PART I

FUNDAMENTALS

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

Among recent attempts to improve productivity in software engineering, model-driven engineering (MDE) is an approach that focuses on the design of artifacts and on generative techniques to raise the level of abstraction of physical systems [142]. As model-driven engineering gains momentum, the transformation of artifacts and domain-specific notations become essential in the software development process.

One of the pre-existing modeling languages that boosted research on MDE is the Unified Modeling Language (UML). UML is a visual design notation [117] for designing software systems. It is a general-purpose modeling language, capable of capturing information about different views of systems, like static structure and dynamic behavior.

In addition to general-purpose modeling languages, MDE relies on domain-specific languages (DSL). Such languages provide abstractions and notations for modeling specific aspects of systems. A variety of domain-specific languages and fragments of their models is used to develop one large software system.

Among artifacts produced by multiple modeling languages, MDE faces the following challenges [57]: support for developers, interoperability among multiple artifacts, and formal semantics of modeling languages. Addressing these challenges is crucial to the success of MDE.

In contrast, issues like interoperability and formal semantics motivate the development of ontology web languages. Indeed, the World Wide Web Consortium (W3C) standard Web Ontology Language (OWL) [61], together with automated reasoning services, provides a powerful solution for formally describing domain concepts in an extensible way, thus allowing for precise specification of the semantics of concepts as well as for interoperability between ontology specifications.

Ontologies provide shared domain conceptualizations representing knowledge by a vocabulary and, typically, logical definitions [62, 161]. OWL provides a class definition language for ontologies. More specifically, OWL allows for the definition of classes by required and implied logical constraints on the properties of their members.

The strength of OWL modeling lies in disentangling conceptual hierarchies with an abundance of relationships of multiple generalization of classes (cf. [128]). For this purpose, OWL allows for *deriving* concept hierarchies from logically and precisely defined class axioms stating necessary and sufficient conditions of class membership. The logics of class definitions may be validated by using corresponding automated reasoning technology.

Ontology engineers usually have to cope with W3C standard specifications and programming languages for manipulating ontologies. The gap between W3C specifications and programming language leads ontology engineers to deal with multiple languages of different natures. For instance, W3C specifications are platform independent, whereas programming languages include platform-specific constructs.

Indeed, addressing these issues has been one of the objectives of model-driven engineering. MDE allows for developing and managing abstractions of the solution domain towards the problem domain in software design, turning the focus from code-centric to transformation-centric.

Understanding the role of ontology technologies like knowledge representation, automated reasoning, dynamic classification, and consistency checking in MDE as well as the role of MDE technologies like model transformation and domain-specific modeling in ontology engineering is essential for leveraging the development of both paradigms.

For example, UML and OWL constitute modeling approaches with strengths and weaknesses that make them appropriate for specifying distinct aspects of software systems. UML provides means to express dynamic behavior, whereas OWL does not. OWL is capable of inferring generalization and specialization between classes as well as class membership of objects based on the constraints imposed on the properties of class definitions, whereas UML class diagrams do not allow for dynamic specialization/generalization of classes and class memberships or any other kind of inference *per se*.

Though schemas [111] and UML extensions (UML profiles) for OWL ontologies exist, an integrated usage of both modeling approaches in a coherent framework has been lacking so far. This book unveils research problems involving the composition of these two paradigms and presents research methods to assess the application of a novel framework integrating UML class-based models and OWL ontologies and technologies.

Investigating the composition of UML class-based modeling and ontology technologies requires a systematic procedure to address a series of research questions. Firstly, we need to characterize the fundamental concepts and technologies around UML class-based modeling and OWL ontologies and to elicit the requirements of an integrated framework. Consequently, we need to specify a framework that realizes the integration of both paradigms and fulfills the requirements previously elicited.

To analyze the impact of an integrated approach, we need to apply it in both domains: model-driven engineering and ontology engineering. In the domain of model-driven engineering, we apply the proposed framework to address shortcomings of software design and software languages. Our aim is to reduce complexity and to improve reusability and interoperability.

In the domain of ontology engineering, we tackle issues addressing the gap in clarity and accessibility of languages that operate ontologies, e.g., ontology translation languages or ontology APIs generation. Our framework is then used to support the development of platform independent models, aiming at improving maintainability and comprehensibility.

In the following subsections, we describe the motivation for investigating an integration between UML class-based modeling and OWL in Section 1.2. We presented the guidelines for reading this book and listed the previous publications covering parts of this book in the preface.

1.2 RESEARCH QUESTIONS

Over the last decade, the semantic web and the software engineering communities have investigated and promoted the use of *ontologies* and UML class-based modeling as modeling frameworks for the management of schemas. While the foci of these communities are different, the following question arises:

Question I *What are the commonalities and variations around ontology technologies and model-driven engineering?*

By identifying the main features of both paradigms, a comparison of both leads to the following sub-questions:

Question I.A *What are the scientific and technical results around ontologies, ontology languages, and their corresponding reasoning technologies that can be used in model-driven engineering?*

Question I.B *What are the scientific and technical results around UML class-based modeling that can be used in ontology engineering?*

While investigating this problem, our goal is to analyze approaches that use both UML class-based technologies and ontology technologies and to identify patterns involving both paradigms. The result of such analysis is a feature model, described in Chapter 4.

The feature model reveals the possible choices for an integrated approach of OWL ontologies and model-driven engineering and serves as a taxonomy to categorize existing approaches. Furthermore, the classification allows for eliciting requirements for a composed approach.

We carry out exploratory research by conducting a domain analysis over approaches involving UML class-based technologies and ontology technologies found in the literature. Domain analysis addresses the analysis and modeling of variabilities and commonalities of systems or concepts in a domain [32].

The research result is a descriptive model characterized by a feature model for the area of marrying UML class-based modeling and ontology technologies.

While there exist mappings between these modeling paradigms [114], an analysis of the outcome of an integrated approach for UML class-based modeling and OWL is lacking so far. The challenge of this task arises from the large number of differing properties relevant to each of the two modeling paradigms.

For example, UML modeling provides means to express dynamic behavior, whereas OWL 2 does not. OWL is capable of inferring generalization and specialization between classes as well as class membership of objects based on restrictions imposed on properties of class definitions, whereas UML class diagrams do not allow for dynamic specialization/generalization of classes and class memberships or any other kind of inference *per se*.

Contemporary software development should make use of the benefits of both approaches to overcome their restrictions. This need leads to the following question:

Question II *What are the techniques and languages used for designing integrated models?*

To address this question, we use the requirements resulting from Question I to propose a framework comprising the following building blocks: (i) an integration of the structure of UML class-based modeling and OWL; (ii) the definition of notations for denoting integrated artifacts; and (iii) the specification of a query solution for retrieving elements of integrated artifacts. Together, these building blocks constitute our original approach to Transform and Weave Ontologies and UML class-based modeling in Software Engineering—*TwoUse* (Figure 1.1).

We analyze the impact of the TwoUse approach with case studies in the domain of model-driven engineering and ontology engineering.

Applying TwoUse in Model-Driven Engineering. In UML class-based modeling, software design patterns provide elaborated, best practice solutions for commonly occurring problems in software development. However, software design patterns that manage variants delegate the decision of what variant to choose to client classes. Moreover, the inevitable usage of several software modeling languages leads to unmanageable redundancy in engineering and managing the same information

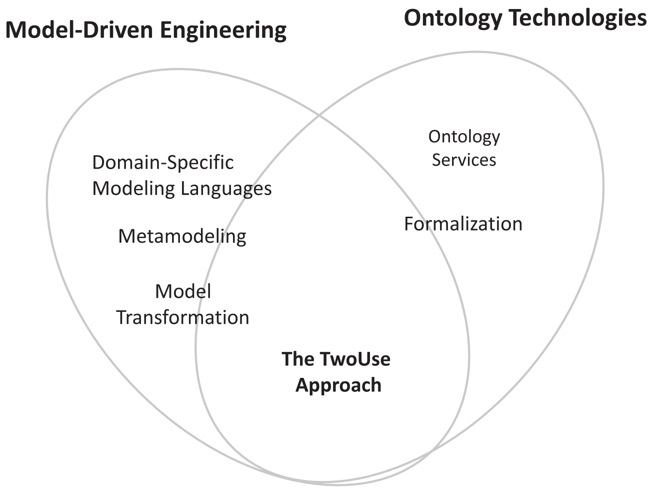

Figure 1.1 Context of the Book.

across multiple artifacts and, eventually, information inconsistency. The growing demand for networked and federated environments requires the convergence of existing web standards and software modeling standards.

In contrast, the strength of OWL modeling lies in disentangling conceptual hierarchies with multiple generalization of classes [128]. OWL allows for *deriving* concept hierarchies from logically and precisely defined class axioms stating necessary *and* sufficient conditions of class membership.

OWL provides exclusive features that distinguish it from class-based modeling languages: class expressions, individual equality, and class expression axioms. Hence, the following question arises:

Question III *What is the structural impact of using OWL constructs in designing software artifacts?*

To address this problem, we work on identifying patterns at the modeling level as well as at the language level. At the modeling level, we analyze the situation where the decision of what class to instantiate typically needs to be specified at a client class. We investigate the following question:

Question III.A *How does one determine the selection of classes to instantiate using only class descriptions rather than by weaving the descriptions into class operations?*

In systems that rely on ontologies, i.e., in ontology-based information systems, the question is the following:

Question III.B *How does one reuse existing knowledge captured by domain ontologies in the specification of functional algorithms of ontology-based information systems?*

At the language level, to support the interrelationships of software modeling languages in distributed software modeling environments, we need to answer the following question:

Question III.C *Which ontology technologies can help existing modeling languages in managing the same information across multiple artifacts and how can they do so?*

The hypothesis is that an ontology-based approach improves software quality and provides guidance to software engineers. To test the hypothesis at the modeling level, we analyze the TwoUse approach with three case studies: software design pattern, designing of ontology-based information systems, and model-driven software languages.

At the modeling level, we analyze the application of TwoUse in addressing drawbacks of software design patterns and in design ontology-based information systems. At the language level, we analyze the application of TwoUse in addressing the transformation and matching of modeling languages into OWL.

Applying TwoUse in Ontology Engineering. In ontology engineering, the design of ontology engineering services [170] has drawn the attention of the ontology engineering community in the last years. However, as ontology engineering services become more complex, current approaches fail to provide clarity and accessibility to ontology engineers who need to see and understand the semantic as well as the lexical/syntactic part of specifying ontology engineering services. Ontology engineers use services in an intricate and disintegrated manner, which draws their attention away from the core task and into the diverging platform details.

From this scenario, the problem of supporting generative techniques in ontology engineering services emerges, adding expressiveness without going into platform specifics, i.e.,

Question IV *How does one fill the abstraction gap between specification languages and programming languages?*

We propose a representation approach for generative specification of ontology engineering services based on model-driven engineering (MDE). In order to reconcile semantics with lexical and syntactic aspects of the specification, we integrate these different layers into a representation based on a joint metamodel.

The hypothesis is that filling the gap between ontology specification languages and general purpose programming languages helps to improve productivity, since ontology engineers do not have to be aware of platform-specific details. Moreover, it simplifies the tasks of maintenance and traceability because knowledge is no longer embedded in the source code of programming languages.

We validate our approach with three case studies of three ontology engineering services: ontology mapping, ontology API generation, and ontology modeling.

For ontology mapping, we present a solution for ontology translation specification that intends to be more expressive than current ontology mapping languages and less complex and granular than programming languages to address the following question:

Question IV.A *How does one fill the abstraction gap between ontology mapping languages and programming languages?*

For ontology API generation, we present a model-driven solution for designing mappings between complex ontology descriptions and object oriented representations—the *agogo* approach—and tackle the following question:

Question IV.B *What are the results of applying MDE techniques in ontology API development?*

For ontology modeling, we present a model-driven approach for specifying and encapsulating descriptions of ontology design patterns and address the following problem:

Question IV.C *How does one allow declarative specifications of templates and tools to test these template specifications and realizations?*