Review Articles

An Overview on the use of Ontologies in Software Engineering

Daniel Strmečki, Ivan Magdalenić and Dragutin Kermek

Faculty of Organization and Informatics, University of Zagreb, Varazdin, Croatia

Abstract: One of the main goals of the Software Engineering (SE) discipline is to find higher abstraction levels and ways to reuse software in order to increase its productivity and quality. Ontologies, which are typically considered as a technique or an artifact used in one or more software lifecycle phases, may be used to help achieve that goal. This paper provides a literature review, discussion and analysis of the existing solutions for implementing ontologies in SE. We selected several software development paradigms (including Software Product Lines, Component-Based Development, Generative Programming and Model-Driven Engineering) for our classification and discussion of different approaches proposed in the literature. It was established that ontologies are suitable for providing a common vocabulary to avoid misunderstanding between different parties in SE, requirements specification, features specification, variability management, components specification, components matching, model transformations and code generation. Based on the conducted review, guidelines for further research are given.

Keywords: Ontologies, Software Engineering, Software Product Lines, Component-Based Development, Generative Programming, Model-Driven Engineering

Introduction

Software Engineering (SE) is the application of a systematic, disciplined and quantifiable approach to the development, operation and maintenance of software (Bartolo Espiritu et al., 2014). A basic goal of SE as a discipline is to successfully manage and control software complexity. The increase of complexity in software products and high development and maintenance costs have resulted in a large number of unsuccessful SE projects. This phenomenon, referred to as software crisis, implies the difficulty of writing useful and efficient code within the required time. Although SE has witnessed great progress since the appearance of software crisis, examples of large failed projects can still be found in the literature (Robal et al., 2015). The two key software projects issues are: (1) Low-level design and implementation techniques; (2) Exposure of more details than intended, in order to make software product design, construction and modification simple (Batory, 2006). Software reuse is an important area of the SE discipline, which has a potential of increasing its productivity and quality (Nianfäng et al., 2010). On the other hand, owing to its complexity, numerous factors like deadlines, budget, technology, architecture and the level of knowledge need to be taken into account. Raising the abstraction level has been the most commonly used SE approach to increasing software reuse. SE researchers are thus constantly looking for higher abstraction levels to enhance productivity and quality of software products. By encapsulating knowledge about lower level operations, developers can think in terms of higher level concepts, thus saving the effort and time of composing lower-level operations, in other words, avoiding reinventing the wheel in every project (Visser, 2008). However, in SE there are no universal solutions, as it entails creative processes which are always critically dependent on the unique abilities of the creative people who perform them (Musen, 2000). Even when the used technology is not very complicated itself, the engineering and realization of an SE project usually is. This is mainly caused by the number of parties involved and many different interpretations of the system (van Ruijven, 2013). Nowadays, high-level object-oriented programming languages are employed in SE with the aim of raising the abstraction level using various
modeling and metaprogramming techniques. In this study we provide a literature review in order to investigate possible approaches to making use of ontologies in SE for dealing with the aforementioned problems.

**Ontology** is a philosophical term that refers to the study of being, becoming, existence and reality. It was introduced to computer science through the field of *Artificial Intelligence* (AI). AI has contributed greatly to the field of SE through conceptual modeling techniques, methods for system analysis and frame-based knowledge-representation systems (Musen, 2000). In AI ontologies are used in knowledge management for limiting complexity and organizing information. The accepted definition of ontology in computer science is that it is a formal and explicit specification of a shared conceptualization (Wilson et al., 2007). *Conceptualization* can be understood as an abstract and simplified version of the presented world, a knowledge representation based on objects, concepts and entities. *Formal* means that a machine can process it, while *explicit* means that there are clear restrictions applied to the representations (Calero et al., 2006). According to another definition of ontology in computer science, it is an effort to formulate an exhaustive and rigorous conceptual schema within a given domain (a hierarchical data structure containing all the relevant elements and their relationships and rules) (Wongthongtham et al., 2008). Dillon et al. argue that a true ontology should contain not only a hierarchy of concepts, but also other semantic relations that specify how one concept is related to another. These authors also announced the dawn of SE 2.0, that is, the use of semantics as a central mechanism that would revolutionize the way software is developed and consumed (Dillon et al., 2008). John stated that ontologies form a pool of reusable, shared knowledge resources. They constitute a special kind of software artifacts which includes a certain view of the world, designed with a purpose of explicitly expressing the meaning of a set of existing objects (John 2010). Over the past decade, with the emergence of the *Semantic Web*, several ontology languages have been presented, one of which became a standard in SE: The *Web Ontology Language* (OWL). It is a *World Wide Web Consortium* (W3C) standard ontology language that provides a complete set of expressions for capturing different concepts and relationships that occur within ontologies (Wongthongtham et al., 2009). It was developed to facilitate greater machine interpretability of human knowledge by providing additional vocabulary along with formal semantics (Bossche et al., 2007). OWL is based on *eXtensible Markup Language* (XML), *Resource Description Framework* (RDF) and *Description Logic* (DL), a family of logic-based knowledge representation formalisms (Duran-Limon et al., 2015).

Although ontologies are typically considered to be a technique or an artifact used in SE, it is also possible to use them for the representation of SE domain knowledge. Whereas SE generic ontologies are aimed at modeling the complete SE body of knowledge, SE specific ontologies conceptualize one part of the discipline with a specific goal (Hilera and Fernandez-Sanz, 2010). Ontologies were initially used by software applications to store data and their semantic meaning, while they are now used to aid software development in every phase of its lifecycle. *Ontology-Driven Software Engineering* (ODSE) is a software development approach where ontologies are used to perform a majority of operations in software development. Those operations can range from system modeling to software generation (Wiebe and Chan, 2012). This paper provides an overview on the usage of ontologies in SE. Since it a relatively new approach in software development, no time limit was set for this literature review. The idea for making use of ontologies in SE emerged at the beginning of the 21st century and is currently still a very popular topic in the computer science research community.

The remainder of this article is organized in five sections. Section 2 describes the research areas, precisely the four SE paradigms that this review will focus on. These four SE paradigms were chosen because they (to some extent) deal with software development automation, a topic of the authors’ special interest. Section 3 provides a literature review and analysis of suggested approaches for implementing ontologies in SE. The review section consists of four subsections that present the existing solutions for the usage of ontologies related to the four selected SE paradigms. Section 4 provides a critical analysis and discussion of the solutions found in the literature. Section 5 summarizes the whole article, while possible directions for our future work on this topic are provided in section 6.

**Research Areas**

*Software Product Lines* (SPLs), sometimes also referred to as software families, imply a set of software products that consist of a common architecture and a set of reusable assets used in systematical production (Asikainen et al., 2007). The SPL discipline focuses on systematic, planned and strategic reuse of core assets in order to produce a number of products that satisfy a particular market segment (Duran-Limon et al., 2015). SPLs tend to make use of a system’s common features to increase the productivity and quality, reduce the development time, costs and complexity (Strmečki et al., 2015). The main goal of SPLs is to avoid developing software from scratch by reconfiguring and reusing existing SPLs across different projects. SPLs development consists of two processes: *Domain
**Engineering (DE) and Application Engineering (AE)** (Magdalenić et al., 2013). The goal of DE is to develop a common architecture for a system family and to devise a production plan for the family members (Strmečki et al., 2015). In the DE process, the commonality and variability of a product line is established, reusable assets that accomplish the desired variability are constructed and mechanisms for resolving variability are defined. The AE process is concerned with derivation and development of a particular product. The product’s specific requirements are taken into account and the defined variability mechanisms are used in order to develop the product timely and with the lowest cost, but at high quality (Nguyen et al., 2015).

**Component-Based Development (CBD or CBE)** is a software development paradigm based on the provision of reusable software components in a plug-and-play development style. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components. A dream for software components integration is to be able to compose complex systems from off-the-shelf components (Ringert et al., 2014). SE by components composition is suited for development in distributed systems such as the Web. The main goal of distributed CBD is to increase the reliability and maintainability of software systems through components reuse (Pahl, 2007a). Components also have an important role in currently popular Service-Oriented Architectures (SOA) because components can be converted into Web services and thus can provide services to other components.

**Model-Driven Engineering (MDE)** is a paradigm that emphasizes model-based abstraction and automated code generation. In MDE, essential features of a system are captured through appropriate models and code generators are used to automatically produce the source code for the various modeled entities (Magdalenić et al., 2013; Lilis et al., 2014). A model represents a partial or a simplified view of a system. It is an abstraction of a system used to replace the system under study. In the MDE paradigm, models are not considered merely as a documentation artifact, but rather as reusable artifacts used throughout the whole SE lifecycle (Rodrigues da Silva, 2015). MDE combines layered modeling techniques with automated transformations and code generation, where the generated code may contain special tags carrying model information. It is based on three layers: (1) Computation Independent Model (CIM) addresses the structural aspects of the system from a computation-independent viewpoint; (2) Platform Independent Model (PIM) defines a system as a technology-neutral virtual machine or a computational abstraction; 3) Platform Specific Model (PSM) consists of a platform model that makes up the platform and an implementation-specific model towards concrete implementation. MDE is platform-neutral by definition, but the archetypical MDE is based on Unified Modeling Language (UML) (Pahl 2007b; Katasonov and Palviainen, 2010). MDE provides an approach to: (1) Specifying systems independently of the platform; (2) Choosing a particular platform; (3) Transforming the specification for the chosen platform (Bartolo Espirito et al., 2014). MDE developers first model the whole system in UML and then take iterating steps to refine the model. The final model is concrete enough for executable code to be generated from it. Since the developer operates at a high level of abstraction, efficiency is also achieved. However, it is hard to use MDE for general purpose programming and a lot of complexity can be hidden in the generators (Zimmer and Rauschmayer, 2004).

**Generative Programming (GP)** is a discipline within Automatic Programming (AP) that uses generators to facilitate the process of application development (Magdalenić et al., 2013). A generator is defined as a program that takes a higher-lever specification of a piece of software and produces its implementation (Czarnecki and Eisenecker, 2000). Generative programming uses DE techniques and can be applied in SPLs AE. In addition to UML modeling, GP uses feature modeling proposed in the Feature-Oriented Domain Analysis (FODA) method. Feature modeling can be defined as a creative activity of modeling common and variable properties of concepts and their interdependencies. The term feature refers to a property of a system relevant to a stakeholder. Feature is used to capture the commonality and variability among products. A generative domain model focuses on mapping between problem space and solution space. Problem space refers to a set of features of a product family, while solution space denotes implementation-based abstractions contained in the specification. A generator maps the two spaces, using a specification to yield the corresponding implementation (Magdalenić et al., 2013). GP uses metaprogramming techniques, which refers to the development of programs designed to read, generate, analyze or transform other programs and even modify themselves while running (Strmečki et al., 2015).

**Literature Review**

The review starts by presenting related work on ontologies categorization, focusing on their usage in SE. The solutions for the usage of ontologies in each of the four selected SE paradigms that have been proposed in the literature are further presented in separate subsections. Happel and Seedorf put forth an idea for integration of SE and Knowledge Engineering (KE) approaches, with examples of ontology applications throughout the SE lifecycle. They suggested concrete approaches for using ontologies in all of the SE phases:
Analysis and design (requirements specification and component reuse), implementation (software modeling, domain object model, coding support and documentation), deployment and runtime (semantic middleware and Web services) and maintenance (project support, updating and testing). Happel and Seedorf also noted that integrating SE and KE approaches tends to be academic, neglecting the applicability aspects and providing little guidance for software engineers. They categorized the use of ontologies in SE into four approaches: (1) Ontology-driven development (the usage of ontologies at development time to describe the problem domain); (2) Ontology-enabled development (the usage of ontologies at development time to support developers in their tasks); (3) Ontology-based architectures (the usage of ontologies as primary runtime artifacts); (4) Ontology-enabled architectures (the usage of ontologies as support to runtime software) (Happel and Seedorf, 2006). Guarino and Fensel established similar classifications of ontologies based on their generality level: (1) Generic ontologies capture general knowledge of the world and are applicable in a variety of domains; (2) Representational ontologies provide entities without expressing what they represent and do not belong to any particular domain; (3) Domain ontologies capture the knowledge applicable in a particular domain; (4) Method and task ontologies capture the knowledge specific to problem resolution methods or specific tasks (Fensel, 2004). Calero et al. provided a broader classification of ontologies based on their subject of conceptualization: (1) Knowledge representation ontologies are used to formalize knowledge under a concrete paradigm; (2) Common or generic ontologies represent reusable common-sense knowledge; (3) High-level ontologies describe general concepts and notions; (4) Domain ontologies offer vocabulary for concepts in a particular domain; (5) Task ontologies describe the vocabulary related to a generic activity; (6) Domain task ontologies are reusable only in a particular domain; (7) Method ontologies are applicable to a reasoning process designed to perform a particular task; (8) Application ontologies are dependent on the application and often specialize the vocabulary of a domain or task ontology. Calero et al. also emphasize that, based on the moment when they are utilized, ontologies can be used during the development or in runtime. The former approach is termed Ontology-driven development, in which, for example, ontology’s semantic content can be converted into a system component. When the system makes use of an ontology with a specific purpose, it is referred to as an Ontology-aware system. An Ontology-driven system is the one in which the ontology is an additional cooperating component. The final taxonomy of ontologies in software engineering and technology proposed by Calero et al. contains two generic categories: Ontologies of domain and Ontologies as software artifacts. The SE Ontologies of domain taxonomy can be categorized as generic of specific (requirements, design, construction, testing, maintenance, configuration management, quality, engineering tools and methods, engineering process and engineering management). Ontologies as software artifacts can be categorized as development time ontologies (development process, maintenance process, customer-supplier process, support process and management process) and runtime ontologies (architectural artifacts and information resources). Based on the mentioned taxonomy, authors provide a comprehensive review and classification of proposals found in the literature (Calero et al., 2006). Gašević et al. investigated the use of ontologies in SE throughout software lifecycle phases. In the analysis phase, an ontology is commonly used for Requirement Engineering (RE). In the design phase, ontologies are used as software models, business vocabularies and reasoning or transformation models. In the implementation phase three possible approaches can be distinguished: (1) Software system implementation can be generated from an ontology created in the analysis phase and refined in the design phase; (2) Ontologies can be used in runtime, e.g., Jena API can be used for handling OWL ontologies in Java; (3) Ontologies can be used as part of the implementation logic in systems implemented using rule-based languages. In the maintenance phase, ontologies may be used as support for managing knowledge. Gašević et al. even propose the use of ontologies in the retirement phase, as a repository of retired software’s knowledge and state that this issue has not been addressed so far. They also identify the need for developing standard ontologies of documentation structure and types. With regards to using ontologies for testing, they highlight the importance of further exploration of topics such as semantic annotation of logs for intelligent monitoring, semantic annotations of unit and integration tests, ontology-based reverse engineering and ontology-based software metrics. They also recommend that further research be undertaken on topics concerning the use of ontologies as software artifacts, including annotation mechanisms of software models and implementation code, integration of ontologies and meta-modeling architectures as well as a comprehensive tractability model of software artifacts. They find ontologies to be suitable for describing SE processes and methodologies, for example, by connecting tasks and activities to artifacts they have produced or used, as well as to responsible participants and their interactions (Gašević et al., 2009).

Software Product Lines (SPLs)

Gašević et al. state that software is a knowledge repository largely related to an application domain, rather than to software as an entity. They argue that an
ontology should be a product of the analysis phase, meaning that all parties involved in the process should agree on the ontology development. The main advantages of their approach are avoiding the risk of misunderstanding the user’s needs, availability of semantically annotated documentation and the ability to use the ontology in the design phase (Gašević et al., 2009). Bossche et al. used an ontology as a ‘contract’ between Business and IT in their Ontology Driven Architecture for Software Engineering (ODASE). In their approach, a Business representative works with a domain modeling expert to build a formal model (an OWL conceptual model). In that way, the Business is forced to provide explicit requirements, which enables the IT to give a reasonable time and cost estimate for the project. The ontology is used to achieve a common agreement between Business and IT. Moreover, business knowledge is reusable and not lost in the code, a single language is used by domain experts and software engineers and IT can rely on formal semantics (Bossche et al., 2007).

According to Happel and Seedorf, an ontology may be used both to describe the requirements specification documents and to formally represent requirements knowledge. In contrast to traditional approaches, ontologies are suited for evolutionary approach to the specification of requirements and domain knowledge (Happel and Seedorf, 2006). Shunxin and Leijun argue that ontologies can achieve a higher degree of knowledge than traditional requirements analysis methods by sharing and reusing requirements. An ontological model can ensure that requirements are traceable, consistent, complete and correct. It also provides a platform for the user, requirements analyst and developer to communicate. Basic concepts and relations of a domain ontology can be formed into a hierarchical structure to form an application ontology. Although the development of each software product starts with requirements analysis, it must be conducted repeatedly throughout the project, in parallel with the management and development of the software. They also predict that reusing and sharing of existing ontologies will become the focus of future research (Shunxin and Leijun, 2010). Karatas et al. indicated that RE is a process that in SPLs usually takes longer than planned and is more costly than originally budgeted for. They also noted that requirements reuse is not getting as much research attention as design and implementation. Addressing systematic requirements reuse necessitates a model for reusable requirements elements. Karatas et al. propose an ontology-based domain knowledge formalization for SPLs. They favor ontology modeling over feature modeling due to its descriptive power. They devised a graphical automation tool for requirements reuse and documentation called OntSRDT that leads users to valid SPL configurations and documentation specification for that configuration. Authors acknowledge some drawbacks in their solution that will be the focus of their future work. They concede that the coverage of the ontology model is not complete, that the conformity of requirements specifications to quality standards should be checked and that more work is required on the tool support, which should be open for public use as a product configurator (Karatas et al., 2014). Siegemund et al. also recommend the usage of ontologies in RE. They argue that OWL is ideal for this purpose as it allows reasoning over incomplete knowledge. By applying requirements reasoning based on formal semantics, many of the shortcomings of other approaches can be avoided, such as insufficient coverage of requirements knowledge, inadequate capture of requirements relationships, late detection of requirement-related problems, completeness and consistency verification and low abstraction levels. They suggest further work on the guidance of the RE process and traceability through other SE stages (Siegemund et al. 2011). Sim and Brouse presented the OntoPersonaURM model to support and enhance the RE activities. Personas are specific and concrete representations of target users. In their model, authors use ontologies to represent knowledge about users. A concept of a persona is integrated into a unified environment to help engineers and developers gain better understanding of user’s needs and identify missing requirements in the shortest possible time. Their further work will be directed at improving the model and constraints checking as well as to checking for requirements correctness, completeness and consistency (Sim and Brouse, 2015). Robal et al. established a domain ontology for describing probable SE concepts to a ‘smart customer’ who has a basic understanding of ICT and can have ontology concepts presented to him from the management perspective. The ontology describes the domain knowledge for common vocabulary and can also be used to derive different customer and developer profiles, so it can be applied in areas such as education, training, customer and team profiling (Robal et al., 2015).

Ceh et al. discuss the ontology-based domain analysis and how it can be incorporated into the Domain Specific Language (DSL) design phase. They argue for the use of ontologies in DE instead of using demanding formal methodologies. Their Ontology2DSL framework enables automated DSL grammar construction from a target ontology. It accepts an OWL document as input and produces the corresponding DSL grammar and programs. Its architecture is comprised of an OWL parser, rule reader, rule executor and transaction logger. Further work is required to fully develop the framework and enable the addition of custom rules and
transformation patterns (Ceh et al., 2011). Mezhuyev proposed a DSL definition based on domain ontology. The model of ontology, which serves as a base for the definition of the meta-model for different DSLs, is expanded in the definition of the meta-model by grammar ruler and mathematical methods. The suggested approach takes into account the domain’s specifics, manages users in accordance with the developed process model, enables application of mathematical methods to solve arising domain tasks and enables formalization of different properties and technologies. The author acknowledges the lack of tool support by mentioning that future work will be focused on finalizing the software tools to implement the proposed approach (Mezhuyev, 2014).

Musen states that ontologies provide language that is understandable by both developers and computers and that they can be used to build knowledge bases containing detailed domain descriptions. Domain ontologies provide both a framework for conceptual analysis and design and the actual code that can be reused for new applications. Musen also argues that in modern technologies problem solving methods are not procedures operating on a predefined data structure, but rather procedures operating on ontologies (Musen, 2000). Asikainen et al. constructed a domain ontology for modeling the variability in SPLs named Kumbang. Their ontology synthesizes existing variability methods based on feature and architecture modeling. Its main purpose is to support the task of managing variability in SPLs and to configure them to meet specific requirements. However, the authors acknowledge the lack of tools support which still needs to be developed. In order to fully demonstrate the practical applicability of the approach it is also necessary to test it on real software product families in real software development contexts (Asikainen et al., 2007). Czarnecki et al. explored the relation between feature modeling and ontology modeling. They concluded that a set of features may be mapped to a set of ontology elements and that a many-to-many association exists between those elements. A feature model represents a set of restrictions that can be applied to ontologies. It is suggested that the combination of feature models and ontologies can be transferred to tool support development utilizing query and constraint mechanisms (Czarnecki et al., 2006). The OntoAD framework, introduced by Limon at al., transforms both the feature model and the SPL architecture into an ontology. The ontology allows reasoning about the relationship between architectural elements and features. The Rule Generator Engine is used to produce the transformation rules for mapping the variability to architecture components. The Ontology Generator Engine is then used for carrying out a model-to-text transformation in which both features and architecture are transformed to an instance of the ontology. OntoAD supports different architecture languages and reduces the manual derivation tasks. Authors suggest that relationships between features and architectural elements should be further explored and the issue of supporting the design of composite components in an SPL architecture should be addressed (Duran-Limon et al., 2015). Lepuschitz et al. establish that modern manufacturing systems face dynamic conditions and need to be capable to quickly react to sudden changes of demands. They employ ontologies as a formal expression of legitimate states of the system and representation of details about its physical components. They used a combination of ontological knowledge and agent approach for dynamic creation and online configuration of component’s control applications. The agents rely on an appropriate resource ontology that provides sufficient details about the physical components they control (Lepuschitz et al., 2011).

Component-Based Development (CBD)

Happel and Seedorf claim that ontologies can be helpful in describing the functionality of components, thus enabling powerful semantic queries. They argue that the use of ontologies in analysis, design and implementation is highly suitable for rapid application development (Happel and Seedorf, 2006). Pahl demonstrated the usage of two types of ontologies in the component-based development context: An application domain ontology, which describes the software being developed and the software development ontology, which describes the development entities and processes (Pahl, 2007a). According to Hesse, although in ODSE an ontology was initially viewed as a special component in the software development process, it is likely that later it will represent knowledge used in many other components. An incremental approach is appropriate for developing an ontology, which gets extended in accordance with the project’s progress and development. Ontologies also use a component-based structure, as in time the ontology gets decomposed to sub-ontologies which are then developed independently (Hesse, 2005). Nianfang et al. argue that an efficient method is needed for describing software components in order to efficiently reuse them. They defined a component as a useful software unit with semantic integrity and correct grammar. They introduce a components descriptive model (based on a 3C model) which uses ontologies to describe components. It contains the description of components’ base information, interface, function, environment and quality (Nianfang et al. 2010). Wiebe and Chan argue that describing the software by using ontology can lead to a high level realization of component-based development. A complex software
problem could be given to semantic agents to piece together a project from specification ontologies and their components (Wiebe and Chan, 2012). XCM is an ontology proposed by Tansalarak and Claypool to provide a standard for defining components crosstalk different component models (COM, JavaBeans, COBRA) and unify the variances between them. A component is classified as either primitive (stand-alone component) or composite (constructed via connection-oriented or aggregation-based compositions). The feature XCM element defines how a component interacts with other components; it is a set of properties, methods and events. A property is the named attribute of a component that can be set/get by other components. A method encapsulates the behavior of a component that can be triggered by other components. An event is the message used by a component to communicate with other components. The design XCM element encapsulates the compositions of a set of pre-existing components (Tansalarak and Claypool, 2004). Arafa et al. also claim that information about components and the services they can provide can be formulated in dedicated ontologies. They see ontologies as a well-founded mechanism for representation and exchange of structured information. OWL-S is the ontology of services that supplies a core set of ontological concepts for describing the properties and compatibilities of Web services. OWL-S is designed to support automated service discovery, execution, interoperation composition and monitoring (Arafa et al., 2012). Andreou and Papatheocharous recommend the usage of Extended Backus-Naur Form (EBNF) based components proofing and an automatic search and retrieval mechanism that delivers the most suitable components. The latter is carried out in three steps: (1) Parsing the ontology profiles for requested and available components; (2) Executing the matching algorithm; (3) Recommending the closest match. The proposed framework consists of five layers: (1) The description layer is responsible for creating a profile that describes the component; (2) The location layer offers the means to search, locate and retrieve components; (3) The analysis layer provides the tools to evaluate the level of suitability; (4) The recommendation layer produces suggestions on the candidate components; (5) The build layer compromises a set of integration and customization tools for combining components. Component profiles are stored as instances of an ontology and matching between components takes place at the level of ontology items. The authors acknowledge the need for further research on their novel approach and highlight the need for a dedicated software tool that would support the whole framework as well as a graphical representation and visual comparison of ontology tree instances (Andreou and Papatheocharous, 2015).

Model-Driven Engineering (MDE)

According to Pahl, ontologies support a number of modeling tasks including domain modeling, architectural configuration, service and process interoperability. The author claims that logic-based ontology languages are suitable to enhance traditional modeling languages, thus enabling model-driven services. Based on that notion, the use of ontology-based semantic modeling to support model-driven architecting of service-based software systems is proposed (Pahl, 2007b). Hesse sees ontologies as reusable model components from which particular implementations can be derived for specific platforms, according to the specific specification and constraints of the project (Hesse, 2005). Similarly, Hou states that models are central development artefacts from which code and other artefacts can be generated through model transformations. Transformability between models means that they can be translated into equivalent executable code. Author presents a model mapping approach using ontologies based on semantic consistency which can be used to build mapping relations between source and target models (Hou, 2010).

Evermann and Wand created a mapping between ontological concepts and object-oriented constructs in order to assign business meaning to object-oriented languages. Accordingly, they argue that object-oriented modeling languages (especially UML) can be used for conceptual modeling, thus bridging the gap between system analysis and design (Evermann and Wand, 2005). Ontology Definition Metamodel (ODM) is the Object Management Group (OMG) standard for integrating ontology languages into the software development process based on model-driven architectural principles. The ODM specifies model transformations using the Query/View/Transformations (QVT) language. By combining the QVT, ODM and RDF schemas, metamodels can be transformed to languages like UML, topic maps and entity-relationship (Gašević et al., 2009). Katanov and Palviainen interpret ODSE as an extension for MDE. They see ontologies as a resolution for loose connection between CIM and PIM encountered in MDE. A domain ontology in the place of CIM can be used to generate certain parts of PIM, resulting in automation even before the executable code generation (PSM). Their Smart Modeler enables ontology-based creation of model elements, discovery and reuse of software components and generation of executable programming code for models. Their future work will be directed to adding an opportunistic way of software composition and code generation for other programming languages (Katanov and Palviainen, 2010). Bartolo Espiritu et al. advocate software development through software architecture while using ontologies and MDE for specification and implementation. Since ontology enables specification of
software, it is used at the CIM stage to document, specify and communicate the architecture, as well as to analyze and evaluate it, providing the first internal architectural elements. Ontologies can then be mapped to PIM models, for example a UML class diagram. The advantages to be realized by such an approach include better architecture specification, improved definition of stages of design and implementation, better architecture documentation and automation of the architecture development process. The authors intend to develop an automatic intermediate step to map ontologies to UML class diagrams and create templates that would allow mapping from PIM to PSM models (Bartolo Espiritu et al., 2014). Zimmer and Rauschmayer demonstrated how better integration between modeling concepts and the source code can be achieved employing the tool named *Tuna*. The authors created a generic source code ontology and presented its instances as topic maps. Topic maps are an ISO standard for graph-based knowledge representation with three basic constructs: Topics, associations and occurrences. A code topic is a node containing the source code or an URL pointing to its resource. A code association links two code topics. There are two kinds of code associations, child and parameter. According to Zimmer and Rauschmayer, topic maps provided them with a foundation for flexible storage, representation and retrieval of source code, along with the seamless integration of non-code artifacts. They acknowledge the need to extend the tool so that it becomes a full-fledged environment for the ontology-based programming language (Zimmer and Rauschmayer, 2004). Shahzad et al. argue for model-based User Interface (UI) development using an ontological framework. The ontology called User Interface Ontology (UIO) defines the basic UI classes, properties and relationships. UIO and targeted domain mapping together constitute the base UI model used to generate a Graphical User Interface (GUI). The mapping is performed in two steps. In the first step, UIO data modeling classes are associated with the domain ontology properties to provide a structure for visualization and architecture. In the second step, UIO user interaction and graphical properties are added to the mapping. The UIO engineering and mapping can be applied to any domain ontology. Authors note that a functional ontology for domain ontology and UIO (user actions and events) needs to be further discussed (Shahzad et al. 2011).

Generative Programming (GP)

Bures et al. introduced an extension to generative programming application and solution space. They refined the application space into the problem specification domain and the background theory domain. The former domain contains concepts used to formulate the high-level specification, while the latter contains concepts not expressible in problem specification but required to formulate constraints. The solution space was refined to five domains: Intermediate language domain, target language domain, algorithm domain, search control domain and meta-programming kernel. The elements of the first three domains are clear from the respective domain names. The search control domain contains concepts used to control the search for applicable schemas, while the meta-programming kernel is comprised of concepts expressing operation on objects defined in other domains and used to implement schemas. Ontologies are used to classify schemas and enrich the engine. Among the advantages of using ontologies mentioned by the authors are their ability to act as documentation for programmers, make writing schemas easier, control the schema interaction, facilitate extensions and validate the output (Bures et al., 2004). In their ODASE platform, Bossche et al. present a process for transferring knowledge from the ontology to the programming language by automatic source code generation. In their case study they used *Mercury* (a strongly typed programming language) and *Hedwig* (a set of tools and libraries used to integrate an OWL ontology into an application) (Bossche et al., 2007).

Damaševičius et al. established that, due to the growing complexity caused by technology capabilities, market demands and user requirements, it is no longer sufficient to rely on content-based and feature-centric analysis and development of SPLs. They introduce enriched feature diagrams that use lightweight domain ontologies and extend feature diagrams notations. The proposed model can be transformed into generative component specifications using meta-programming techniques. They define two transformation levels. At level one, the enriched feature diagrams are transformed to a meta-program model. At level two, the meta-program model is transformed into a meta-program. The approach results in the creation of generative components for specifying families of domain systems (Damaševičius et al., 2008). Goldman demonstrated a technique to support implementation of ontology-specific application by automating the generation of ontology-specific class libraries. According to Goldman, a well written ontology already contains a declarative representation of knowledge needed to construct such a library. He suggests that the generated library should comprise both a class and an interface for each ontology class, thus mirroring between the subclass ontology relationship and inherits object-oriented relationship between interfaces. Interfaces are required since they provide multiple inheritance that is not supported by classes in most object-oriented languages (Goldman, 2003). Wiebe and Chan developed *Specification
Ontology to Software (SOS) as a solution to common SE problems (complexity, high development and maintenance costs). SOS implies the usage of ODSE that integrates the transition from design to implementation. Authors argue that the design ontology which describes the real world should be differentiated from the specification ontology that describes the software to be implemented. The main goal of the proposed approach is development process automation. By mapping the software specification in an ontology, a semantic agent could be used to find a component that fulfills the requirements. According to the Wiebe and Chan, it is possible to extend the SOS system in many ways and create other types of software based on different ontologies (Wiebe and Chan, 2012). Djuric and Devedzic showed how metaprogramming can be used to incorporate ontology modeling into a Java based programming environment as an embedded DSL for modeling business domains. Their approach relies on 

**Clojure**, a language for **Java Virtual Machine (JVM)** with advanced metaprogramming support. It blends ontologies with functional and object-oriented paradigms for the development of business domain models. Authors state that a programming language should natively support ontologies as a natural means for business domain modeling and domain-driven programming. The DSL which they utilized to incorporate ontologies with the **Clojure** programming language is called **Magic Potion**. It enables developers to use DL abstractions like concepts, properties, roles and restrictions as if they were parts of the programming language. It provides a way to create both a semantically rich domain model and executable code at the same time. The DSL also supports the definition of standard feature relationships (mandatory, optional, alternative, additional) and additional constraints (Djuric and Devedzic 2012).

**Discussion**

Ontologies are commonly recognized as a convenient way to describe and organize domain knowledge. As demonstrated in (Musen, 2000; Robal et al., 2015; Sim and Brouse, 2015; Shunxin and Leijun, 2010; Bossche et al., 2007; Gašević et al., 2009), using an ontology in DE can undoubtedly help to avoid misunderstanding between different parties (e.g., users and developers). In addition, we are in favor of the ODASE’s initiative, proposed in (Bossche et al., 2007), concerning the involvement of Business representatives in modeling and development of domain ontologies. We believe that such an approach can be helpful in bridging the gap between Business and IT, enabling a more realistic estimate of the required time and cost to finish the project and reusing the developed domain ontology repeatedly across the software’s lifecycle. We agree that ontologies are suitable for specification of requirements in an evolutionary approach, as stated in (Happel and Seedorf, 2006; Shunxin and Leijun, 2010; Karatas et al., 2014; Siegemund et al., 2011). Requirements that are specified using ontologies are suited for inheritance, extensibility, share and reuse. They can also be used as a communication tool between different stakeholders. Ontologically defined requirements can be checked for completeness and consistency and, owing to collaboration between IT and Business, possible problems can be detected earlier during requirements analysis.

Similar to (Karatas et al., 2014; Asikainen et al., 2007), we favor ontology modeling over feature modeling and believe that ontologies expressivity can be efficiently used for specifying features. We argue that ontologies may be used for feature specification, without the need of mapping them to feature diagrams. If features are specified in an ontology, then a feature diagram can be easily generated from it. In addition, a feature specification ontology can be mapped to component specification ontology, thus providing glue for the system’s components and a specification for a partial or full software generation. Ontologies can also be successfully applied in components definition and variability management. We support the approaches presented in (Asikainen et al., 2007; Duran-Limon et al., 2015; Lepuschitz et al., 2011), while being aware that more research and development is required in that respect. The possibility of generating a DSL grammar from a target ontology, as proposed in (Ceh et al., 2011; MezhuyeV, 2014), is also a very interesting topic. In our opinion, the success of such an approach largely depends on tool support. We believe that the viability of the development of such a tool is questionable but worth researching. The tool support is of the utmost importance for the transition from academic proposals to practical usage. Testing in real software development contexts is required in order to fully demonstrate the applicability of the proposed solutions.

The authors of (Andreu and Papatheocharous, 2015; Arafa et al., 2012; Nianfang et al., 2010; Happel and Seedorf. 2006) have noted that ontologies are well suited for the definition and decryption of system’s components. We agree with that notion and believe that formalized ontological knowledge can be very useful in component searching, matching and building systems based on components. Ontologies provide an efficient method for describing components, which is required in order to efficiently reuse them and achieve a high level realization of CBD. Ontologies describing components are also suited for rapid application development and incremental approach in development. Among successful ontology-based approaches that have successfully penetrated into SOA is OWL-S, which uses ontologies to describe components and services they provide. Since
using semantic markup for Web services has already been extensively explored, we argue that ontologies can also be utilized to describe components of GUI-based desktop, Web or mobile applications (which are not necessarily service-oriented). The ontological knowledge of the available components can be used to automate the development of component-based applications. Knowledge about existing components and their relationships can be applied to facilitate documentation management and software maintenance. Automation can be achieved by combining a set of existing components based on a feature specification. As already mentioned, a feature specification knowledge can also be represented in an ontology and additional ontologies can be employed to capture other domain-specific knowledge.

We argue that several ontologies can be used to enrich the process of automated component-based application development and enable higher abstraction levels for developers. However, a high level dedicated tool support is required to enable such automation. Thus, we recommend that more effort is invested in research and development of tools for component-based composition of ontologically described components.

Owing to the establishment of MDE as the OMG’s standard and its growing popularity over the last decade, a significant amount of research has been dedicated to the usage of ontologies for model transformation. Although we concede that there are plausible arguments for integrating ontologies into MDE processes, we do not share the view that ODSE is an extension of MDE, as ODSE can also be successfully integrated with other SE paradigms. We support the approach presented in (Katasonov and Palviainen, 2010; Bartolo Espiritu et al., 2014) for using ontologies in the CIM stage. Ontologies can help automate the transformation from CIM to PIM, leading to an even higher abstraction level in MDE. Ontologies can also be used to assign business meaning to object-oriented models or languages. We find the association of UI models with domain ontologies for GUI generation proposed in (Shahzad et al., 2011) to be interesting. Since UI elements can also be considered system components, we believe that ontologies describing them can be efficiently used in the GUI generation process. We advocate for research on automated systems development, where both the internal system components and GUI components are ontologically defined.

We highly support the ideas put forth in (Bossche et al., 2007; Damaševičius et al., 2008; Goldman, 2003; Wiebe and Chan, 2012) regarding transferring knowledge from the ontology to the programming language by using generative and metaprogramming techniques. We also agree with the notion expressed in (Goldman, 2003) that a well written ontology can be used to generate the source code, without the need for an additional representation layer between them. However, more research should be conducted to test the usability of the additional layer approach for complex projects, as proposed in (Damaševičius et al., 2008). Native support for ontologies in programming languages, presented in (Djuric and Devedzic, 2012), is another idea we are very much in favor of. Unfortunately, presently we must rely on external libraries for reading and creating ontologies in popular object-oriented languages like C# and Java. The approach introduced in (Wiebe and Chan, 2012), which includes the use of agents to find and retrieve the component that fulfills the requirements based on its ontological description, also deserves interest. We believe that the proposed approaches show that GP techniques and ontologies can be successfully coupled to provide higher level knowledge-based software automation. We therefore contend that further research on a possible synthesis of ontologies, DE, RE, CBD and GP approaches should be undertaken to enable higher level knowledge-based automation for SPLs.

We fully support the authors’ perspective in (Happel and Seedorf 2006) that the currently mostly academic integration of ontologies into SE should be implemented in practice and tested on large scale SE projects. Software engineers need specific methodologies, guides and tool support on how they can apply (and make use of) ontologies in their process. As pointed out in (Hesse, 2005), ontologies are a promising instrument for transferring knowledge from one project to another and from one development cycle to the next. ODSE might indeed become a paradigm enabling more compatible models, more reusable components and lower costs in software development. We agree with another important notion from (Hesse, 2005), which states that ontologies can facilitate software development processes, but only in the long term. It is not realistic to expect results to be yielded from ODSE without significant investment in ontologies development and their integration with current SE development paradigms and their processes. As the use of ontologies requires additional modeling efforts, authors of (Happel and Seedorf, 2006) warn that savings must be made in other places. A significant level of ontological knowledge reusability is thus required across the whole SE lifecycle to achieve cost effectiveness in ODSE.

**Conclusion**

In the conducted literature review we discussed several approaches to using ontologies to increase the reuse of software or its artifacts. While the specific aim of reusing software is to enhance its productivity and quality, the ultimate goal of software engineers is to successfully manage and control complexity and to reduce the development and maintenance costs. Similarly to models in MDE, in ODSE ontologies are
reusable artifacts used throughout the software lifecycle. Our literature review summarizes the proposed solutions for using ontologies in SE with reference to four SE paradigms they belong to: SPLs, CBD, GP and MDE, with their respective processes or sub-paradigms like DE, AE and RE. As ontologies are commonly utilized to describe domain knowledge, we found that most of the research on using ontologies in SPLs focuses on DE and RE. In the DE domain, several approaches were found that use ontologies to provide a common vocabulary and avoid misunderstanding between different parties. In the RE domain, we also encountered several examples of using ontologies in specifying system’s requirements to make them extensible, sharable and reusable. Reasoning over requirements specification and early problem detection were also presented. Although in our review we established that ontologies are highly suitable for feature specification, tools enabling ontology-based feature specification still need to be developed. As with most newly proposed solutions, SE engineers need the adequate tool support in order to start implementing them in practice. Most of the research on using ontologies in CBD has focused on Web services, as SOA is presently a highly popular SE paradigm and many large systems are currently in the process of shifting their architecture to it. In that respect, ontologies are used to describe components and services they provide in order to enable their searching, matching and building service-based systems. However, ontologies may also be applied to not necessarily service-oriented, GUI-based Web, desktop and mobile applications to describe their components and enable automatic construction of new systems based on existing components. Our research review also indicated that ontologies are very useful in model transformations, which has led to their successful integration into MDE processes. Most of the proposed solutions regarding MDE use ontologies in the CIM stage and are aimed at automating the models transformation from CIM to PIM. In the GP paradigm, ontologies are used as specifications for generators that produce the source code. A well written ontology can be directly used in the generation process or an additional layer can be introduced between the ontology and source code. More research is recommended in this field since the integration of ontologies describing domains and system’s components by employing GP techniques can bring higher abstraction levels and knowledge-based software automation suitable for the development of SPLs. With adequate investments in their development and integration, ontologies can enrich software development processes in the long term. However, a high level of ontological knowledge reusability is required for this investment to pay off. It is our opinion that ontologies will strongly penetrate the SE discipline in the next few years. This prediction is based on the fact that ontologies are a means of successfully describing knowledge, which, along with investments into technology, is one of the critical factors for achieving growth.

Future work

In our future work we will focus on investigating the areas that were identified as interesting and insufficiently researched in our literature review. We intend to develop ontologies for feature and components specification. Those two ontologies will be combined together to enable automatic code generation. The introduction of domain-specific ontologies into the generation process will also be explored. We also plan to implement GP techniques to ontological knowledge in generating both the GUI and the backend code for Web, desktop or mobile applications. Finally, several case studies need to be conducted to determine the effectiveness of the proposed concept in different types of applications and domains.

Funding Information

This work has been supported in part by the Croatian Science Foundation under the project number 8537.

Author’s Contributions

Daniel Strmečki: The main responsible author for conducting the literature review. Contributed in preparation, reading and writing.


Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References


