Understanding Ontological Engineering

VLADAN DEVEDŽIĆ

Ontological engineering has garnered increasing attention over the last few years, as researchers have recognized ontologies are not just for knowledge-based systems—all software needs models of the world, and hence can make use of ontologies at design time [1]. A recent survey of the field [4] suggests developers of practical AI systems may especially benefit from their use. This survey earmarked several application classes that benefit from using ontologies, including natural language processing, intelligent information retrieval (especially from the Internet), virtual organizations, and simulation and modeling.

Several special issues of journals and magazines dedicated to the field of ontologies [1] have described current trends in the field of ontologies, which include creating large-scale ontologies [6], defining expressive languages for representing ontological knowledge [7], and implementing systems that support ontology-based applications [12]. Unfortunately, a vast majority of these articles don't cover the relations between ontological engineering and other disciplines. As a result, specialists from other disciplines struggle to understand the benefits of ontologies, and to map the terminology of ontological engineering to their own fields. This article illustrates what ontological engineering borrows from other disciplines and what feedback it can provide to other disciplines. Also, it strives to clarify the skills useful in ontological engineering.

Ontologies, or explicit representations of domain concepts, provide the basic structure or armature around which knowledge bases can be built [10]. Each ontology is a system of concepts and their relations, in which all concepts are defined and interpreted in a declarative way. The system defines the vocabulary of a problem domain and a set of constraints on how terms can be combined to model the domain. In a distributed environment, agents use ontologies to establish communication at the knowledge level using specific languages and protocols. Ontologies are explicit representations of agents' commitments to a model of the relevant world; hence they enable knowledge sharing and reuse.

Ontological engineering encompasses a set of activities conducted during conceptualization, design, implementation and deployment of ontologies. Ontological engineering covers topics including philosophy, metaphysics, knowledge representa-

VLADAN DEVEDŽIĆ (devedzic@galeb.etf.bg.ac.yu) is an associate professor in the department of information systems, at the FON – School of Business Administration, at the University of Belgrade in Yugoslavia.

tion formalisms, development methodology, knowledge sharing and reuse, knowledge management, business process modeling, commonsense knowledge, systematization of domain knowledge, information retrieval from the Internet, standardization, and evaluation [8]. It also gives us design rationale of a knowledge base, helps define the essential concepts of the world of interest, allows for a more disciplined design of a knowledge base, and enables knowledge accumulation.

Déjà Vu in Ontological Engineering

Literature on ontologies and ontological engineering usually covers the concepts shown in Figure 1. However, other seldom-discussed viewpoints are useful for AI practitioners. If we put ontological engineering in the context of other disciplines, many similarities and analogies arise. These similarities allow practitioners to make connections between ontological engineering and other disciplines, to bridge comprehension gaps, and to see known concepts and practices in another light. Desirable qualities for ontologies, such as being decomposable, extensible, maintainable, modular and interfacable, tied to the information being analyzed, universally understood, and translatable, are also desirable for interoperable software components, or even classes of objects in object-oriented design. Practitioners from other fields may use different terminology, but its meanings are often similar. Hence the following question naturally arises: can ontologies practitioners borrow from other disciplines? Figure 2 shows aspects of two general disciplines that can help develop ontologies at the specification and conceptualization stage: modeling and metamodeling. In practice, knowledge of these disciplines helps:

- Organize the knowledge acquisition process;
- Specify the ontology's primary objective, purpose, granularity, and scope; and
- Build its initial vocabulary and organize taxonomy in an informal or semiformal way, possibly using an intermediate representation.

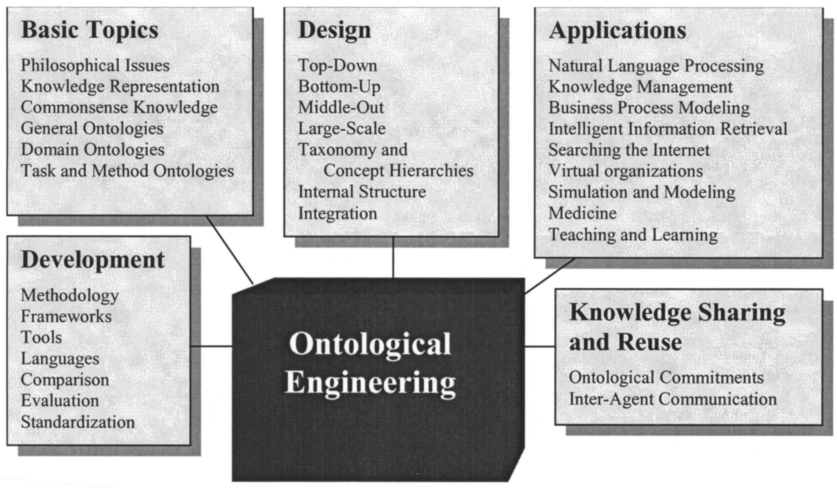

Figure 1. General themes of ontological engineering.

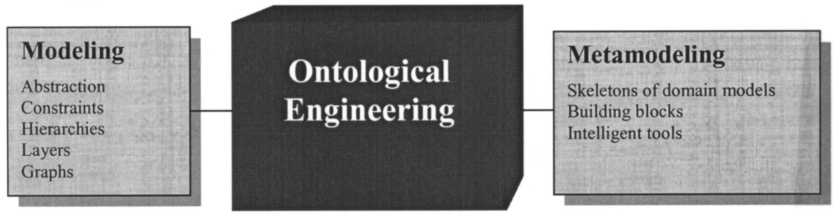

Figure 2. Modeling and metamodeling are useful to ontological engineering at the specification and conceptualization level.

Modeling. Ontologies are specific, high-level models of knowledge underlying all things, concepts, and phenomena. As with other models, ontologies do not represent the entire world of interest. Rather, ontology designers select aspects of reality relevant to their task [12]. In the domain of books, for example, the ontology designer selects one set of book attributes when developing the ontology of a library, and quite a different set when developing the ontology of bookbinding. All models follow principles and constraints, which are called concept relations and axioms.

Although there are different ways to represent ontologies, ontological engineers most frequently use hierarchical modeling (at least at the conceptualization level) [2, 6]. They often represent concept hierarchies and taxonomies in layers, and use graphs to visually enhance the representation. The layers in ontology representation range from domain-independent (core) to task- and domain-specific. Thus, ontologies contain knowledge of appropriate hierarchical and/or layered models of the relevant world.

Metamodeling. Conceptualizing and specifying ontologies have a strong metamodeling flavor. A metamodel, or conceptual model of a modeling technique, improves the rigor of different but similar models [3]. Ontologies do the same for knowledge models. Without ontologies, knowledge bases representing knowledge of the same domain are generally incompatible even if they use similar knowledge models. The good thing about using such metamodeling is we never sacrifice the usefulness of any specific model. The ontology simply provides the skeleton for the corresponding models of the domain knowledge.

Generally, an ontology is a metamodel describing how to build models. Its components—the concepts it defines and the relations between them—are always (re)used as building blocks when modeling parts of the domain knowledge. When developing a practical software system, it helps if our tools have some built-in knowledge—a metamodel or an ontology—of the models we deploy. The metamodeling function of the ontology makes the tools intelligent.

Many potentially useful parallels exist between ontological engineering and software engineering disciplines such as software architectures and software patterns, but few have been discussed explicitly or used in practical developments. Many practitioners understand similarities between ontological engineering and the object-oriented paradigm, and similarities between phases of the ontology development and software development processes, especially when looking at special-purpose software tools for developing ontologies, such as Ontolingua from Stanford University, or ODE from Polytechnic University of Madrid [7]. But practitioners can benefit from knowing more about such useful parallels. Certain software engineering disciplines

and issues rarely discussed by ontology researchers can help advance ontological engineering: software architectures, programming languages and compilers, traditional software engineering, object-oriented analysis and design, design patterns, and component-based software engineering (see Figure 3).

Software architectures. Suppose you are discussing the basics of ontological engineering with a software engineer, whose field involves designing and specifying the overall system structure and underlying organization. Conveying that ontologies are architectural armatures for building knowledge bases, models, and software architectures is probably one of the best ways to help such an individual grasp the basics of ontologies. Architectural style, an important concept in the field of software architecture, characterizes a family of systems related by shared structural and semantic properties. Mary Shaw describes several common architectural styles [9]. A style typically defines a vocabulary of design elements, design rules (constraints) for compositions of those elements, semantic interpretation of design element compositions, and analyses on systems built in that style. Many successful designs can share a style.

Styles contain condensed skeletons of the architectural knowledge gained by experienced software designers, and provide a means to reuse that knowledge. For example, layered style is suitable for applications involving distinct classes of services that can be arranged hierarchically. Designers often define layers for: basic system-level services, utilities appropriate to many applications, and specific application tasks. Similarly, some ontologies structure knowledge in layers to separate the use-specific knowledge from the core (and more reusable) knowledge [8, 12]. Other archi-

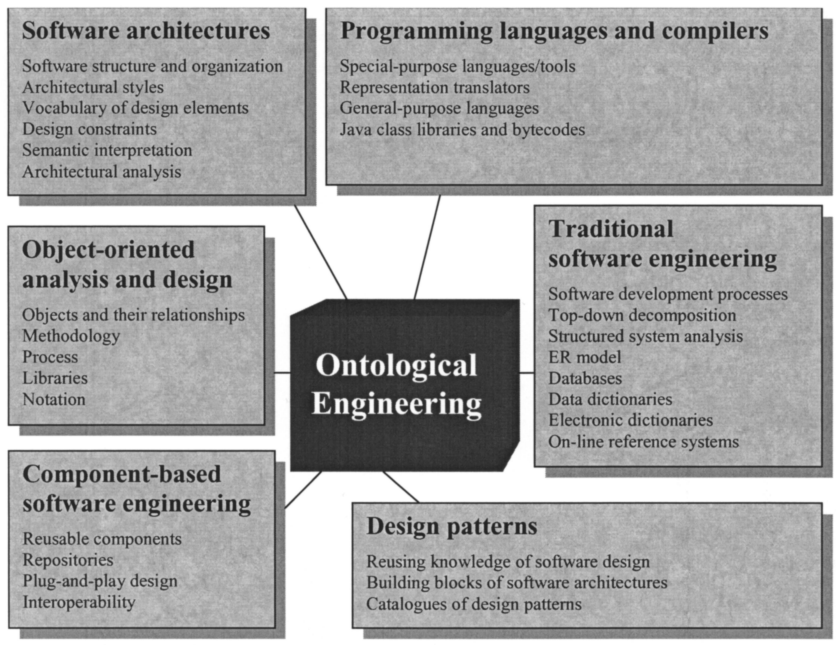

Figure 3. Software engineering disciplines useful to ontological engineering.

tectural styles that help define ontological engineering solutions include pipeline and data abstraction [9].

Programming languages and compilers. Special-purpose languages/tools for implementing ontologies, such as Ontolingua, CycL, and LOOM, use a frame-based formalism, a logic-based formalism, or both [4, 6, 7, 12]. For example, Ontolingua is a frame-based language that uses KIF (Knowledge Interchange Format), a language for publication and knowledge communication that has notation and semantics of an extended version of first-order predicate calculus. It enables writing knowledge-level specifications, independent of particular data or programming languages, and translating knowledge bases from one specialized representation language into another. Several other languages/tools for building ontologies also use such a translation approach, which enables the building of ontologies directly at the knowledge level. This approach eliminates the need to master implementation languages and use translators to translate from the knowledge level to the implementation level. But there are problems with this approach. First-order logic is rather restricted, to enable painless translation of rich and often ill-structured or unstructured knowledge-level specifications into well-formed predicate calculus expressions. Also, there are reports of problems using specific translators. For example, when using the Ontolingua translator to produce the equivalent Loom ontology of time from a prebuilt Ontolingua-based ontology, the resulting translation had value only as a first draft [12]. Extensive manual adaptations of the translated ontology were necessary in order to make it fully applicable.

Practitioners should note more recent languages and tools make extensive use of techniques and tricks from the compiler theory to improve the quality and the capabilities of the translation process. For example, the ODE environment uses a generic translator that allows the user to specify the ontology in a user-oriented internal representation and translate it automatically into the target language (in ODE's case, into Ontolingua) [7]. The translator uses a grammar to declaratively express the conceptual model (the internal representation) in the BNF form. For each type of valid definition in the language, there is a table that relates the terms used in the transformation rules to the terms employed in the conceptual model. It is easy to build a new translator by merely changing the rules that identify the transformation rules of the terms to be generated, and changing the appropriate table relating the conceptualization to the implementation.

It is also useful to consider other programming languages besides special-purpose languages from the perspective of ontological engineering. In addition to defining a general ontology of a programming language, with concepts like identifier, reserved word, and construct, one may abstract an ontological skeleton from any programming language. This is actually a hint for practitioners that many ontologies implicitly exist in programming languages and should not be reinvented. For example, Java has at least two obvious but rarely noted parallels with ontological engineering concepts:

- Classes from Java class libraries make up a hierarchy (with the Object class on top) that can be viewed as an extremely well elaborated ontology; and
- Java bytecodes are actually an "interlingua" any Java interpreter can understand, in a common interchange format that makes Java fully interoperable and platform-independent—the same idea behind KIF and other ontology-related languages.

The GKS (Graphical Kernel System) language and standard for creating graphical primitives also represents an ontology of such primitives. Various platform-specific implementations of GKS libraries allow for sharing and reuse of GKS primitives in many graphical systems.

Traditional software engineering. Since the AI community develops ontologies, and uses special-purpose tools and languages to do so, many think of ontologies as a trend involving sophisticated methodology and technology. But an ontology is always about entities and relationships, and often methodology from traditional software engineering, such as using the ER model, top-down decomposition strategy, and structured system analysis, are used to represent it. For example, the Methontology framework for developing ontologies [7] proposes a close relative of the traditional waterfall process of software development for an ontology development lifecycle. Moreover, the entire chemicals ontology developed using the Methontology framework is stored in a relational database, which can encode its ontology in its data dictionary [8]. Fridman-Noy and Hafner discuss examples of using online lexical reference systems and electronic dictionaries as general ontologies [4]. All design criteria for ontologies, such as clarity, extensibility, coherence, and minimal encoding bias also represent design criteria for software systems modules. Ontology researchers and developers can explore a large variety of iterative and incremental traditional software development methodologies for new ideas in ontological engineering.

Object-oriented analysis and design. The processes of ontology development [7, 8, 12] nearly coincide with those of object-oriented analysis and design [3, 11]. In both cases, it is important to assemble the domain vocabulary in the beginning, often starting from the domain's generic nouns, verbs, and adjectives. Object-oriented analysis stresses different aspects than ontological analysis [8], but parallels are obvious. The result of object-oriented analysis is a draft of the domain ontology relevant to the application (although analysts don't call that result an ontology). And as object-oriented designers define classes, objects, hierarchies, interface functions, and system behavior, ontological engineers use intermediate representations such as semantic networks, graphs, and tables to design hierarchies and other concept relationships. Both types of specialists use templates to specify product details [3, 7]. Classes can be merged or refined, as with ontologies. Class libraries and previous design specifications often provide reuse in object-oriented design, as do previously encoded and publicly available ontologies.

One area of ontological engineering requiring additional efforts involves developing a generally accepted notation for representing ontologies. Software engineers have used several different notations in object-oriented design over the past decade, but all have converged to the Unified Modeling Language (UML) notation [3], which provides a metamodel of object-oriented design. It defines graphical notation for representing classes, objects, and their relationships in four different views (logical, use-case, component, and deployment), covering all practical aspects of object-oriented design. It would be nice if ontological engineers had a standard notation that everyone accepted, understood, and used in practice. Unified graphical representation for such a meta-language could help construct visually rich and easy-to-use tools [12], and would increase knowledge reuse at the design level, but unfortunately most ontology developers currently use their own notation.

From the practitioner's perspective, important differences exist between ontological engineering and object-oriented analysis and design. "Ontological" means taking a knowledge-level stance in describing a system [1], while "object-oriented" largely refers to the means of design and implementation. In a semantic-based information retrieval system, for example, ontologies specify the meaning of the concepts to be searched for, while in the object-oriented design of such a system, ontologies represent the domain models. Object-oriented design languages like UML offer explicit design methodology and notation for all design artifacts, but ontological and metamodeling principles are only implicit in those languages [8]. In other words, an ontology is what can be abstracted at the knowledge level from the corresponding class diagrams, object diagrams, and use-case diagrams, represented in any object-oriented notation such as UML. The role of ontology is to convey and explicitly specify domain concepts, terms, definitions, relations, constraints, and other semantic contents that object-oriented analysis and design should rely on and support.

Design patterns. Design patterns, described as "simple and elegant solutions to specific problems in object-oriented software design" [5], provide a common vocabulary for designers to communicate, document, and explore design alternatives. They contain the knowledge and experience underlying many redesign and recoding efforts to achieve greater software reuse and flexibility. Although design patterns and ontologies are not identical, practitioners should be aware these two concepts overlap, and software patterns may be used along with other sources of ontology design in practical developments. Both concepts involve vocabularies, knowledge, and "architectural armatures," and describe concepts at the knowledge level. Ontologies are more commonsense-oriented, while design patterns are more concrete. But besides activities such as software design, software patterns can also involve abstract activities such as organizational and analysis patterns [2, 5]. One can draw an analogy between libraries of ontologies and catalogues of software patterns. Design pattern catalogues are not ready-to-use building blocks as are ontologies from libraries, but efforts are ongoing to make them ready-to-use blocks. It doesn't take a hard mental shift to view ontologies as abstract patterns, or knowledge skeletons of some domains. Likewise, it is easy to understand software pattern templates as knowledge of how software pattern ontologies may look.

Component-based software engineering. A long-term objective of ontological engineering is to build libraries of reusable knowledge components and knowledge-based services that can be invoked over networks. Similarly, the component-based software engineering field is struggling to develop repositories of reusable, pretested, interoperable, and independently upgradable software components that enable plug-and-play design and software development. These objectives necessitate designing systems from application elements constructed independently by developers using different languages, tools, and computing platforms [11].

Can ontologies be components and vice versa? Ontologies are conceptually more abstract than components, but it seems components can be parts of ontologies. It is also possible to develop a component that fully corresponds to an ontology. Different domains and ontologies can share components from repositories. Ontologies should be, in a sense, a basis for designing and developing interoperable software components in practice, since they can precisely define the semantics of components and their parts, as well as the types of relations and communication between software components.

Goals for Practitioners

For practitioners, developing an ontology is never simple-practical experience in related disciplines is extremely important. The main purpose of developing ontologies is to clarify the domain's structure of knowledge and to enable knowledge sharing and reuse [1]. However, in practical terms ontological engineering means achieving goals such as the following [1, 2, 6, 8]:

- Precisely defined terms and highly structured definitions of domain concepts, not just text-based information;
- Consensus knowledge of a community of people;
- High expressiveness, enabling the ontology users to say what they wish to say;
- Coherence and interoperability of resulting knowledge bases;
- Stability and scalability of ontologies; and
- A foundation for solving a variety of problems and constructing multiple applications.

Although ontologies are rather content-related than representation-related, achieving these goals calls for formalization and co-existence of artistic creativity and systematically applied knowledge from other disciplines. An ontology can be developed collaboratively by many distributed individuals and organizations with differing expertise, goals, and interactions. Various communities of experts and practitioners examine problems from different angles and are concerned with different dimensions of the content's semantics and representation. These individuals all need to properly understand each other and meaningfully communicate their views of domain knowledge to form meaningful higher-level knowledge: the ontology.

Once application developers are ready to use the ontology, they should be able to convert it into a desired form, such as a database, object base, or knowledge base, using a representation and language of their own choice. Thus, ontological engineering must rely on several content formats, translation frameworks, and development strategies that reduce semantic ambiguity and allow for sharing and reusing knowledge and practices from other disciplines.

Finally, developing ontologies in order to enable knowledge sharing and reuse most often means it is actually intelligent agents and agent-based systems that will use the ontologies for communicating and exchange knowledge among themselves. So, ontological engineering also involves developing higher-level knowledge-based products that express the consensus knowledge of a community of agents.

Conclusion

Ontologies are needed in all software systems, which must always "know" about entities and their attributes and relationships in the relevant world. All systems need knowledge, whether about data structures, methods, or algorithms. Domain ontology encodes such knowledge, as in a relational database management system that "knows" about its relational tables, data records, and their fields. Because ontologies are everywhere, they make possible to smoothly integrate artificial intelligence with other software disciplines.

Putting ontological engineering in the context of other disciplines enables both ontological engineers and other specialists to view their fields from different perspectives. To specialists, drawing analogies between their fields and ontologies helps

explain the déjà vu feeling engendered by certain ontological engineering concepts. To ontological engineers, awareness of such similarities may create new ways build and improve ontologies. Building, using, and reusing ontologies requires much work and many difficult modeling decisions. However, practitioners can facilitate their job of rooting applications in ontological foundations if they use knowledge, practices, and solutions from other disciplines. A clear understanding of what, when, and how to borrow from other disciplines helps enormously.

References

1. Chandrasekaran, B., Josephson, J.R., Benjamins, V.R. What are ontologies, and why do we need them? *IEEE Intelligent Systems 14*, 1 (Jan./Feb. 1999), 20–26.

2. Devedzic, V. and Radovic, D. A framework for building intelligent manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 29*, 3 (Aug. 1999), 422–439.

3. Fowler, M. and Scott, K. *UML Distilled: Applying the Standard Object Modelling Language.* Addison-Wesley, Reading, MA, 1997.

4. Fridman-Noy, N., Hafner, C.D. The state of the art in ontology design: a survey and comparative review. *AI Magazine* (Fall 1997), 53–74.

5. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

6. Lenat, D.B. CYC: A large-scale investment in knowledge infrastructure. *Commun. ACM 38*, 11 (Nov. 1995), 33–38.

7. Lopez, M.F., Gomez-Perez, A., Sierra, J.P., Sierra, A.P. Building a chemical ontology using methontology and the ontology design environment. *IEEE Intelligent Systems 14*, 1 (Jan./Feb. 1999), 37–46.

8. Mizoguchi, R. A step towards ontological engineering. *Proceedings of The 12th National Conference on AI of JSAI* (June 1998), 24–31.

9. Shaw, M. Patterns for software architectures. In: J.Coplien, D. Schmidt (eds), *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA (1995), 453–462.

10. Swartout,W. Tate, A. Ontologies, Guest Editors' Introduction, *IEEE Intelligent Systems 14*, 1, Special Issue on Ontologies (Jan./Feb. 1999), 18–19.

11. Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1998.

12. Valente, A., Russ, T., MacGregor, R., Swartout, W. Building and (re)using an ontology of air campaign planning, *IEEE Intelligent Systems 14*, 1 (Jan./Feb. 1999), 27–36.