ODYSSEY: Software Development Life Cycle Ontology

J. I. Olszewska and I. K. Allison

School of Computing and Engineering, University of West Scotland, U.K.

Keywords: Software Engineering, Knowledge Engineering, Domain Ontology, Domain Analysis and Modeling, Decision Support System, Intelligent Agents, Autonomous Systems.

Abstract: With the omnipresence of softwares in our Society from Information Technology (IT) services to autonomous agents, their systematic and efficient development is crucial for software developers. Hence, in this paper, we present an approach to assist intelligent agents (IA), whatever human beings or artificial systems, in theirs task to develop and configure softwares. The proposed method is an ontological, developer-centred approach aiding a software developer in decision making and interoperable information sharing through the use of the ODYSSEY ontology we developed for the software development life cycle (SDLC) domain. This ODYSSEY ontology has been designed following the Enterprise Ontology (EO) methodology and coded in Descriptive Logic (DL). Its implementation in OWL has been evaluated for case studies, showing promising results.

1 INTRODUCTION

Software Development Life Cycles (SDLCs) have been introduced in order to design and develop softwares in a coherent, consistent, and efficient way as recommended in (ISO/IEC/IEEE 15288 International Standard, 2015) and (ISO/IEC/IEEE 12207 International Standard, 2017).

SDLCs are mainly task-oriented processes. In particular, they adopt a task decomposition approach (Dix et al., 2004), i.e. they describe how the task to develop a software is split into sub-tasks and in which order these sub-tasks have to be performed.

On the other hand, software development has been barely studied from the developers’ perspective (Roehm et al., 2012). Hence, with the raise of intelligent and autonomous agents in today’s applications, we suggest to assist the developers with the software development task by adopting a software development approach based on developer experience or more generally on intelligent agent experience we called AX.

Thenceforth, we propose to study the SDLCs from a developer’s knowledge point of view and thus to perform the SDLCs’ task analysis by applying both knowledge-based technique and entity-relation based analysis. Indeed, the knowledge-based technique (Dix et al., 2004) looks at what users or developers need to know about the objects and actions involved in a task and how that knowledge is organized. On the other hand, the entity-relation-based analysis (Dix et al., 2004) identifies actors (e.g. developers), objects and the relationships between them as well as the action they perform.

For this purpose, we adopted an ontological approach, since an ontology is an artificial-intelligence method (Davies et al., 2003) which provides an explicit specification of a conceptualization (Gruber, 1995) and encompasses both knowledge concepts of the analysed domain (i.e. classes such as actors, actions, objects) and the relations between them (Guarino, 1998).

Ontologies have been used for various domains, from intelligent vision systems (Olszewska, 2011), (Olszewska, 2012), to autonomous and robotic systems (Olszewska et al., 2017), (Fiorini et al., 2017), as ontologies are a convenient mode to share common knowledge between various agents in an interoperable way (Bayat et al., 2016).

The few ontologies developed for the software engineering domain (Bermejo-Alonso, 2006) aim to primarily contribute to the Model-Driven Software Development (MDSD) (Leonard et al., 2017) or Model-Driven Architecture (MDA) and other software development processes such as meta-models involving, e.g. the Unified Model Language (UML) and/or the Business Process Model and Notation (BPMN) (Olszewska et al., 2014), resulting in the Ontology-Driven Software Development (ODSD) (Pan et al., 2012). Hence, the ODSD area focuses, despite its
name, on the MDE analysis of a specific project (Ios
tani et al., 2015) or a specific service (Knublauch,
2004) rather than on any SDLC process itself.

On the other hand, some ontologies have been
produced to serve as specific Computer-Aided Soft-
ware Engineering (CASE) tools for software quality
check, such as the Ontology-based Development En-
vironment (ODE) (Falbo et al., 2003), for software
requirement capture like the Ontology-driven Requir
iments Analysis Tool (OntoRAT) (Al-Hroub et al.,
2009), or for software requirements traceability, e.g.
through the Marrying Ontology and Software Techno-
logy (MOST) project product (Pan et al., 2012). Ho-
wever, these ontologies are not dedicated to the full
software development life cycle modelling.

Hence, in this paper, we propose to develop an
ontology for the Software Development Life Cycle
(SDLC) domain.

This domain ontology for software development life
cycles we called ODYSSEY aims in first instance
to capture the knowledge included in the SDLCs and
then, to formalize and implement the SDLC major
concepts and their properties.

The proposed ontology reflects a developer-
centric approach for software development life cy-
cles; the developer being an Intelligent Agent (IA),
whatever a human or an artificial one, i.e. an agent ca-
ble of knowing and acting as well as capable of em-
ploying its knowledge in its actions (Bermejo-Alonso,
2006). Furthermore, ODYSSEY ontology targets the
software SDLC methodologies conceivably followed
by developers rather than the possibly created soft-
ware artifacts.

The contributions of this paper are twofold. On
one hand, as far as we know, ODYSSEY is the
first ontology for the software development life cy-
cle domain. On the other hand, ODYSSEY ontology
is the basis for software development based on intel-
ligent agent experience (AX).

The paper is structured as follows. In Section 2,
we present our ontology domain, i.e. the software de-
velopment life cycles (SDLC), while the domain on-
tology itself (ODYSSEY) is described and evaluated
in Section 3. Conclusions are drawn up in Section 4.

2 PRELIMINARIES: SOFTWARE
DEVELOPMENT LIFE CYCLES

Existing Software Development Life Cycles (SDLCs)
(Sommerville, 2015) could be categorized as Plan-
Driven Life Cycles Models or as Agile Development
Models.

Plan-driven life cycle models are characterized by
sequences of steps to undertake the solution formali-
sation, product(s) development, solution delivery, and
solution support. These software development met-
ods include SDLCS such as the Waterfall model, the
Prototyping model, the V-Model, and the Spiral mo-
del.

On the other hand, the Agile Development Models
are underpinned by the Agile manifesto\(^1\) and mainly
consist in iteratively repeating three major phases, i.e.
the implementation phase, the delivery and feedback
phase, and the next-iteration planning phase. Well-
established Agile SDLCs are the Rapid Application
Development (RAD), the Dynamic System Develop-
ment Method (DSDM), Extreme Programming (XP),
and SCRUM.

Among all these SDLCs, the first SDLC which has
been used in a more systematical way by software
developers is called Waterfall and was proposed by
(Royce, 1970). It is basically a linear, sequential mo-
del where each phase ‘is dependent of’ the previous
one, i.e. must be completed before the following one
can be started, leading to a ‘cascade’ effect, whence
a ‘waterfall’ model. This model has been extensively
applied to large engineering systems, with all the pro-
cess activities planned and scheduled before the start
of the software development, usually spanning over a
time scale of twelve months.

Originally, the Waterfall model was constituted by
seven, consecutive stages, namely, system require-
ments, software requirements, analysis, program de-
sign, coding, testing, and operations as presented in
Fig. 1.

The first stage consists in specifying the system
requirements, i.e. the description of what the end sys-
tem is expected to provide to the customer/user. That
includes the particular functions the system must per-
form as well as the information about the environment
in which the final product will operate. This descrip-
tion is usually recorded in the client’s language.

Thence, in the second stage, the software devel-
oper/designer expresses the software requirements in a
language suitable for potential implementation.

The next stage focuses on the analysis of how the
system will produce the intended output(s). That le-
ads to the architectural design of the system and in-
volves a high-level decomposition of the system into
components or software units. This implies not only
the functional decomposition of the system to deter-
mine which component will provide which services,
but also the characterization of the components’ inter-
dependencies and the resources’ sharing between the
components. It is worth noting that the components
\(^1\)http://agilemanifesto.org/
can either be imported from existing available solutions or be independently developed from scratch.

The next stage concentrates on the program design. This detailed design is a refinement of the component description provided by the architectural design. Whilst the program design aims first to address the behavioural constraints of higher-level description, it is even a better product when it also satisfies non-functional requirements of the system such as efficiency, reliability, security, etc. Thus, the language used for the detailed design must allow some analysis of the design in order to assess these systems’ properties. At this stage, it is also important to keep track of the considered design options and the justifications of the choices which have been made.

Then, the fifth stage is the implementation of the detailed design in some executable programming language.

After this coding stage, the next one consists in testing each software component to verify it performs correctly according to some test criteria. Once all the components have been tested individually, they are integrated as per architectural design. Further testing is done to ensure both the dependable behaviour of the entire system and the acceptable use of any shared resources. It is also possible at this time to perform some acceptance testing with the clients to check that the system meets their expectations. It is only after acceptance of the integrated system that the product could be released to the customer. However, in the meantime, it may also be necessary to certify the final system according to international standards and/or further requirements imposed by some authority.

The final stage or operation stage deals mainly with the system maintenance until a new version of the product is developed or the product is phased out. Consequently, the last stage is the longest one in terms of duration (Dix et al., 2004).

The waterfall model presents the advantages of being a simple linear process that addresses software quality management and project management, and that tries to eliminate as many problems as possible in each phase. However, one of the main flaws of this sequential approach is that the idea of iteration has not been embedded in the original waterfall’s model (Dix et al., 2004).

Indeed, on one hand, in case of rapidly changing businesses’ needs and operating environment, which lead to important changes of the requirements over time, freezing the system and software requirements for months or years, while completing the design and implementation, could be not efficient. On the other hand, if a requirement is identified in the implementation and unit testing phase as too expensive to be implemented, this requires the update of the overall requirement document in order to remove that requirement, i.e. the rework of the first phase, and possibly of the system and software design phase as well. Consequently, that drains resources and stirs up delays in the overall development process. The early freeze of the software specifications to avoid any change to it could be a short-term solution, but such freeze could be considered later as premature and be not effective, since problems will then have to be overcome by implementation tricks (Sommerville, 2015). Hence, the need to rework on a software system when changes are made to the requirements for whatever reason implies the waterfall model is only appropriate for some types of systems such as embedded systems, critical systems, or large software systems (Sommerville, 2015).

As in practice, a SDLC has to inherently handle change along the software development, some further variations of the Waterfall model have been presented in the literature, with some versions incorporating some levels of iterations (Sommerville, 2015). Thence, the Waterfall SDLC could also be modeled in terms of five fundamental software development activities, where the outputs from one activity are inputs for the next one, while the last activity’s feedback informs all the previous ones (Fig. 2).

More specifically, the first phase or requirements analysis establishes the system’s services, constraints, and goals through consultations between the software developer and the system user(s), leading to the detailed system specification.

Secondly, the system and software design phase defines the software system architecture, identifying and describing the fundamental abstractions and their relationships of the overall system as well as allocating the requirements to the corresponding subsystems.

During the third phase called implementation and unit testing, the software code is produced and results in a set of programs or program units which are indi-
3 PROPOSED ONTOLOGY

To develop the ODYSSEY ontology, we followed an ontological development life cycle (Gomez-Perez et al., 2004) based on the Enterprise Ontology (EO) Methodology (Dietz, 2006), since EO is well suited for software engineering applications (van Kervel et al., 2012).

The adopted ontological development methodology consists of four main phases covering the whole development cycle as follows:

1. identifications of the purpose of the ontology (Section 3.1);
2. ontology building which consists of three parts: the capture to identify the domain concepts and their relations; the coding to represent the ontology in a formal language; and the integration to share ontology knowledge (Section 3.2);
3. evaluation of the ontology to check that the developed ontology meets the scope of the project (Section 3.3);
4. documentation of the ontology (Section 3.4).

3.1 Ontology Purpose

The ODYSSEY ontology domain has been defined by the software development life cycles (SDLCs) presented in Section 2, while the scope of this ODYSSEY domain ontology is to assist intelligent agent(s) when using a SDLC to develop a software.

Indeed, it has been found that, on one hand, human software developers spent most of their time in activities such as information seeking (Ponzanelli et al., 2017) or interacting (Ciccozzi et al., 2017). On the other hand, it has been recently identified that the production of softwares for robotic systems (Ciccozzi et al., 2017) requires, among other, subsystems interoperability and human-robot synergy such as Human-Robot Interactions (HRI) (Calzado et al., 2018) or Human-Swarm Teaming (HST) (Kolling et al., 2016).

Since ontologies intrinsically allow to elucidate concepts, to share information in an interoperable way, and to perform automated reasoning, the ODYSSEY ontology purpose is to contribute to (i) support a human developer in decision making about SDLCs; (ii) help human developers in collaborating on team’s software development; (iii) guide autonomous system(s) in reconfiguring their softwares; (iv) aid collaborative mixed human-robot teams in interacting, synchronising or configuring software agents.
3.2 Ontology Building

The knowledge capture consists in the identification of concepts and relations of the SDLC domain described in Section 2. The knowledge coding is done in Descriptive Logic (DL). In particular, the concept of SDLC “waterfall model” (Fig. 1) is defined in DL as follows:

\[
\text{Waterfall Model} \subseteq \text{Model}
\]
\[
\forall \exists \text{Phase} . \langle P_1 \rightarrow \text{System}\_\text{Requirements} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_2 \rightarrow \text{System}\_\text{Requirements} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_3 \rightarrow \text{Analysis} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_4 \rightarrow \text{Design} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_5 \rightarrow \text{Implementation and Test} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_6 \rightarrow \text{Maintenance} \rangle
\]

Furthermore, the original waterfall model could be formalised in temporal DL as follows:

\[
\text{Waterfall1} \subseteq \text{Waterfall Model}
\]
\[
\forall \exists \text{Phase} . \langle P_1 \rightarrow \text{System}\_\text{Requirements} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_2 \rightarrow \text{System}\_\text{Requirements} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_3 \rightarrow \text{Analysis} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_4 \rightarrow \text{Design} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_5 \rightarrow \text{Implementation and Test} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_6 \rightarrow \text{Maintenance} \rangle
\]

with \( k = 7 \), the number of phases of the SDLC waterfall model, and \( \text{meet} \), the temporal relation defined in temporal DL as introduced by (Olszewska, 2016):

\[
P_m \hspace{0.3cm} P_f \hspace{0.3cm} \text{meet} \hspace{0.3cm} P_{m+1} \sqsubseteq \text{Temporal Relation}
\]
\[
\forall \exists \text{Phase} . \langle P_1 \rightarrow \text{System}\_\text{Requirements} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_2 \rightarrow \text{System}\_\text{Requirements} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_3 \rightarrow \text{Analysis} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_4 \rightarrow \text{Design} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_5 \rightarrow \text{Implementation and Test} \rangle
\]
\[
\forall \exists \text{Phase} . \langle P_6 \rightarrow \text{Maintenance} \rangle
\]

where the temporal DL symbol \( \diamond \) represents the temporal existential qualifier, and where a time interval is an ordered set of points \( T = \{ t \} \) defined by endpoints \( t^- \) and \( t^+ \), such as \((t^- < t^+) \land (t^+ < t^+)\).

The enhanced waterfall model (Fig. 2) could be then represented in DL as follows:

\[
\text{Enhanced Waterfall Model} \subseteq \text{Model}
\]
\[
\forall \exists \text{Phase} . \langle S_1 \rightarrow \text{Development and Test} \rangle
\]
\[
\forall \exists \text{Phase} . \langle S_2 \rightarrow \text{System}\_\text{Design} \rangle
\]
\[
\forall \exists \text{Phase} . \langle S_3 \rightarrow \text{Integration and Test} \rangle
\]
\[
\forall \exists \text{Phase} . \langle S_4 \rightarrow \text{Operation and Maintenance} \rangle
\]

with \( S_1 = P_1 \sqcup P_2 \), \( S_2 = P_3 \sqcup P_4 \), \( S_3 = P_5 \), \( S_4 = P_6 \), and \( S_5 = P_7 \).

3.3 Ontology Evaluation

The ontology evaluation ensures that the developed ontology meets all the requirements (Dobson et al., 2005).
Figure 5: Excerpt of the scenario implementation, with the instantiation waterfall1 of the ‘waterfall model’ concept as well as the asserted and inferred, related properties.

Figure 6: Excerpt of the scenario implementation, with the instantiation waterfall2 of the ‘enhanced waterfall model’ concept as well as the asserted and inferred, related properties.

Figure 7: Some samples of query results in context of the proposed test scenario.
For this purpose, experiments have been carried out using Protege v5.2.0 IDE and applying HermiT reasoner v1.3.8.413. In the experiment scenario, an agent 1 which is a human being adopts the waterfall model (instantiated as Waterfall1) to develop a project 1, while an agent 2 which is an artificial system uses the enhanced waterfall model (instantiated as Waterfall2) to work on the same project, as illustrated in Figs. 5-6, respectively. A sample of queries for this scenario is presented in Fig. 7.

In this case, the agent 1 could query the system, on one hand, to receive guidance about the chosen SDLC’s steps to follow (e.g. waterfall 1) and, on the other hand, to further understand/synchronize with another agent like the agent 2 which is working on the same project, while having adopted a similar but different SDLC (e.g. waterfall 2).

In these experiments, ODYSSEY has provided 100% correct answers and no inconsistency has been reported.

Moreover, the evaluation of ODYSSEY uses metrics presented in (Tartir et al., 2018) and (Hlomani and Stacey, 2014). The extracted values by Protege are presented in Fig. 8. We can observe that the DL expressivity is $SHOIF(D)$. It is worth noting the reasoner works under the open-world assumption, i.e. if for a question, there is no information in the ontology, then the answer of the system is ‘does not know’, and not ‘does not exist’. To obtain the latter one, information should be explicitly provided, but adding all these closure-type assertions can slow down the reasoner. So, in practice, a trade-off should be achieved between computational efficiency and completeness. Actually, ODYSSEY performs in real time and contains 186 axioms for 25 classes, leading to its high richness.

Furthermore, ODYSSEY cohesion could be assessed using the number of root classes which is equal to 1, the number of leaf classes which is equal to 17, and the average depth which is equal to 4.

All these metrics indicate ODYSSEY shows promising performance and could efficiently serve as the basis for further extensions to include even more SDLC model knowledge.

### 3.4 Ontology Documentation

The ODYSSEY ontology has been documented in Section 3. It is a middle-out, domain ontology which has been developed from scratch using non-ontological resources such as primary sources, e.g. (Royce, 1970). ODYSSEY is not dependent of any particular software/system/service/project, but it focuses rather on the knowledge included in the main SDLCs. It is worth noting the ODYSSEY has not reuse any existing ontology, since it is the first ontology in its kind for the SDLC domain. However, the ODYSSEY ontology could be reused itself in the future, since on one hand, more SDLC models (Abrahamsson et al., 2003) could be captured and formalised. On the other hand, the formalization of the potential changes which might occur through the software development process (Allison and Merali, 2007) could be added as well to expand the present ODYSSEY ontology and lead to a dynamic ontology.

### 4 CONCLUSIONS

In this work, ODYSSEY ontology aids developers to choose and use SDLCs by expliciting the knowledge contained in the SDLCs and by allowing implicitly a flexible use of the SLDCs. On the other hand, ODYSSEY allows an increased interoperability between humans and/or autonomous systems, when developing a software or reconfiguring it, and thus constitutes a step further towards intelligent agents’ collaboration.

### REFERENCES


Bayat, B., Bermejo-Alonso, J., Carbonera, J. L., Facchinetti, T., Fiorini, S. R., Goncalves, P., Jorge, V., Ha-


