УДК 004.43:004.02

ОПИС ЗНАНЬ ПРО ВИХІДНИЙ КОД З ВИКОРИСТАННЯМ ОНТОЛОГІЇ

DOI 10.36994 / 2788-5518-2023-01-05-14

Ткачук А.В., *аспірант, Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна, https://orcid.org/0000-0002-9127-6381, andrewtkachuk@yahoo.com*

Булах Б.В.*, к.т.н., доц., Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського", Київ, Україна. https://orcid.org/0000-0001-5880-6101, bogdan.bulakh@gmail.com*

Анотація: *Дане дослідження зосереджене на використанні онтології як формальної баз знань для представлення вихідного коду в автоматизованих системах. Шляхом створення онтології, яка відображає концепції, зв'язки та властивості елементів вихідного коду, стає можливим зберігання та організація знань про кодову базу. Це надає різноманітні функціональні можливості, такі як автоматизований аналіз коду та створення знань про структуру та поведінку коду. Метою цього дослідження є перевірка можливості використання онтології як формальної бази знань про вихідний код в автоматизованих системах. Огляд останніх статей показав, що існуючі інструменти для роботи з онтологіями не надають безпосередньо функціональності для перетворення коду в знання, тому ідея, запропонована в статті, має новизну. Стаття досліджує використання мови OWL як основної мови представлення онтології через її виразність, підтримку механізмів логічного виведення та сумісність зі стандартами семантичної мережі. Пропонується створення представлень онтології для різних елементів коду, з класами, що представляють основні типи сутностей, та окремими екземплярами, що представляють конкретні екземпляри всередині кодової бази. Для перевірки концепції та демонстрації очікуваних результатів будується система-прототип, яка складається з клієнта для розбору вихідного коду, RESTful API для зв'язку між компонентами та менеджера онтологій. Система використовує такі бібліотеки, як SwiftSyntax для розбору Swift-коду, Alamofire для HTTP-зв'язку та бібліотеки маніпулювання OWL (наприклад, OWL-API та ONT-API) для роботи з онтологіями. Система має на меті вилучення знань з вихідного коду, перетворення його на сутності та аксіоми онтології та обробку онтологію за допомогою системи отримання знань (наприклад, HermiT). Зроблено висновок, що онтології ефективно зберігають знання про вихідний код та дозволяють різноманітні маніпуляції над ними. Однак, для реалізації збору більшого обсягу знань про вихідний код необхідно провести подальше дослідження для визначення оптимального набору властивостей та сутностей, які має підтримувати система перетворення коду в онтологію.*

Ключові слова: *онтологія; OWL; база знань; система перетворення коду в онтологію; REST; семантичний веб; reasoner.*

DESCRIBING THE KNOWLEDGE ABOUT THE SOURCE CODE USING AN ONTOLOGY

Andrii Tkachuk*, Ph.D. student, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine. https://orcid.org/0000-0002-9127-6381, andrewtkachuk@yahoo.com*

Bogdan Bulakh*, Ph.D., Ass. Prof., National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine. https://orcid.org/0000-0001-5880-6101, bogdan.bulakh@gmail.com*

Abstract: *This research focuses on the use of ontologies as formal knowledge bases for representing source code in automated systems. By creating an ontology that represents the concepts, relationships, and properties of source code elements, it becomes possible to capture and organize knowledge about the codebase. This enables various functional abilities such as automated code analysis and reasoning about code structure and behavior. The aim of this research is to verify the ability to use ontology as a formal knowledge base about source code in automated systems. A review of the recent articles showed that existing tools to work with ontologies does not provide direct functionality to convert the code into a knowledge, so the idea proposed in the article has novelty. The research explores the use of the Web Ontology Language (OWL) as the primary ontology representation language due to its expressiveness, reasoning capabilities, and compatibility with semantic web standards. The research proposes the creation of ontology representations for various code entities, with classes representing main entity types and individuals representing specific instances within the*

codebase. To validate the concept and demonstrate the expected results, a proof-of-concept system is built, consisting of a parsing client, a RESTful API for communication, and an ontology manager. The system utilizes libraries such as SwiftSyntax for parsing Swift code, Alamofire for HTTP communication, and OWL manipulation libraries (such as OWL-API and ONT-API) for working with ontologies. The system aims to extract knowledge from the source code, convert it into ontology entities and axioms, and reason over the ontology using a reasoner (such as HermiT). The research concludes that ontologies can effectively store knowledge about source code and enable various further manipulations on that knowledge. However, further investigation is needed to determine the optimal set of properties and entities that should be supported by the code ontology extraction system to infer more knowledge about the source code.

Keywords: *ontology; OWL; knowledge base; code ontology extraction; REST; semantic web; reasoner.*

Introduction

An ontology is a formal representation of knowledge that defines concepts, their properties, and the relationships between them within a particular domain. It provides a structured framework for organizing and categorizing information, allowing for better understanding and communication between humans and machines. Ontologies are commonly used in information science and knowledge management to enable efficient data integration, reasoning, and semantic interoperability.

By creating an ontology that represents the concepts, relationships, and properties of source code elements such as classes, methods, variables, and their interactions, it becomes possible to capture and organize knowledge about the codebase. This can enable various useful features such as an automated code analysis, code documentation generation, semantic searching, and reasoning about code structure and behavior. Ontologies can help developers and software engineers better understand and manipulate source code by providing a structured representation of its semantics [1].

Advanced automated refactoring and source code analysis systems may benefit from properties of a knowledge base which are backed by the ontology. They are able to store references between code entities, strictly define their types, query the base, and perform any kinds of data reasoning. They may help to add or rewrite source code with insurance that it will compile (in contrast with AI systems with unspecified or fuzzy production rules) [2].

Problem definition. Using ontologies to store the knowledge about the source code can be complex to create and maintain, requiring domain expertise and ongoing effort. Additionally, capturing the full scope of a codebase and managing scalability can be challenging, making ontology-based approaches less suitable for large and diverse codebases.

The aim of this research is to investigate the possibility to use ontology as a formal knowledge base about source code in automated systems.

Recent research articles review. Ontologies can be represented using different formal languages, with the most commonly used one being the Web Ontology Language (OWL). OWL provides a standardized and expressive syntax to define concepts, properties, relationships, and axioms within an ontology. Other representations include RDF (Resource Description Framework), which allows for the creation of semantic triples, and graphical notations like UML (Unified Modeling Language) for visualizing ontologies [3].

OWL is considered better for representing ontologies due to its higher expressiveness, semantic web standards compliance, reasoning capabilities, and a well-developed tooling ecosystem, enabling precise modeling and interoperability. However, the choice of ontology representation language ultimately depends on the specific requirements and complexity of the domain being modeled [4; 5].

Taking into consideration all aforementioned facts, OWL was selected as a main tool to create a knowledge base about the source code.

OWL (Web Ontology Language) consists of classes for representing concepts, properties for defining relationships, and individuals to represent specific instances within a domain. It also includes axioms to express logical statements, such as subclass relationships and property characteristics, as well as logical constructs and reasoning capabilities for inferring new knowledge and checking

ontology consistency. Together, these components form a comprehensive framework for creating and utilizing ontologies to represent knowledge in a structured and semantically rich manner.

To verify the concept and prove that a knowledge base powered by an ontology will give some expected result, the following representations may be created: large group types (like classes, structures, enumerations, functions, etc.) may be presented by OWL classes, each entity in the source code may be represented as named individual belonging to its respective class, properties of entities may be expressed by data properties, and relations between entities may be expressed object properties. Also, to fulfill all the gaps, to build truly semantic representation of the codebase, and to enable reasoning, a large set of axioms may be used (including but not limited to subclass, equivalent class, disjoint class, data property, object property axioms).

To be able to get the missing axioms in ontology, to verify consistency with code smells (if any defined in ontology), and to make amends in codebase respectively, a reasoner should be used. A reasoner is a software tool or component that performs automated logical reasoning over an ontology. It utilizes the defined axioms, rules, and logical constructs within the ontology to derive new knowledge, check consistency, and answer queries. Reasoners can infer implicit relationships, identify inconsistencies, validate data against the ontology, and provide logical conclusions based on the ontological knowledge. For the purposes of this research, HermiT reasoner has been selected [6; 7].

Preliminary research results show that there are solutions available that aim to convert a codebase into an ontology or utilize ontologies to represent code-related knowledge. These solutions typically employ techniques from software analysis, natural language processing, and ontology engineering. Some examples include Code Ontology Extraction, Reverse Engineering Tools, Ontology Mapping and Integration, etc.

It's important to note that while these solutions exist, the process of converting a codebase into an ontology can be challenging, and the resulting ontology may require manual refinement and maintenance. The complexity and nuances of code often make it difficult to capture all the relevant information accurately. Additionally, the choice of specific tools or techniques may vary depending on the programming language, codebase size, and desired ontology representation.

Further research on possible applications of those tools (JOIE, SemTK (over Jena), SourcererCC) revealed that even though they provide some possibilities to easily manage RDF triplets or OWL ontologies, they do not provide API or any other ability to convert codebase into ontology. Such conclusion testifies that the aim of current research (to use the ontology as knowledge base about the source code) has some novelty.

Code ontology extraction is the process of automatically extracting knowledge from source code and representing it in the form of an ontology. It involves analyzing the codebase to identify various code elements such as classes, methods, variables, and their relationships. The goal is to capture the structure, semantics, and interdependencies within the codebase and transform them into a structured representation that adheres to ontology concepts, properties, and relationships.

Code ontology extraction typically involves techniques from software analysis, parsing, and semantic understanding. These techniques may include static analysis, abstract syntax tree (AST) traversal, type inference, and pattern recognition. By analyzing the codebase, extracting relevant information, and mapping it to ontology constructs, code ontology extraction aims to provide a formal representation of the code's structure and semantics.

The extracted code ontology can enable various applications such as code search, code reuse, documentation generation, and automated reasoning about code behavior. It enhances the understanding and utilization of the codebase by providing a structured and semantic representation of its elements and relationships.

In this research we are concentrating at code ontology extraction process and aiming at implementing the model which may be used by automated tools for refactoring and analysis. At first, the idea was to use the same programming language (which is used to write source code) to

represent the knowledge base [8; 9]. It will give the possibility to quickly interpret source code and convert it to OWL entities and axioms.

For such reason, several frameworks and tools were considered. As the main research language is Swift, only libraries with such language supported were taken into consideration. One of them is OWL-API for iOS [10] with respective reasoner – Mini-ME Swift [11]. Their authors state that there is a lack of semantic reasoners and OWL manipulation libraries for Swift which makes it reasonable to develop one. Porting existing solutions or use of multiplatform libraries may be hard due to architectural differences.

Unfortunately, provided frameworks and libraries are on their initial stage of development. They do not support ontology creation (only parsing) and specific OWL entities (such data properties) are still not available.

Also, using the same language for parsing client and knowledge base makes these two instances tightly coupled which may lead to unnecessary rework when one of them changes (Fig. 1).

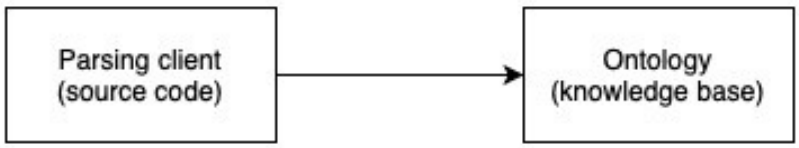

Fig. 1. Dependency between client and ontology

In order to break the coupling, an abstraction should be used (according to Dependency Inversion principle). It empowers creation of flexible and maintainable code by inverting the dependency direction, where higher-level modules depend on abstractions rather than concrete implementations, enabling easier substitution and extensibility of components. Data Transfer Object (DTO) may be used as intermediary in this case. DTO agreement may be established and DTO models to be built on both sides – client and ontology handler (Fig. 2). This will help to uncouple the instances and provide the ability to integrate abstract ontology with different parsing clients (several programming languages) or ontology management systems.

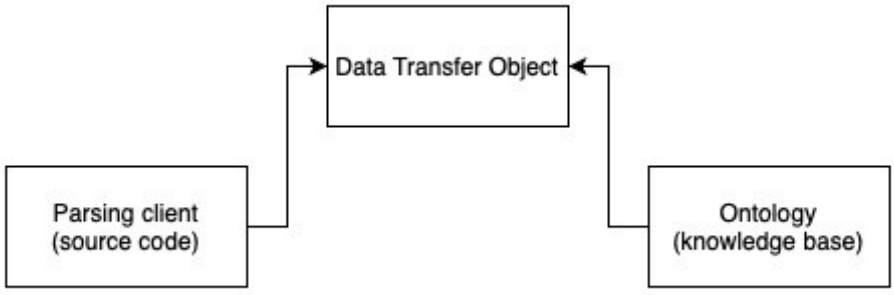

Fig. 2. Breaking dependency with DTO

It is worth saying that implementing our own library for OWL ontology manipulation is not reasonable. Firstly, it will require significant efforts and resources to build and maintain, though writing such library is not the primary aim of the research. Secondly, developers and designers should possess distinctive knowledge about those technologies. Lately, existing OWL libraries and frameworks provide well-tested and optimized solutions with extensive tooling support, which can save development effort and ensure compatibility with established standards and practices.

In order to support all the required traits of knowledge base generated by code ontology extraction system (and to build such system), specific libraries should be taken into consideration.

OWL-API [12] and ONT-API [13] are the two worth mentioning. They are Java libraries to work with ontologies. OWL-API provides support of OWL2 ontologies (ability to create, update, serialize/deserialize them). In contrast, ONT-API (which is built on top of OWL-API) provides support for RDF and is built over Apache Jena. These libraries have all the necessary ontology properties supported to be able to be used in code ontology extraction system.

Research Details

Taking into consideration all the finding from latest research and articles, it is worth building code ontology extraction system, which is language independent, uses abstractions, utilizes OWL manipulation libraries with sufficient functionality, easily extensible and able to be integrated with other systems (ontology or codebase manipulating ones).

In order to build a proof of concept and show that proposed idea makes sense, it is decided to build:

- the parsing client in Swift language which will parse the Swift codebase;
- RESTful API which will accept JSON DTOs of code entities [14];
- OWL ontology manager, which will handle all the interactions with ontology.

The principal component diagram is displayed on fig. 3. Due to the architecture, there may be several clients connecting to the Ontology service and Ontology service itself may work with different ontology managers or tools for ontology reasoning or visualization.

Before work with ontology can be started, it is worth designing its structure.

On Fig. 4 a structure of ontology is depicted. There were three classes created: *Class, Function, Property*. All the entities from a codebase which fit the type of class will be transformed into named individuals which belong to respective ontology class. Those instances will be linked with each other by the following object properties: "*hasMethod*", "*hasProperty*", "*hasArgument*". In order to support bidirectional links, two inverse object properties counterparts should be created: "*isInClass*", "*isInMethod*".

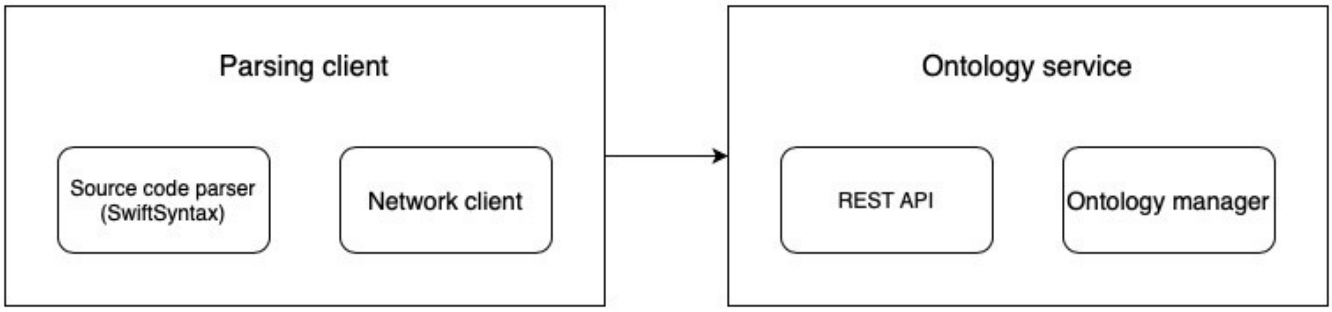

Fig. 3. Principal diagram of code ontology extraction system

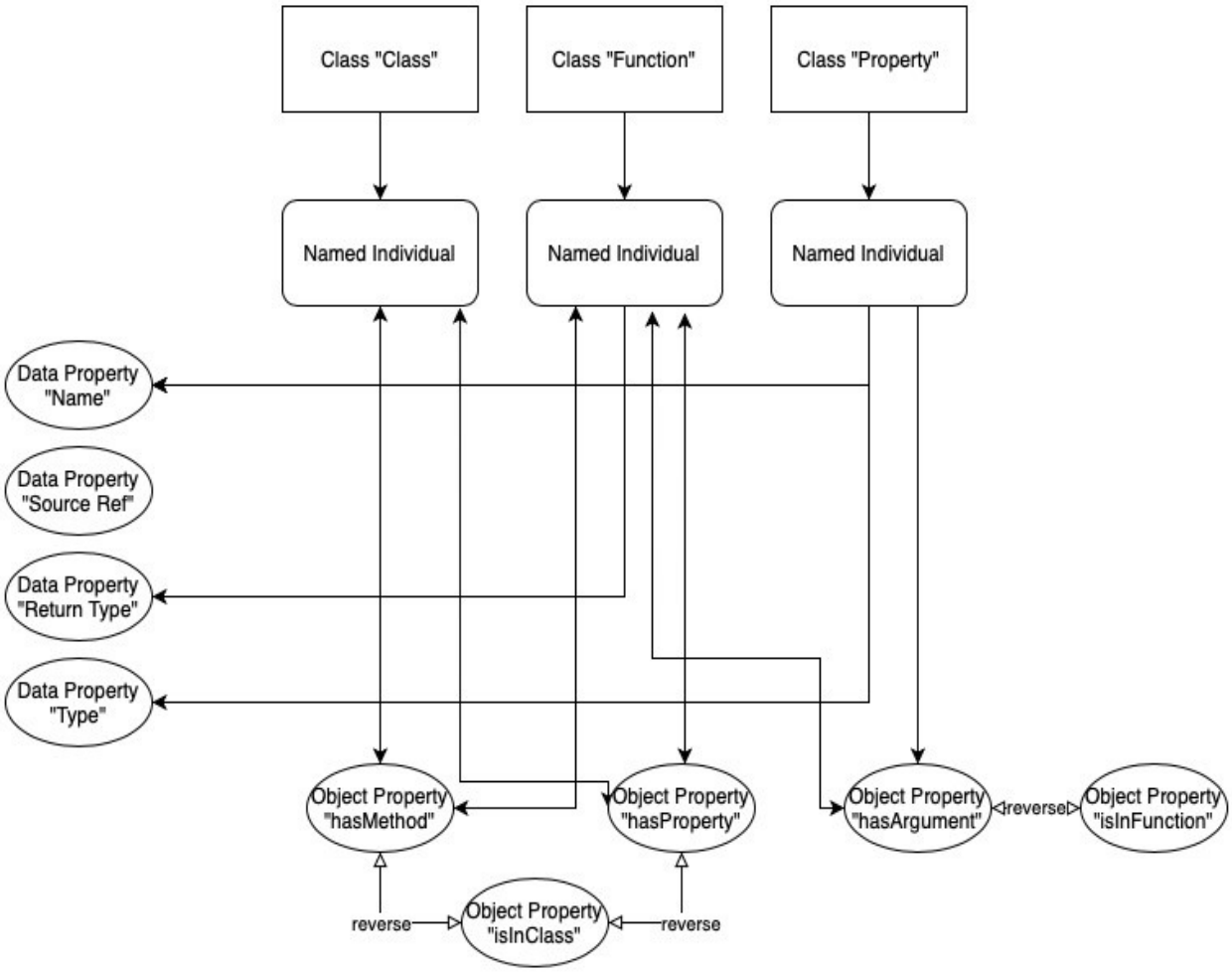

Fig. 4. Ontology structure

To be able to describe named individuals, a set of data properties is created. They are "*name*", "*type*" (data type, like *integer*), "*returnType*" (data type). Additionally, there is a "*sourceRef*" (meaning "*SourceCodeReference*") property which states where in the codebase specific entity is located. That property is beneficial as it will allow to uniquely identify the source code entity and provide backward compatibility – to be able to generate or alter code from the ontology.

Additionally, "*isInClass*" and "*isInMethod*" object properties are defined as disjoint, meaning that named individual which can have both these properties (Property in our case) cannot have these two relations associated at one time.

On the Parsing client side the SwiftSyntax library [9] is used to parse source code written in Swift language. Then parsed nodes of AST are converted into DTO – defined entities, which are known by Ontology service. The following code fragment depicts a piece of code which parses the VariableDeclSyntax node (represent properties of class, structure, enumeration, or regular variables) and transforms it to a Property object:

```
override func visit(_ node: VariableDeclSyntax) -> SyntaxVisitorContinueKind {
    let name = node.bindings.first?.pattern.description ?? ""
    let type = node.bindings.first?.typeAnnotation?.type.description ?? ""
    let property = Property(name: name,
                            type: type,
```

```
                    sourceCodeReference: node.sourceRange(
                                                converter: converter).debugDescription)
        properties.append(property)
        return .skipChildren
    }
```

As the last step, Alamofire library is used to connect to the Ontology service via HTTP and transfer all visited entities in provided source code. On the Ontology service side, a simple HTTP server from the Sun package provides REST API for communication. Here is a piece of Java code which handles the input and created new named individual that belongs to class "Class":

```
@Override
public String handleInput(String input) {
    Class classEntity = converter.fromJson(input, classOfT:Class.class);
    return InternalOntology.sharedInstance().addClass(classEntity);
}
```

Data which is transferred between Parsing client and Ontology service is in JSON format. The following code sample creates a new named individual:

```
private OWLNamedIndividual createFunction(Function func) {
    IRI identifier = IRI.create(iri, UUID.randomUUID().toString());
    OWLNamedIndividual funcDecl = dataFactory.getOWLNamedIndividual(identifier);
    ontology.addAxiom(dataFactory.getOWLDeclarationAxiom(funcDecl));
ontology.addAxiom(dataFactory.getOWLClassAssertionAxiom(
        functionEntity, funcDecl));
    ontology.addAxiom(dataFactory.getOWLDataPropertyAssertionAxiom(
        nameProperty, funcDecl, func.name));
    ontology.addAxiom(dataFactory.getOWLDataPropertyAssertionAxiom(
        returnTypeProperty, funcDecl, func.returnType));
    ontology.addAxiom(dataFactory.getOWLDataPropertyAssertionAxiom(
        sourceCodeReferenceProperty, funcDecl, func.sourceCodeReference));

    for (Property prop: func.arguments) {
        OWLNamedIndividual propDelc = createProperty(prop);
        ontology.addAxiom(dataFactory.getOWLObjectPropertyAssertionAxiom(
            hasArgumentProperty, funcDecl, propDecl));
    }
    return funcDecl;
}
```

Firstly, a unique IRI is created for a new instance. Secondly, new named individual object is created, and all possible axioms are added. Lately, all dependent entities (Property named individuals in this case) are created in the same manner, which is described in first two steps.

To verify that expected ontology contains all the necessary axioms, a test should be conducted. For testing purposes, a simple code snippet was created. Lets assume that this snipped is a part of some bigger source file (note that we have added some line numbers for better understanding the idea of how *Source Code Reference* is used). This demo code snippet has some very simple class hierarchy yet demonstrating the inheritance and method overriding:

```
 7 ...
 8 protocol Say {
 9     func say() -> String
10 }
11
12 class Animal: Say {
13     let age: Int
14
15     init() {
16         age = 0
17     }
18
19     func say() -> String {
20         return ""
21     }
22 }
23
24 final class Cat: Animal {
25     private var lives: Int
26
27     override init() {
28         lives = 9
29         super.init()
30     }
31
32     override func say() -> String {
33         return "meow"
34     }
35 }
36
37 final class Dog: Animal {
38     private var senses: Int
39
40     override init() {
41         senses = 6
42         super.init()
43     }
44
45     override func say() -> String {
46         return "woof"
47     }
48 }
49 ...
```

After parsing the source code, ontology was created. Then, Hermit reasoner was run to infer all possible axioms. As a result, inversions, and basic ontology class hierarchy were added. Updated ontology is on fig. 5, where bold elements depict the inference results added to the initial ontology by Hermit reasoner.

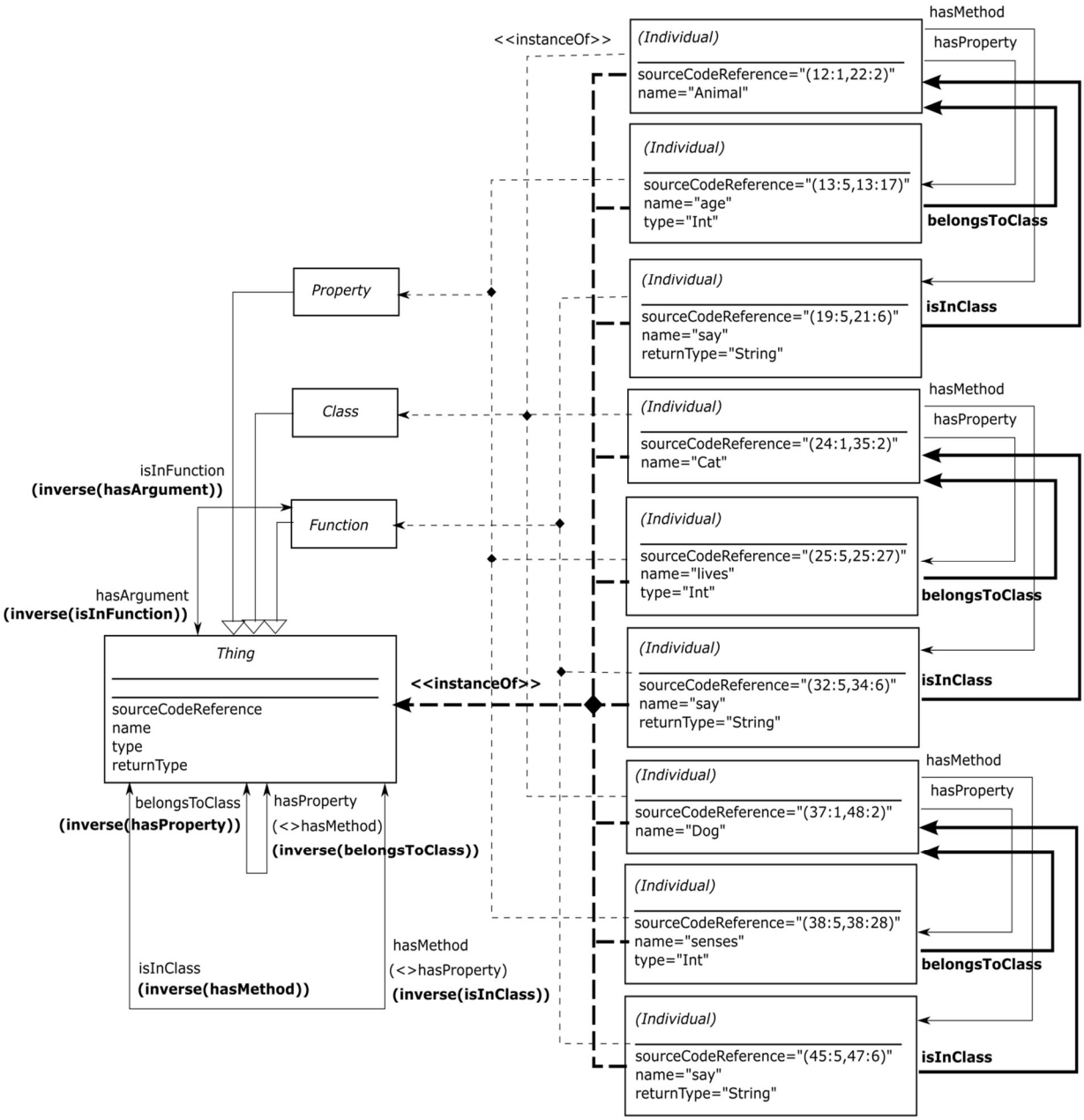

Fig. 5. Ontology after reasoning (inferred axioms are in bold, individual's identifiers are skipped for brevity)

Unfortunately, not all expected knowledge about the code appeared in the ontology. As it may be seen on the code snippet, there are hierarchical relations (inheritance and protocol conformance) in the code, but neither initial ontology nor updated one after reasoning do not contain those relations. This means that further investigation and design should be conducted to find and describe the optimal set of properties and entities which should be supported by code ontology extraction system to be able to infer more knowledge about source code.

Conclusions

Review of the recent research and articles has showed that ontology is a powerful tool to represent domain semantics and knowledge. It is safe to state that ontologies can be used to efficiently keep the knowledge about the source code in particular.

In order to be able to represent codebase efficiently, a wide range of the source code ontology's 'building blocks' must be defined. They include but not limited to classes, named individuals, data properties, object properties and wide variety of axioms.

When all the axioms in ontology are correctly representing the state of the codebase, a reasoner may help to infer implicit or unknown relations/axioms and add them to the initial ontology. Such ability of the proposed system allows to add rules to the ontology (such as code smell anti-patterns and approaches to fix them) to remove, refactor or generate code which will compile.

Several approaches to convert codebase to ontology were discussed. This article has shown that the code ontology extraction is quite suitable approach to transform the source code into a knowledge base represented by OWL2 ontology.

In order to verify the proposed concept, a design of simple basic knowledge base was developed, and a code ontology extraction system prototype was built. Its architecture and principal design points were discussed. As a result, a code snippet was transformed into a simple ontology. Implicit and unknown axioms were inferred by the reasoner and successfully added to the initial ontology.

That simple knowledge base structure used does not cover all the possible relations and code entities, so in order for the system be able to infer complex data (like protocol conformance) it is necessary to design a set of axioms which will allow the reasoner to do so and submit sufficient data about source code to the ontology manager. The further research will be devoted to the development of the more complex semantic foundation for the real-world source code knowledge extraction and manipulation.

Література

1. Happel, H-J., Seedorf, S. Applications of ontologies in software engineering. *In Proc. of Workshop on Sematic Web Enabled Software Engineering (SWESE) on the ISWC*, 2006, pp. 5-9 .
2. As we may code. [Електронний ресурс] - Режим доступу: https://nshipster.com/as-we-may-code/#skip
3. OWL 2 Web Ontology Language. [Електронний ресурс] - Режим доступу: https://www.w3.org/TR/owl2-syntax/
4. Semantic Web Tool. [Електронний ресурс] - Режим доступу: https://www.w3.org/wiki/SemanticWebTools
5. Resource Description Framework (RDF). [Електронний ресурс]. - Режим доступу: https://www.w3.org/RDF/
6. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z. HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning*, Springer Netherlands, 2014, No. 53, pp. 245 – 269.
7. Horrocks, I., Motik, B., Wang, Z. The HermiT OWL Reasoner. *CEUR Workshop Proceedings*, 2012, Vol. 3(858).
8. Tkachuk, A., Bulakh, B. Research of possibilities of default refactoring actions in Swift language. *Technology Audit and Production Reserves*, 2022, No. 5(2(67), pp. 6–10. DOI: 10.15587/2706-5448.2022.266061
9. Tkachuk, A., Bulakh, B. Usage of formalized knowledge about source code for refactoring actions in Swift. *Technology Audit and Production Reserves*, 2022, No. 6(2(68), pp. 6–10. DOI: 10.15587/2706-5448.2022.267160
10. Ruta, M., Scioscia, F., Di Sciascio, E., Bilenchi, I. OWL API for iOS: early implementation and results. *13th OWL: Experiences and Directions Workshop and 5th OWL reasoner evaluation workshop (OWLED - ORE 2016)*, Springer, 2016, vol. 10161, pp. 141–152, DOI: 10.1007/978-3-319-54627-8
11. Ruta, M., Scioscia, F., Gramegna, F., Bilenchi, I., Di Sciascio, E. Mini-ME Swift: The First Mobile OWL Reasoner for iOS, *In: The Semantic Web. ESWC 2019. Lecture Notes in Computer Science,* 2019, vol. 11503, Springer, Cham, pp. 298-313. DOI: 10.1007/978-3-030-21348-0_20
12. An Introduction to the OWL API. [Електронний ресурс] - Режим доступу: http://syllabus.cs.manchester.ac.uk/pgt/2019/COMP62342/introduction-owl-api-msc.pdf
13. ONT-API: an RDF-centric Java library to work with OWL2. [Електронний ресурс] - Режим доступу: https://github.com/owlcs/ont-api
14. Dirsumilli, R., Mossakowski, T. RESTful Encapsulation of OWL API. *5th International Conference on Data Management Technologies and Application*s, 2016, pp. 150 - 157. DOI: 10.5220/0005987201500157.

References

1. Happel, H-J., Seedorf, S. (2006). Applications of ontologies in software engineering. *In Proc. of Workshop on Sematic Web Enabled Software Engineering (SWESE) on the ISWC*, pp. 5-9 .
2. As we may code. [Електронний ресурс] - Режим доступу: https://nshipster.com/as-we-may-code/#skip
3. OWL 2 Web Ontology Language. [Електронний ресурс] - Режим доступу: https://www.w3.org/TR/owl2-syntax/
4. Semantic Web Tool. [Електронний ресурс] - Режим доступу: https://www.w3.org/wiki/SemanticWebTools
5. Resource Description Framework (RDF). [Електронний ресурс]. - Режим доступу: https://www.w3.org/RDF/
6. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z. (2014). HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning*, Springer Netherlands, No. 53, pp. 245 – 269.
7. Horrocks, I., Motik, B., Wang, Z. (2012). The HermiT OWL Reasoner. *CEUR Workshop Proceedings*, Vol. 3(858).
8. Tkachuk, A., Bulakh, B. (2022). Research of possibilities of default refactoring actions in Swift language. *Technology Audit and Production Reserves*, No. 5(2(67), pp. 6–10. DOI: 10.15587/2706-5448.2022.266061

9. Tkachuk, A., Bulakh, B. (2022). Usage of formalized knowledge about source code for refactoring actions in Swift. *Technology Audit and Production Reserves*, No. 6(2(68), pp. 6–10. DOI: 10.15587/2706-5448.2022.267160

10. Ruta, M., Scioscia, F., Di Sciascio, E., Bilenchi, I. (2016). OWL API for iOS: early implementation and results. *13th OWL: Experiences and Directions Workshop and 5th OWL reasoner evaluation workshop (OWLED - ORE 2016)*, Springer, vol. 10161, pp. 141–152, DOI: 10.1007/978-3-319-54627-8

11. Ruta, M., Scioscia, F., Gramegna, F., Bilenchi, I., Di Sciascio, E. (2019). Mini-ME Swift: The First Mobile OWL Reasoner for iOS, *In: The Semantic Web. ESWC 2019. Lecture Notes in Computer Science*, vol. 11503, Springer, Cham, pp. 298-313. DOI: 10.1007/978-3-030-21348-0_20

12. An Introduction to the OWL API. [Електронний ресурс] - Режим доступу: http://syllabus.cs.manchester.ac.uk/pgt/2019/COMP62342/introduction-owl-api-msc.pdf

13. ONT-API: an RDF-centric Java library to work with OWL2. [Електронний ресурс] - Режим доступу: https://github.com/owlcs/ont-api

Dirsumilli, R., Mossakowski, T. (2016). RESTful Encapsulation of OWL API. *5th International Conference on Data Management Technologies and Application*s, pp. 150 - 157. DOI: