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

Olszewska, J. and Allison, I.
ODYSSEY: Software Development Life Cycle Ontology.
DOI: 10.5220/0006957703030311
In *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2018) - Volume 2: KEOD*, pages 303-311
ISBN: 978-989-758-330-8

name, on the MDE analysis of a specific project (Isotani 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 Software Engineering (CASE) tools for software quality check, such as the Ontology-based Development Environment (ODE) (Falbo et al., 2003), for software requirement capture like the Ontology-driven Requirements Analysis Tool (OntoRAT) (Al-Hroub et al., 2009), or for software requirements traceability, e.g. through the Marrying Ontology and Software Technology (MOST) project product (Pan et al., 2012). However, 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 cycles; the developer being an Intelligent Agent (IA), whatever a human or an artificial one, i.e. an agent capable of knowing and acting as well as capable of employing 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 software 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 cycle domain. On the other hand, ODYSSEY ontology is the basis for software development based on intelligent agent experience (AX).

The paper is structured as follows. In Section 2, we present our ontology domain, i.e. the software development life cycles (SDLC), while the domain ontology 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 formalisation, product(s) development, solution delivery, and solution support. These software development methods include SDLCS such as the Waterfall model, the Prototyping model, the V-Model, and the Spiral model.

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 Development 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 model 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 process 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 requirements*, *software requirements*, *analysis*, *program design*, *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 system is expected to provide to the customer/user. That includes the particular functions the system must perform as well as the information about the environment in which the final product will operate. This description is usually recorded in the client's language.

Thence, in the second stage, the software developer/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 leads to the architectural design of the system and involves a high-level decomposition of the system into components or software units. This implies not only the functional decomposition of the system to determine which component will provide which services, but also the characterization of the components' interdependencies and the resources' sharing between the components. It is worth noting that the components

[1]http://agilemanifesto.org/

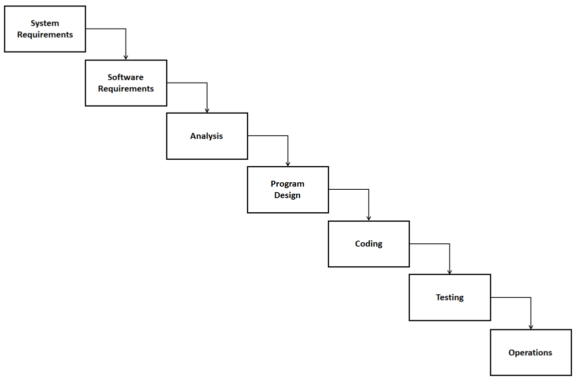


Figure 1: Overview of the original SDLC Waterfall model.

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-

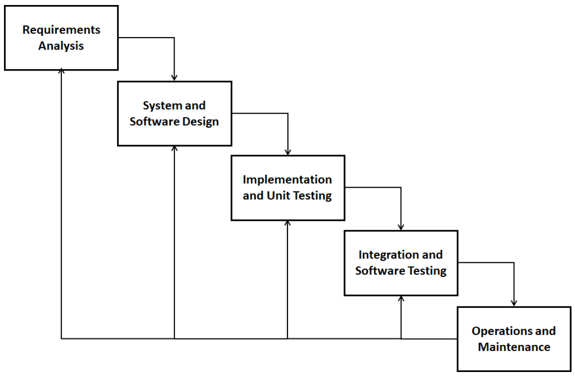

Requirements Analysis
System and Software Design
Implementation and Unit Testing
Integration and Software Testing
Operations and Maintenance

Figure 2: Overview of the enhanced SDLC Waterfall model.

vidually tested and verified to ensure each unit meets its specification.

The fourth phase is dedicated to the integration and system testing, i.e. the individual program units are integrated into the overall system and the overall system is tested to check that all the software requirements have been met, before delivering the software system to the customer.

Finally, the operations and maintenance phase aims to install the product in its work environment and to put it in practical use. In particular, maintenance, which is the longest phase of the waterfall SDLC, involves the correction of errors in the system which were not discovered in the earlier stages of the life cycle, i.e. before the release of the product. Maintenance also consists in improving the implementation of the software units and enhancing the system's services when new requirements are discovered. Therefore, maintenance provides feedback to all of the other activities in the life cycle (Fig. 2). For example, when a program error emerges, it implies a rework on the implementation and unit testing phase, whereas if a design errors appears, it involves repeating the system and software design phase. On the other hand, in case of omissions or when the need for a new functionality is identified, the requirement analysis phase is rerun to upgrade the product (Sommerville, 2015).

Hence, a SDLC such as the Waterfall approach could be modelled in various ways, leading among others to different names of the phases and to different sequences of its execution. Therefore, the ODYSSEY ontology as described in Section 3 aims to contribute to the elicitation of the SDLC intra-model knowledge as well as its interoperable sharing.

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 than 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:

$$\begin{aligned} Waterfall\_Model \sqsubseteq\ & Model \\ & \sqcap \exists hasPhase_{=\{P_1=System\_Requirement\}} \\ & \sqcap \exists hasPhase_{=\{P_2=Software\_Requirement\}} \\ & \sqcap \exists hasPhase_{=\{P_3=Analysis\}} \\ & \sqcap \exists hasPhase_{=\{P_4=Program\_Design\}} \\ & \sqcap \exists hasPhase_{=\{P_5=Coding\}} \\ & \sqcap \exists hasPhase_{=\{P_6=Testing\}} \\ & \sqcap \exists hasPhase_{=\{P_7=Operations\}}. \end{aligned} \tag{1}$$

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

$$\begin{aligned} Waterfall1 \sqsubseteq\ & Waterfall\_Model \\ & \sqcap (\diamond t_1 \ldots t_k) \\ & (P_1 m P_2) \ldots (P_{k-1} m P_k) \\ & \cdot (P_1 @ t_1 \sqcap \ldots \sqcap P_k @ t_k) \end{aligned} \tag{2}$$

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

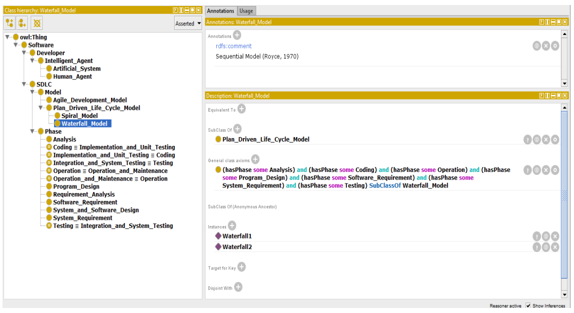
$$\begin{aligned} P_i m P_j \equiv meet(P_i @ t_i, P_j @ t_j) \sqsubseteq\ & Temporal\_Relation \\ & \sqcap (\diamond t_i)(\diamond t_j) \\ & (t_{i^+} = t_{j^-}) \\ & \cdot (P_i @ t_i \sqcap P_j @ t_j), \end{aligned} \tag{3}$$

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 end-points t^- and t^+, such as $(t^-, t^+) : (\forall t \in T)(t > t^-) \wedge (t < t^+)$.

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

$$\begin{aligned} Enhanced\_Waterfall\_Model \sqsubseteq\ & Model \\ & \sqcap \exists hasPhase_{=\{S_1=Requirement\_Analysis\}} \\ & \sqcap \exists hasPhase_{=\{S_2=System\_and\_Software\_Design\}} \\ & \sqcap \exists hasPhase_{=\{S_3=Implementation\_and\_Unit\_Testing\}} \\ & \sqcap \exists hasPhase_{=\{S_4=Integration\_and\_System\_Testing\}} \\ & \sqcap \exists hasPhase_{=\{S_5=Operation\_and\_Maintenance\}}, \end{aligned} \tag{4}$$

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



Class hierarchy: Waterfall_Model

Asserted

- owl:Thing
  - Software
    - Developer
      - Intelligent_Agent
        - Artificial_System
        - Human_Agent
    - SDLC
      - Model
        - Agile_Development_Model
        - Plan_Driven_Life_Cycle_Model
          - Spiral_Model
          - Waterfall_Model
      - Phase
        - Analysis
        - Coding ≡ Implementation_and_Unit_Testing
        - Implementation_and_Unit_Testing ≡ Coding
        - Integration_and_System_Testing ≡ Testing
        - Operation ≡ Operation_and_Maintenance
        - Operation_and_Maintenance ≡ Operation
        - Program_Design
        - Requirement_Analysis
        - Software_Requirement
        - System_and_Software_Design
        - System_Requirement
        - Testing ≡ Integration_and_System_Testing

Annotations | Usage

Annotations: Waterfall_Model

Annotations
rdfs:comment
Sequential Model (Royce, 1970)

Description: Waterfall_Model

Equivalent To

SubClass Of
Plan_Driven_Life_Cycle_Model

General class axioms
(hasPhase some Analysis) and (hasPhase some Coding) and (hasPhase some Operation) and (hasPhase some Program_Design) and (hasPhase some Software_Requirement) and (hasPhase some System_Requirement) and (hasPhase some Testing) SubClassOf Waterfall_Model

SubClass Of (Anonymous Ancestor)

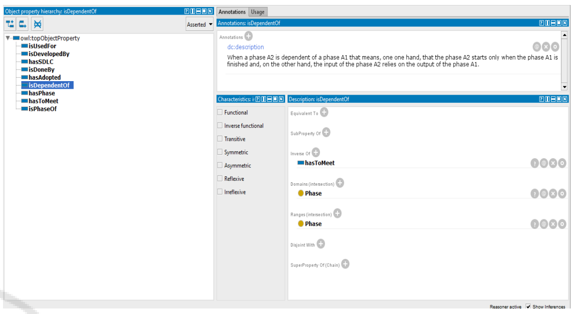
Instances
Waterfall1
Waterfall2

Target for Key

Disjoint With

Reasoner active ✓ Show Inferences

Figure 3: Main classes of the SDLC domain.



Object property hierarchy: isDependentOf

Asserted

- owl:topObjectProperty
  - isUsedFor
  - isDevelopedBy
  - hasSDLC
  - isDoneBy
  - hasAdopted
  - isDependentOf
  - hasPhase
  - hasToMeet
  - isPhaseOf

Annotations | Usage

Annotations: isDependentOf

Annotations
dc:description
When a phase A2 is dependent of a phase A1 that means, one one hand, that the phase A2 starts only when the phase A1 is finished and, on the other hand, the input of the phase A2 relies on the output of the phase A1.

Characteristics: i

☐ Functional
☐ Inverse functional
☐ Transitive
☐ Symmetric
☐ Asymmetric
☐ Reflexive
☐ Irreflexive

Description: isDependentOf

Equivalent To

SubProperty Of

Inverse Of
hasToMeet

Domains (intersection)
Phase

Ranges (intersection)
Phase

Disjoint With

SuperProperty Of (Chain)

Reasoner active ✓ Show Inferences

Figure 4: Main properties of the SDLC domain.

Moreover, the enhanced waterfall model could be expressed in temporal DL as follows:

$$\begin{aligned} Waterfall2 \sqsubseteq\ & Waterfall\_Model \\ & \sqcap (\diamond t_1 \ldots t_l) \\ & (S_1 m S_2) \ldots (S_{l-1} m S_l) \\ & (S_l m S_1) \ldots (S_l m S_{l-1}) \\ & \cdot (S_1 @ t_1 \sqcap \ldots \sqcap S_l @ t_l), \end{aligned} \tag{5}$$

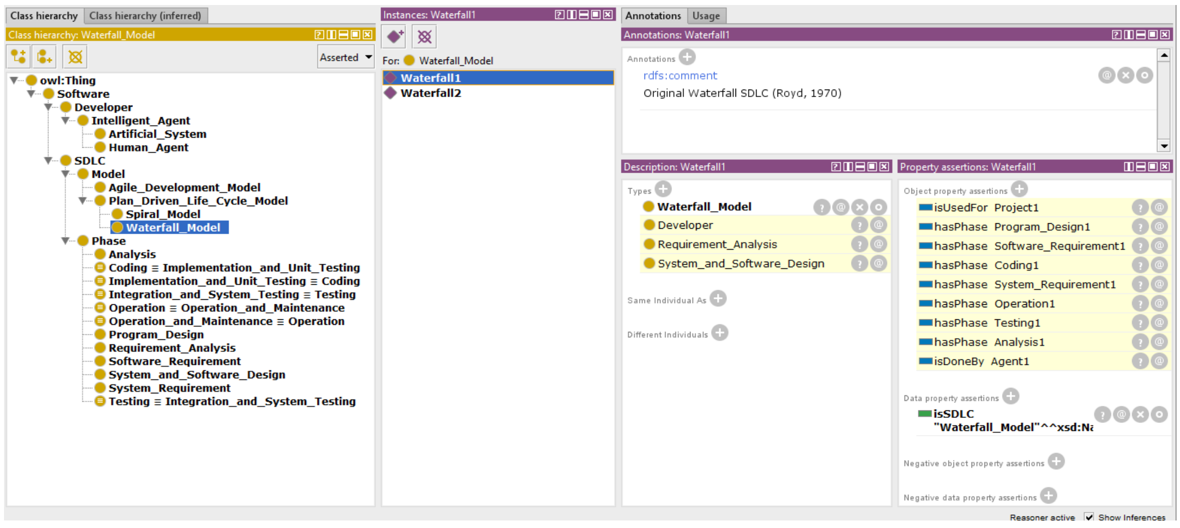
with $l = 5$, the number of stages of the SDLC enhanced waterfall model.

The ontology is implemented in the Web Ontology Language (OWL) language, using an appropriate tool like Protege v5.2.0 IDE and applying HermiT reasoner v1.3.8.413 (Glimm et al., 2014) to perform automated reasoning. An excerpt of the encoded concepts is presented in Fig. 3, while some properties are shown in Fig. 4. It its worth noting that the property *isDependentOf* could be represented in temporal DL as follows:

$$\begin{aligned} isDependentOf \sqsubseteq\ & Phase\_Property \\ & \sqcap (\diamond t_i \ldots t_j) \\ & (P_i m P_j) \cdot (P_i @ t_i \sqcap P_j @ t_j). \end{aligned} \tag{6}$$

3.3 Ontology Evaluation

The ontology evaluation ensures that the developed ontology meets all the requirements (Dobson et al., 2005).


Class hierarchy | Class hierarchy (inferred)

Class hierarchy: Waterfall_Model

Asserted

```
owl:Thing
  Software
    Developer
      Intelligent_Agent
        Artificial_System
        Human_Agent
    SDLC
      Model
        Agile_Development_Model
        Plan_Driven_Life_Cycle_Model
          Spiral_Model
          Waterfall_Model
      Phase
        Analysis
        Coding ≡ Implementation_and_Unit_Testing
        Implementation_and_Unit_Testing ≡ Coding
        Integration_and_System_Testing ≡ Testing
        Operation ≡ Operation_and_Maintenance
        Operation_and_Maintenance ≡ Operation
        Program_Design
        Requirement_Analysis
        Software_Requirement
        System_and_Software_Design
        System_Requirement
        Testing ≡ Integration_and_System_Testing
```

Instances: Waterfall1

For: Waterfall_Model

Waterfall1

Waterfall2

Annotations | Usage

Annotations: Waterfall1

Annotations

rdfs:comment

Original Waterfall SDLC (Royd, 1970)

Description: Waterfall1

Types

Waterfall_Model

Developer

Requirement_Analysis

System_and_Software_Design

Same Individual As

Different Individuals

Property assertions: Waterfall1

Object property assertions

isUsedFor Project1

hasPhase Program_Design1

hasPhase Software_Requirement1

hasPhase Coding1

hasPhase System_Requirement1

hasPhase Operation1

hasPhase Testing1

hasPhase Analysis1

isDoneBy Agent1

Data property assertions

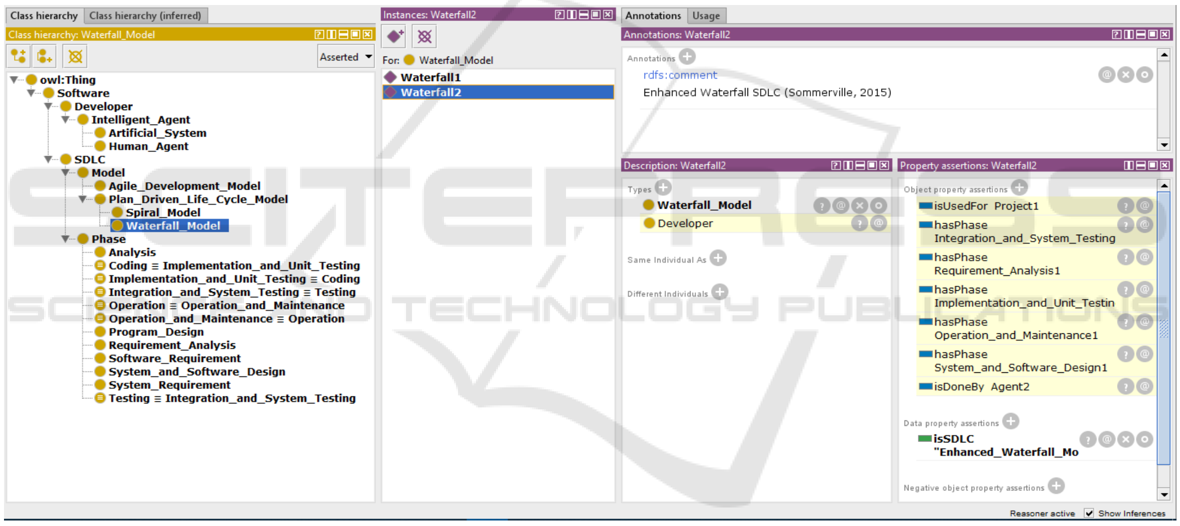
isSDLC "Waterfall_Model"^^xsd:Na

Negative object property assertions

Negative data property assertions

Reasoner active ✓ Show Inferences

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.


Class hierarchy | Class hierarchy (inferred)

Class hierarchy: Waterfall_Model

Asserted

```
owl:Thing
  Software
    Developer
      Intelligent_Agent
        Artificial_System
        Human_Agent
    SDLC
      Model
        Agile_Development_Model
        Plan_Driven_Life_Cycle_Model
          Spiral_Model
          Waterfall_Model
      Phase
        Analysis
        Coding ≡ Implementation_and_Unit_Testing
        Implementation_and_Unit_Testing ≡ Coding
        Integration_and_System_Testing ≡ Testing
        Operation ≡ Operation_and_Maintenance
        Operation_and_Maintenance ≡ Operation
        Program_Design
        Requirement_Analysis
        Software_Requirement
        System_and_Software_Design
        System_Requirement
        Testing ≡ Integration_and_System_Testing
```

Instances: Waterfall2

For: Waterfall_Model

Waterfall1

Waterfall2

Annotations | Usage

Annotations: Waterfall2

Annotations

rdfs:comment

Enhanced Waterfall SDLC (Sommerville, 2015)

Description: Waterfall2

Types

Waterfall_Model

Developer

Same Individual As

Different Individuals

Property assertions: Waterfall2

Object property assertions

isUsedFor Project1

hasPhase Integration_and_System_Testing

hasPhase Requirement_Analysis1

hasPhase Implementation_and_Unit_Testin

hasPhase Operation_and_Maintenance1

hasPhase System_and_Software_Design1

isDoneBy Agent2

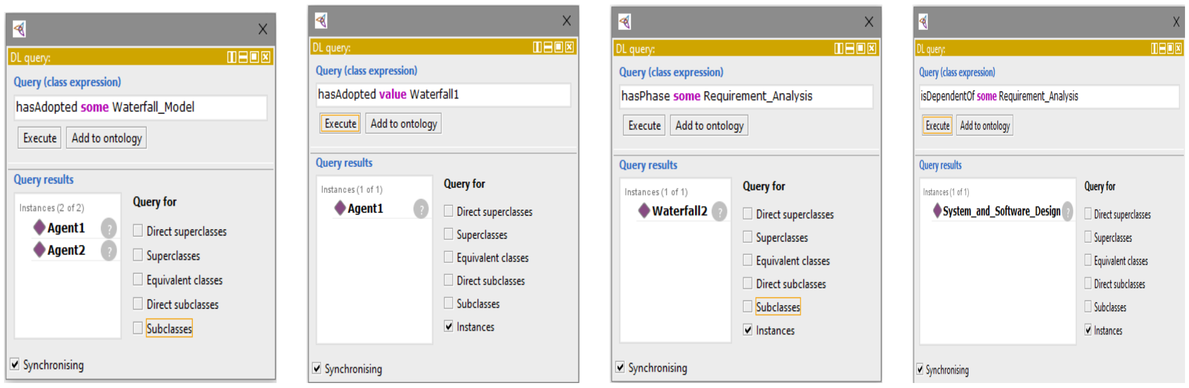
Data property assertions

isSDLC "Enhanced_Waterfall_Mo

Negative object property assertions

Reasoner active ✓ Show Inferences

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.


DL query:

Query (class expression)

hasAdopted some Waterfall_Model

Execute | Add to ontology

Query results

Instances (2 of 2)

Agent1

Agent2

Query for

☐ Direct superclasses
☐ Superclasses
☐ Equivalent classes
☐ Direct subclasses
☐ Subclasses

✓ Synchronising

DL query:

Query (class expression)

hasAdopted value Waterfall1

Execute | Add to ontology

Query results

Instances (1 of 1)

Agent1

Query for

☐ Direct superclasses
☐ Superclasses
☐ Equivalent classes
☐ Direct subclasses
☐ Subclasses
✓ Instances

✓ Synchronising

DL query:

Query (class expression)

hasPhase some Requirement_Analysis

Execute | Add to ontology

Query results

Instances (1 of 1)

Waterfall2

Query for

☐ Direct superclasses
☐ Superclasses
☐ Equivalent classes
☐ Direct subclasses
☐ Subclasses
✓ Instances

✓ Synchronising

DL query:

Query (class expression)

isDependentOf some Requirement_Analysis

Execute | Add to ontology

Query results

Instances (1 of 1)

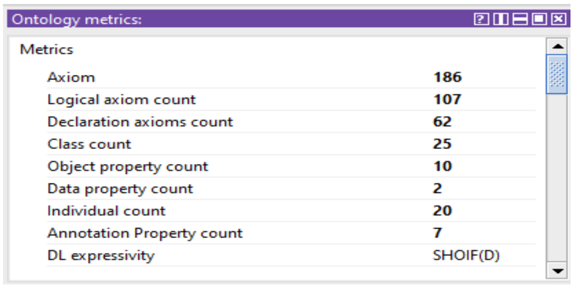
System_and_Software_Design

Query for

☐ Direct superclasses
☐ Superclasses
☐ Equivalent classes
☐ Direct subclasses
☐ Subclasses
✓ Instances

✓ Synchronising

Figure 7: Some samples of query results in context of the proposed test scenario.



Ontology metrics:

Metrics

Axiom	186
Logical axiom count	107
Declaration axioms count	62
Class count	25
Object property count	10
Data property count	2
Individual count	20
Annotation Property count	7
DL expressivity	SHOIF(D)

Figure 8: Some values of the ODYSSEY metrics.

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 $\mathcal{SHOIF}^{(D)}$. 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

Abrahamsson, P., Warsta, J., Siponen, M. T., and Ronkainen, J. (2003). New directions on Agile methods: A comparative analysis. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, pages 244–254.

Al-Hroub, Y., Kossmann, M., and Odeh, M. (2009). Developing an Ontology-driven Requirements Analysis Tool (OntoRAT): A use-case-driven approach. In *Proceedings of the IEEE International Conference on the Applications of Digital Information and Web Technologies*, pages 130–138.

Allison, I. K. and Merali, Y. (2007). Software process improvement as emergent change: A structurational analysis. *Information and Software Technology*, 49(6):668–681.

Bayat, B., Bermejo-Alonso, J., Carbonera, J. L., Facchinetti, T., Fiorini, S. R., Goncalves, P., Jorge, V., Habib, M., Khamis, A., Melo, K., Nguyen, B., Olszewska, J. I., Paull, L., Prestes, E., Ragavan, S. V., Saeedi, S., Sanz, R., Seto, M., Spencer, B., Trentini, M., Vosughi, A., and Li, H. (2016). Requirements for building an ontology for autonomous robots. *Industrial Robot*, 43(5):469–480.

Bermejo-Alonso, J. (2006). Ontology-based software engineering. Technical Report ASLab-ICEA-R-2006-016.

Calzado, J., Lindsay, A., Chen, C., Samuels, G., and Olszewska, J. I. (2018). SAMI: Interactive, multi-sense robot architecture. In *Proceedings of the IEEE International Conference on Intelligent Engineering Systems*, pages 317–322.

Ciccozzi, F., Ruscio, D. D., Malavolta, I., Pelliccione, P., and Tumova, J. (2017). Engineering the software of robotic systems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 507–508.

Davies, J., Fensel, D., and Harmelen, F. V. (2003). *Towards the semantic web: Ontology-driven knowledge management*. Wiley.

Dietz, J. (2006). *Enterprise Ontology*. Springer.

Dix, A., Finlay, J., Abowd, G. D., and Beale, R. (2004). *Human Computer Interaction*. Pearson, 3rd edition.

Dobson, G., Lock, R., and Sommerville, I. (2005). QoS-Ont: A QoS ontology for service-centric systems. In *Proceedings of the EUROMICRO International Conference on Software Engineering and Advanced Applications*, pages 80–87.

Falbo, R. D. A., Natali, A. C. C., Mian, P. G., Bertollo, G., and Ruy, F. B. (2003). ODE: Ontology-based software development environment. In *Proceedings of the Congreso Argentino de Ciencias de la Computacion*.

Fiorini, S. R., Bermejo-Alonso, J., Goncalves, P., de Freitas, E. P., Alarcos, A. O., Olszewska, J. I., Prestes, E., Schlenoff, C., Ragavan, S. V., Redfield, S., Spencer, B., and Li, H. (2017). A suite of ontologies for robotics and automation. *IEEE Robotics and Automation Magazine*, 24(1):8–11.

Glimm, B., Horrocks, I., Motik, B., Stoilos, G., and Wang, Z. (2014). HermiT: An OWL 2 Reasoner. *Journal of Automated Reasoning*, 53(3):245–269.

Gomez-Perez, A., Fernandez-Lopez, M., and Corcho, O. (2004). *Ontological Engineering*. Springer-Verlag.

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal Human-Computer Studies*, 43(5-6):907–928.

Guarino, N. (1998). Formal ontology in information systems. In *Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS)*, pages 3–15.

Hlomani, H. and Stacey, D. (2014). Approaches, methods, metrics, measures, and subjectivity in ontology evaluation: A survey. *Semantic Web Journal*, 1(5):1–11.

ISO/IEC/IEEE 12207 International Standard (2017). Systems and Software Engineering - System Life Cycle Processes.

ISO/IEC/IEEE 15288 International Standard (2015). Systems and Software Engineering - System Life Cycle Processes.

Isotani, S., Bittencourt, I. I., Barbosa, E. F., Dermeval, D., and Paiva, R. O. A. (2015). Ontology driven software engineering: A review of challenges and opportunities. *IEEE Latin America Transactions*, 13(3):863–869.

Knublauch, H. (2004). Ontology-driven software development in the context of the semantic web: An example scenario with Protege/OWL. In *Proceedings of the IEEE International Workshop on the Model-Driven Semantic Web (MDSW)*, pages 381–401.

Kolling, A., Walker, P., Chakraborty, N., Sycara, K., and Lewis, M. (2016). Human interaction with robot swarms: A survey. *IEEE Transactions on Human-Machine Systems*, 46(1):9–26.

Leonard, S., Allison, I. K., and Olszewska, J. I. (2017). Design and test (D&T) of an in-flight entertainment system with camera modification. In *Proceedings of the IEEE International Conference on Intelligent Engineering Systems*, pages 151–156.

Olszewska, J. I. (2011). Spatio-temporal visual ontology. In *Proceedings of the EPSRC Workshop on Vision and Language*.

Olszewska, J. I. (2012). Multi-target parametric active contours to support ontological domain representation. In *Proceedings of RFIA*, pages 779–784.

Olszewska, J. I. (2016). Temporal Interval Modeling for UML Activity Diagrams. In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development (KEOD)*, pages 199–203.

Olszewska, J. I., Barreto, M., Bermejo-Alonso, J., Carbonera, J., Chibani, A., Fiorini, S., Goncalves, P., Habib, M., Khamis, A., Olivares, A., de Freitas, E. P., Prestes, E., Ragavan, S. V., Redfield, S., Sanz, R., Spencer, B., and Li, H. (2017). Ontology for autonomous robotics. In *Proceedings of the IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 189–194.

Olszewska, J. I., Simpson, R. M., and McCluskey, T. L. (2014). Dynamic OWL ontology design using UML and BPMN. In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development*, pages 436–444.

Pan, J. Z., Staab, S., Assmann, U., Ebert, J., and Zhao, Y. (2012). *Ontology-driven software development*. Springer.

Ponzanelli, L., Scalabrino, S., Bavota, G., Mocci, A., Oliveto, R., Penta, M. D., and Lanza, M. (2017). Supporting software developers with a holistic recommender system. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 94–105.

Roehm, T., Tiarks, R., Koschke, R., and Maalej, W. (2012). How do professional developers comprehend software? In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 255–265.

Royce, W. W. (1970). Managing the development of large software systems. In *Proceedings of IEEE Conference of Western Electronic Show and Convention*, pages 205–210.

Sommerville, I. (2015). *Software Engineering*. Pearson, 10th edition.

Tartir, S., Arpinar, I., Moore, M., Sheth, A., and Aleman-Meza, B. (2018). OntoQA: Metric-based ontology quality analysis. In *IEEE International Conference on Data Mining Workshop*, pages 559–564.

van Kervel, S., Dietz, J., Hintzen, J., van Meeuwen, T., and Zijlstra, B. (2012). Enterprise ontology driven software engineering. In *Proceedings of the International Conference on Software Technologies (ICSOFT)*, pages 205–210.