concepts (or classes), instances (or individuals), properties describing characteristics, and relationships between concepts as well as between instances (e.g., a "Person" has a "birthdate" and “Konrad Zuse” is an instance of "Person" and has his birthdate on “June, 22”).

Practical applications demonstrate that ontologies can be successfully used in a variety of domains, such as artificial intelligence, the semantic web, enterprise architecture, or information retrieval, often in the form of knowledge graphs. This makes them particularly attractive when the goal is to capture, share, and reason about domain knowledge.

When comparing the language constructs, the ontology approach is very closely related both to UML class diagrams and to MOF-based metamodels. However, in both cases expressibility, but in particular their purposes differ to a large extent. It is therefore worthwhile to compare both individually.

✉ Bernhard Rumpe
bernhard.rumpe@sosym.org

Marsha Chechik
marsha.chechik@sosym.org

Benoit Combemale
benoit.combemale@sosym.org

Jeff Gray
jeff.gray@sosym.org

[1] University of Toronto, Toronto, ON, Canada

[2] University of Rennes, Rennes, France

[3] University of Alabama, Tuscaloosa, AL, USA

[4] RWTH Aachen University, Aachen, Germany

1 Ontologies vs. MOF-based metamodels: domain modeling

When looking at domain modeling from a conceptual perspective, a comparison between ontologies and MOF-based metamodels may be appropriate. Both ontologies and metamodels aim to describe a domain, but they again do so with fundamentally different objectives.

Ontologies are primarily designed to capture domain knowledge and to some extent also to support reasoning (even though not that intensively used in practice). They allow the derivation of implicit knowledge, the detection of inconsistencies, and the alignment of heterogeneous data sources. Their semantics follows an open-world assumption, where incomplete knowledge is acceptable.

MOF-based metamodels, in contrast, define the abstract syntax of modeling languages. They specify the concepts, relationships, and constraints that can be used to construct models. Their purpose is not only to describe a domain, but to enable the definition of domain-specific modeling languages (DSMLs) and to support model creation, validation, and transformation. As such, they are inherently constructive, complete, and operational.

Furthermore, a MOF-metamodel usually comes acquainted with a concrete syntax completing the modeling language definition, while ontologies remain "abstract" in that they act toward database structures.

This differences lead to distinct characteristics. Metamodels rely on strong typing, explicit constraints such as cardinalities, and typically a closed-world assumption to ensure that models are well-formed and can be processed reliably by tools. Ontologies, on the other hand, emphasize flexibility and extensibility, allowing knowledge to evolve.

From a pragmatic point of view, these approaches are complementary. Ontologies can serve as a semantic foundation that captures shared domain knowledge, while metamodels operationalize this knowledge within concrete and stable modeling environments. A systematic mapping between ontologies and metamodels—where concepts become metaclasses, and relationships become references—could enable

both reuse of domain knowledge and integration with model-driven engineering toolchains.

2 Ontologies vs. UML: software development perspective

A UML class diagram is at the beginning of a software development, i.e., in the requirements phases, used for capturing domain knowledge for that specific project. But later on, i.e., in design, implementation, and testing, it is used to describe software structures. Between requirements elicitation and architectural design, a class diagram takes a strong semantical shift (without changing the syntax!).

It is thus worthwhile to compare UML class diagrams for requirements capture and ontologies for domain knowledge capture the context of software development. This comparison remains relevant, as both are used to model domains at a level closer to implementation.

The ontology approach is indeed closely related to UML class diagrams when comparing language constructs and purposes. Both describe concepts, properties, and relationships within a domain. However, there are notable differences.

Most obviously, UML class diagrams do not directly describe instances. To some extent, the "<<singleton>>" stereotype partially mimics this, but a better approach is to use UML object diagrams for instance-level modeling. Here each object also has its class used as type. This is a nice separation, using one kind of diagram for one purpose, where ontologies use a one-fits-all-approach and integrate both the conceptual and instance levels within a single framework. On the other hand, UML class diagrams provide a much richer language for describing structures: for example, their associations describe multiplicities and navigability, subclassing has impact on the typing of instances. UML class diagrams have enumerations, interfaces, abstract classes, derived properties, and more.

Another strong difference lies in the role of constraints. UML is primarily designed for software engineering and focuses on specifying the set of allowed data structures. The above-mentioned types, attributes, associations, and cardinalities are used to precisely define and restrict possible data configurations. This is essential for developing robust software systems that must behave correctly for all possible inputs and data given. Thus, developers have to include all forms of corner cases, because they might occur in the unforeseeable forms of software uses. A strong, statically checkable type system defined by the classes are therefore a key asset.

Ontologies, on the other hand, are often used in data analytics contexts, where systems operate on existing and largely immutable datasets. In such scenarios, unusual or unforeseen relationships are less critical, and the emphasis is on capturing and exploring knowledge rather than enforcing strict constraints.

Interestingly, while UML class diagrams and ontologies are both widely used, UML object diagrams are rarely applied in industrial practice. One possible reason is that tool support has historically neglected them. Strengthening their role could allow developers to define concrete data sets for initialization, testing, or data exchange—bringing UML closer to some practical uses currently occupied by ontologies.

It is often argued that a major advantage of ontologies is their ability to automatically derive relationships between instances through reasoning. However, this capability is less frequently exploited in industrial settings, where ontologies are often used primarily to define structural relationships, sometimes reduced to simple ownership hierarchies—similar to a bill of materials. In such cases, their expressive power is not fully utilized and may even fall short of what UML class diagrams can provide in terms of precise structural specification.

3 Toward an integration of both worlds

Considering both perspectives, the purposes of ontologies and model-based software engineering approaches start to overlap. Ontologies aim to capture domain knowledge and support reasoning, while metamodels and UML models aim to enable the design and implementation of software systems.

Several integration scenarios can be envisioned. One possible direction is to use ontologies as a domain-semantic foundation for metamodels. In this approach, an ontology captures the domain knowledge, including concepts and their relationships, while a corresponding metamodel defines how this knowledge is operationalized within a modeling language. Then, here we assume that the domain ontology is "known" knowledge and the metamodel can be semantically grounded in that knowledge. The metamodel introduces the necessary constraints and structures required for model creation and manipulation, while a corresponding ontology tool provides reasoning, validation, and interoperability.

A second option is a much tighter integration, where ontology concepts are systematically mapped to metamodel elements. Concepts become metaclasses, properties become attributes or references, and ontology constraints are translated into metamodel constraints where possible. At the same time, reasoning capabilities from the ontology world are mapped to metamodel-based validation by detecting inconsistencies or inferring additional model elements. If the metamodel then also is equipped with a concrete syntax, then a concrete domain-specific language with ontological grounding, reasoning capabilities, and most importantly consistency checking is defined. A variant of this option would

be to use an ontology as an early intermediary step when designing a new DSL.

A third possibility finally would be to integrate ontology and UML class diagram (as well as partly to an object diagram). Here, an ontology serves as a library model, capturing domain constructs and relationships (i.e., classes, associations, and attributes) like it does now. Those are mapped to UML class diagram elements, thus serving as a good foundation for the requirement phase of a software development project. Additional UML class diagrams, e.g., capturing technical classes, GUI classes, communication, security, etc., can be defined and merged and thus used in the downstream software development process. Here, the ontology does not describe a metamodel and thus a new modeling language, but a concrete class model for a concrete software project, which makes ontologies a perfect library for reusable class diagram models.

Such integrations also open perspectives for connecting behavioral models from the UML with domain knowledge. For example, state machines, activity models, or process models could operate on instances defined by an ontology, enabling richer semantic validation and more expressive model transformations.

Ultimately, combining ontologies and MOF-based metamodels or UML modeling techniques allows leveraging the strengths of all three worlds: the reasoning capabilities of ontologies, and the constructive, tool-supported nature of metamodeling and model-driven engineering. This synergy could significantly enhance model-based software development by improving reuse, interoperability, and the alignment between domain knowledge and system design.

Wouldn't that significantly advance model-based software development toward a more knowledge-driven discipline?

For a deeper discussion and more ideas on this, we may also refer to Brian Henderson-Sellers paper on "Bridging metamodels and ontologies in software engineering".

Funding Open Access funding enabled and organized by Projekt DEAL.