Towards the Automation of Scientific Publications Processing: Generating OWL Files from PDF Documents

Mykola Petrenko[1,*], Mykola Boyko[1]

[1] Glushkov Institute of Cybernetics of the National Academy of Sciences of Ukraine, 40 Glushkov ave., Kyiv, 03187, Ukraine

Abstract

The article deals with the process of automating the processing of scientific publications by converting PDF documents into OWL files. The conception of Knowledge-related System for processing scientific publications, which uses Semantic Web technologies to work with scientific articles from leading Ukrainian journals, is briefly described. The main information structure is knowledge a graph, which enables effective knowledge accumulation and transfer. The architecture of the Knowledge-related System for processing scientific publications is presented, including the preparatory stage using Protege application, as well as the main stages of the system's operation in the local network and via a remote access point. Particular attention is paid to the automatic generation of OWL files from PDF documents, which comprises three stages: extracting and cleaning text, identifying entities, and generating OWL files. This research contributes to the development of scientific information processing tools and the expansion of the possibilities of applying artificial intelligence in science.

Keywords

Knowledge-related system, knowledge graphs, ontology, Semantic Web technologies

1. Introduction

Information technologies are constantly evolving and have recently reached a transformative level, especially to address societal needs in timely and reliable presentation of information content. An essential step to improve the presentation of the latter is the introduction of artificial intelligence systems, in particular, Large Language Models (LLMs) [1]. In Ukraine, their use is still limited for obvious reasons. In addition, the technologies for using LLM are quite complex and require a high level of knowledge of the English language and relevant terminology. Therefore, researchers primarily need free applications with significant content of scientific information from well-known professional journals published in Ukraine.

The Knowledge-related system for processing scientific publications (KrS) is presented as this application using Semantic Web technologies, and the main journals for processing scientific articles are Cybernetics and Systems Analysis, Programming Problems, Control Systems and Computers, and others.

Knowledge graphs are the basic information structure used at different stages of KrS functioning. In [2, 3] the definition of conceptual graphs is presented, that is currently the most accepted by the scientific community.

> *A knowledge graph is a data graph intended to capture and transmit real-world knowledge, with nodes representing objects and edges representing relationships between these objects. A data graph (also known as a data graph) corresponds to a graph model of data, which can be an oriented graph with labels on the edges, a property graph, etc.*

Workshop "Software engineering and semantic technologies" SEST, co-located with 15th International Scientific and Practical Programming Conference UkrPROG'2025, May 13-14, 2025, Kyiv, Ukraine

* Corresponding author.

petrng@ukr.net (M. Petrenko); xeldag@ukr.net (M. Boyko)

0000-0001-6440-0706 (M. Petrenko); 0000-0003-1723-5765 (M. Boyko)

In addition to these challenges, the integration of knowledge graphs into digital health and telerehabilitation [4, 5, 6, 7, 8, 9] presents an opportunity to enhance patient care and streamline remote healthcare services. By structuring medical knowledge through knowledge graphs, telerehabilitation systems can leverage semantic relationships between medical concepts, enabling more accurate diagnostics and personalized treatment recommendations. Furthermore, the integration of structured medical data with telehealth platforms can improve interoperability, facilitating seamless communication between healthcare [7, 8, 9] providers and patients. These advancements have the potential to enhance the accessibility and efficiency of digital health solutions while addressing key challenges in remote patient monitoring and rehabilitation.

It should be noted that the *KrS* prototype is described in [10, 11, 12], which implies functioning at the preparatory stage and the main stages *A* and *B*.

The preparatory stage involves configuring the *Protege* application and creating (manually) an RDF repository of scientific publications (a set of OWL files).

The main stage *A* is designed to work in a local network, which includes the main computer of *the Knowledge Engineer* (KE) and user computers. The KE computer runs the *Protege* application with all the necessary functionality extensions, in particular the Hermit logical inference mechanism. At this stage, all the mechanisms of the system were tested.

The main stage *B* is designed to work with a remote endpoint deployed on the *Apachi Jena Fuseki* server.

After testing the system prototype, two main issues arose that concerned both system developers and users. The first of them is the automatic formation of the SP database, which consists of a set of ontologies of pdf documents of scientific articles or OWL files. The second issue is the automatic generation of SPARQL queries that describe natural language queries of users. The first issue will be discussed below, namely, the automatic generation of an OWL file from a pdf description of a scientific article.

In Figs. 1–3 show three diagrams that describe the architecture of KrS, from the context diagram (Figure 1, C4 model, [13, 14]), the container diagram (Figure 2, C4 model, [15]) to the component diagram (Figure 3, Archi Mate model [16, 17]).

2. Automatically generate OWL files from PDF articles

Automating the process of creating OWL files is the next logical step after agreeing on their structure. As a source of articles, we have selected open access materials from several journals, including Cybernetics and Systems Analysis. An article from this journal will be used as an example in this text.

Currently, the supported file format for conversion is pdf, but the list of supported formats may be expanded. The entire task of automatically converting PDF files to OWL files can be divided into three stages.

Extracting text from a PDF file and its initial processing. Since the received text may contain unnecessary information (page numbers, headers, etc.) that is mixed with the main text of the article, it is necessary to “clean it up” at this stage.

Separate the essences of the article. From the resulting monolithic text, you need to isolate the key entities that will serve as the “building blocks” for the OWL file.

Generation an OWL file. Based on the selected entities, you need to generate an OWL file that will correspond to the previously agreed structure.

Let's take a closer look at each of these stages.

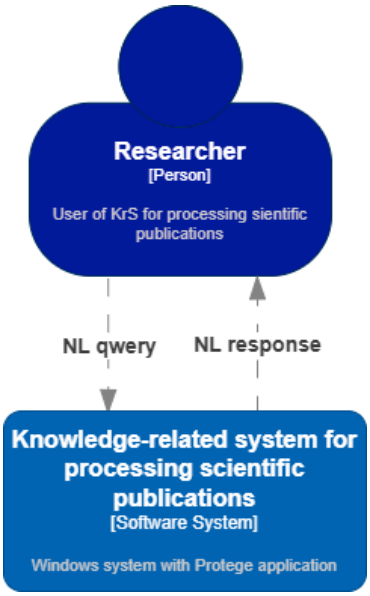

Figure 1: Context diagram of KrS

2.1 First stage

There are a number of well-known software libraries and methodologies for working with PDF files, both with paid and free content. For example, IronPDF, iText7, jPDFProcess, OpenPDF, iText, Apache PDFBox [18] and [19]. To understand these tools better, you can refer to [20].

For our project, we chose the free Apache PDFBox library, which is well matched to the Apache Jena Fuseki server used in our system. It is a "pure" open source Java library. Apache PDFBox allows you to create, visualize, print, split, combine, modify, validate, and extract text and metadata from PDF files. It is the last of these features that is used in this project. To learn more about PDFBox, consult official sources. You may also find additional details in [21].

Maven, a tool for automating the construction of Java projects, is used in this work. To integrate Apache PDFBox into a software project, you need to add the following lines to the pom.xml configuration file, which will allow you to use the library's functionality in the program you are developing.

```
<dependencies>
  <dependency>
    <groupId>org.apache.pdfbox</groupId>
    <artifactId>pdfbox</artifactId>
    <version>2.0.31</version>
  </dependency>
</dependencies>
```

Getting text from a PDF file using the Apache PDFBox library is quite simple. To do this, just create an object of the PDDocument class and call the load() method to load the file. To get a text string, you can use the getText() method of the PDFTextStripper class.

Thus, the result of these conversions will be a monolithic text string with the above features.

The program code may look like this:

```
text = getText(new File(args[0]));
static String getText(File pdfFile) throws IOException {
PDDocument doc = PDDocument.load(pdfFile);
return new PDFTextStripper().getText(doc);
}
```

Note that the program code snippets are written in Java, unless otherwise specified. Also, the program code is provided for example only and is not completely ready for compilation. You can use any Java editor and compiler to experiment with this code.

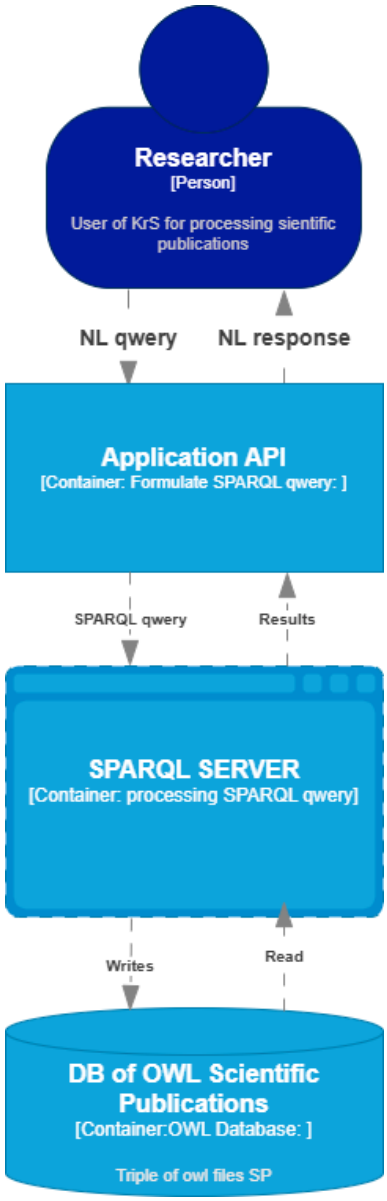

Figure 2: Container diagram of KrS

Next, let`s take an arbitrary PDF document on the basis of which we will consider the operation of the algorithm (program). To do this, you can take one of the open-access articles from the journal "Cybernetics and Systems Analysis", in particular, the 2017 article, volume 53, number 4, entitled "Noospheric Paradigm of Science Development and Artificial Intelligence".

Next, let's take a look at the text obtained with Apache PDFBox. It shows that the text has an incorrect character encoding instead of the expected one. The reason for this is the use of outdated encoding methods. In the past, eight-bit encoding was used to encode English alphabet characters, numbers, and some service characters (character information is stored in eight bits). This basic encoding table is called ASCII. In this system, seven bits used to actually encode the character, and the eighth bit was left free.

Subsequently, many encodings for national alphabets were developed using the eighth bit, for example: Windows-1251, CP-866, KOI-8R, ISO-8859-5. We do not know what encoding is used in the text we received. For a program, it looks like a string of bytes without any additional meaning.

In the modern Unicode encoding system, there are no such problems, since there is no need to assign the same codes to different characters. It is likely that the PDF file we use as an example was created using one of the older encoding systems. Since Java treats any text string as Unicode, we get incorrect text as output. To learn more about challenges in processing human languages, see [22].

Since we have no additional information about the encoding used, it seems that the only way to determine it is to do it manually. There are many services that try to "guess" the encoding or allow

the user to choose it themselves, for example(https://2cyr.com/decode/?lang=uk). With the help of such a tool, you can choose a suitable encoding system manually. It is worth noting that sometimes it is impossible to achieve 100% transcoding accuracy when converting text from one code page to another, and some characters may be lost.

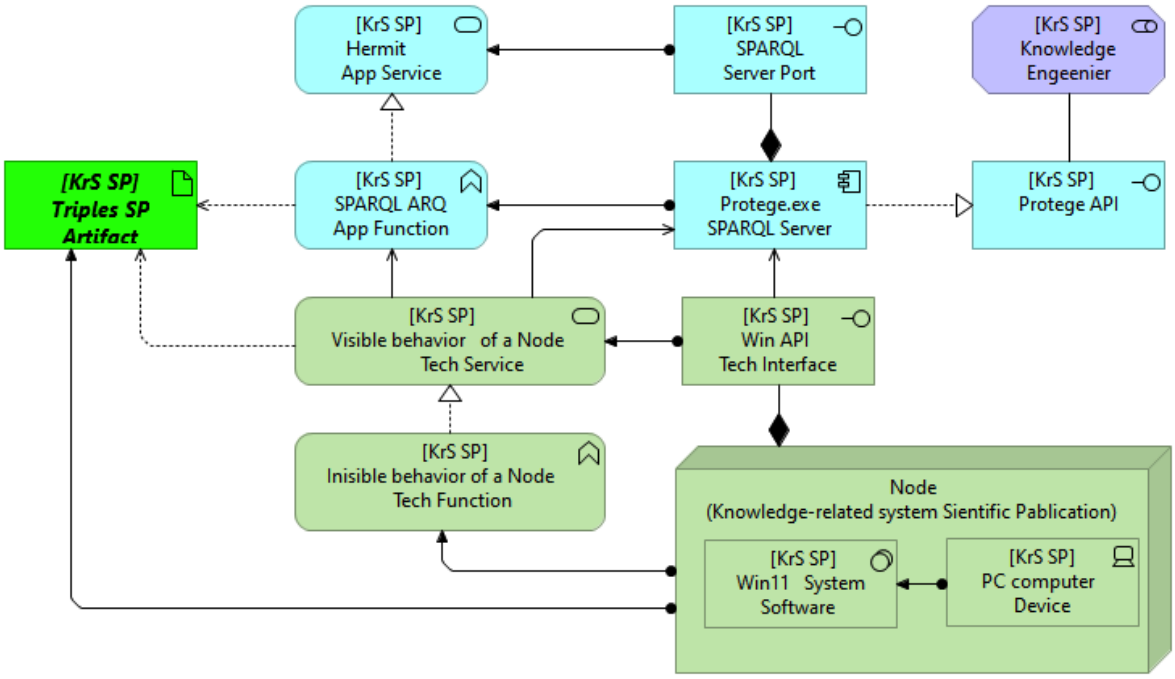

Figure 3: Component diagram of KrS

After determining the correct encoding, we can adjust our text string.

The program code for this will look like this:

convertedString = new String(convertThis.getBytes("Windows-1252"), "Windows-1251");

Now that we have the normalized text, we can look at the correct lines from the article we have selected, such as

"ISSN 1019-5262. Cybernetics and System Analysis, 2017, Vol. 53, No. 4, p. 13" '© O.V. Palagin, O.P. Kurgaev, A.I. Shevchenko, 2017',

Next, you need to select these lines from the monolithic text and delete them, as they may interfere with the next step. Regular expressions can be used for this purpose.

Regular expressions are sequences of characters that define a pattern of matches in the text. They are commonly used in algorithms to search for text fragments, replace substrings, or validate input. For a more in-depth look at regular expressions, see [23].

The Java language supports regular expressions as part of the standard libraries, so there is no need for additional downloads. Below are some tables with metacharacters explaining the regular expressions used.

Table 1

Meta symbols to find lines or text borders

Meta symbols	Purpose
^	Beginning of line
$	End of line
\b	The boundary of words
\B	The boundary of “no words”
\A	Beginning of input
\G	End of previous match
\Z	End of input
\z	End of input

Table 2

Meta symbols for finding symbols classes

Meta symbols	Purpose
\d	Digital symbol
\D	Non digital symbol
\s	Space bar
\S	Non space bar
\w	A digital symbol, either alphabetic or underscore
\W	Any symbol other than those described in the previous case
.	Any symbol

Table 3

Meta symbols to search for symbols that edit text

Meta symbols	Purpose
\t	The symbol of tab
\n	The symbol of new line

\r	The symbol of carriage return
\f	The symbol for going to a new page
\u0085	The symbol for the next line
\u2028	The symbol for line breaks
\u2029	The symbol for a paragraph section

Table 4

Meta symbols for grouping symbols

Metacharacters	Purpose
[a B C]	Symbols a, b, or c
[^abc]	Any symbols except a, b, c
[a-zA-z]	Combine ranges (Latin symbols from a to z are case insensitive)
[ad[mp]]	Concatenation of symbols from a to d and from m to p
[az&&[def]]	Intersection of symbols (symbols d, e, f)
[az&&[^bc]]	Subtracting symbols (a, d, z)

Table 5

Meta symbols for indicating the number of symbols are quantifiers. Quantifiers always follow a symbol or group of symbols

Meta symbols	Purpose
?	The symbol occurs once or does not occur
*	The symbol does not appear or appears several times
+	The symbol appears once or more times
{n}	The symbol appears n times
{N}	The symbol appears N times or more
{n, m}	The symbol appears at least n times but no more than m times

A more detailed description of regular

expressions can be found online, but the tables above are sufficient to explain the process.

Here are the regular expressions to find the strings we are interested in:

"(\d+\s+)?ISSN\s\d+-\d+\..+\r\n" – to search for the footer.

"©\s.+\d+\r\n" – to search for copyright information.

Parsing regular expressions

The first thing you should pay attention to is the double "" symbol.

One such symbol is enough in a regular expression entry, but in Java, it must be doubled because "" is a service symbol.

A regular expression to find the footer:

(\\d+\\s+)?ISSN\\s\\d+-\\d+\\.+\r\n

Let's analyze it in more detail:

\d+ – finds one or more numeric symbols (for example, a page number).

\s+ – checks for a space after the page number.

? – indicates that the previous part (page number) is optional.

ISSN\s\d+-\d+\ – a fixed part of the string (ISSN + number with a hyphen).

... - any symbols after ISSN (for example, page number).

\r\n – a new line (newline).

This expression finds all occurrences of footers in our text.

A regular expression to search for copyright:

©\\s.+\\d+\r\n

©\s – finds the copyright symbol "©" and a space after it.

.+ – any symbols after © (usually the names of authors).

\d+ – year of publication.

\r\n – new line.

After finding all occurrences of these strings, the program deletes them.

The next step is to split the text into sentences. Why is the word "sentence" in quotation marks? Because it is impossible to automatically split the text into sentences with 100% accuracy. However, this step will make further processing much easier.

To perform this task, NLP (Natural Language Processing) tools are used: OpenNLP, Stanford NLP, LingPipe, GATE.

Why did we choose Stanza[24]?

Support for many languages, including Ukrainian.

- Comprehensive text analysis: sentence and word breakdown, morphological analysis, parsing, etc.
- High accuracy of results.

How does Stanza work? According to the official website:

Stanza is a set of accurate and efficient tools for linguistic analysis. It can split text into sentences and words, recognize parts of speech and entities, perform parsing, and more.

Stanza is a Python library, so to use it, you need to configure the environment accordingly.

The program code for splitting text into sentences might look like this:

```
import stanza
stanza.download('uk')  # Download the model for the Ukrainian language m
nlp = stanza.Pipeline(lang='uk', processors='tokenize')
text = " This is where your text should be processed"
doc = nlp(text)
for sentence in doc.sentences:
print(" ".join([word.text for word in sentence.words])).
```

This example is written in Python.

This completes the first stage of the conversion program.

2.2 Second stage

Templates for PDF files. The general idea of search templates

So, after getting the text of an article from a PDF file, the next step is to parse it. Parsing means processing the text in such a way that its result can be used to build an OWL file. This processing can be done by applying a search pattern. A template is a separate XML file created before the conversion program is launched. A particular template can only be applied to the type of article for which it was created; each type requires its own template. By article type, we mean a clear structure of the article required by the journal and followed by the authors. Obviously, with several templates, it is possible to process a certain number of articles. Since journals rarely change the format, you can create several templates in advance to process different articles. Formally speaking, a search template is a "hint" for a conversion program. This "hint" allows the program (or script) to find and separate specific parts of the article.

- Information about the authors;
- Title of the article;
- UDC;
- Keywords;
- Introduction;
- Section headings;
- Section sentences;
- Conclusions;
- References.

You can immediately answer the question of why this separation is needed at all. The fact is that the program "has no idea" about the text it receives from the PDF file at the initial stage of conversion, for the program this text looks like a single text string. At the same time, depending on how the PDF file was created, this line may contain information that is not formally related to the article itself, such as the page numbers in the journal on which the article is published, which creates additional difficulties during conversion. If we have just a monolithic text string, the program cannot create an OWL file from it that contains all the above-mentioned article entities, so it is necessary to search for and separate these entities. So, we need a mechanism by which the program can get more detailed information about the text string. This mechanism is the use of search patterns. It should be noted right away that creating a hint template is a manual process. The person who creates the template must find the necessary positions of the parts of the article, as well as their features. The latter will help distinguish them from any other entities in the article. The above is explained in more detail in the detailed analysis of templates.

Detailed analysis of search templates

Just as with header and footer search and copyright information, we use regular expressions to find and separate the information we need. But unlike searching the entire text, there is a pattern that "tells" us where and what entities are located. So, we know that the first "sentence" in our example is the line "UDC 004.05, 004.42". Having analyzed this string, we can say that it also has a fairly clear structure. It is possible to use a more rigid regular expression to select this string, but the template indicates that it is enough to simply select everything in the first line up to the characters of the new line. The program code might look like this:

UDC = currentSentence.substring(0,currentSentence.indexOf("\n"));

The template also indicates which entity is contained in this line. An example of a template string may look like this:

```
<entity id=1>
<name>UDC</name>
```

```
<pattern>.+\r\n </pattern>
</entity>
```

The name indicates the entity to be found, and the pattern is a regular expression that can be used to select this entity. The next entity seen in our example is the list of authors. It is known that an article will at least have one author, but it can also have several co-authors. It is also known that the author information takes three lines, and the last line contains the substring “e-mail:”

O.V. PALAGIN
Glushkov Institute of Cybernetics, National Academy of Sciences of Ukraine, Kyiv, Ukraine,
e-mail: oleksandr.palagin_a@ukr.net.

This information allows us to separate the “author” entity, including all information about the author. The program code may look like this:

```
Authors.add(currentSentence.substring(0, currentSentence.indexOf(",")));
        currentSentence = currentSentence.substring(currentSentence.indexOf(",")+1);
        while (true)
        {
           if (currentSentence.contains(","))
           {
              Authors.add(currentSentence.substring(0, currentSentence.indexOf(",")));
              currentSentence = currentSentence.substring(currentSentence.indexOf(",")+1);
           }
           else
           {
              String tmp = currentSentence.substring(0, currentSentence.indexOf("\n"));
              currentSentence = readFromInputStream(br);
              Authors.add(tmp + " " + currentSentence.substring(0, currentSentence.indexOf("\n")));
              currentSentence = currentSentence.substring(currentSentence.indexOf("\n")+1);
              break;
           }
        }
```

This code uses the simplest way to search for the “author” entity. It works because the article has a clear structure that is described in the template. The description of this part of the text in the template may look like this:

```
<entity id=2>
<name>Authors</name>
<pattern> </pattern>
</entity>
```

The *pattern* field is empty because the regular expression is not used in this example.

So, you can search for strings that match the following regular expression: +e-mail:+\r\n. Also, since there may be several authors, you need to apply this pattern until you find all the information about the authors. When you have finished processing the information about the authors, the next “sentence” will be the *title of the article*. You can use the .+\r\n pattern again. You may also need to adjust these simple patterns in the future to increase the search accuracy, but for our example, these expressions are sufficient.

The program code for separating the title of an article may look like this:

```
Title = currentSentence.substring(0, currentSentence.indexOf("Анотація")-1);
        while (true)
        {
           currentSentence = readFromInputStream(br);
           if (currentSentence.contains("Ключові слова:"))
              break;
```

```
        else
            AnnotaionSentences.add(currentSentence);
    }.
```

The title of the article is followed by the abstract, which consists of the line “Abstract” and a certain number of sentences following this line. You can use an expression to search for the “Abstract” line, for example, “^Abstract.\s”. In order to make sure that it is this part of the article. (In the case of this example, it is enough that we know the position of this part of the article). This allows us to remove sentences of the annotation until we get a line containing “Keywords:”. Once we have this line, the next symbols are the keywords. We separate the keywords until we come across the line “INTRODUCTION”. Under this line, you can apply a pattern that checks for case. If you need additional control, when you find this line, keep the sentence until you reach the heading of the next section. To check if the section heading has been found, you can use the following expression “”^[\\p{Lu}\\s]+\\n“”. It searches for punctuation, either uppercase letters or spaces, and the search for a combination of these symbols is repeated until we encounter a newline symbol. With this regular expression, it is possible to find the text of all the headings in our article. Once a heading is found, the sentences related to that heading are stored. And in this way, the entire text is iterated through until all the entities that should be included in the OWL file are selected.

2.3 Third stage

Creating an OWL file

So, we have all the components to create an OWL file. At this stage, all we need to do is to use a tool that will allow us to create and manipulate ontologies. You can use the OWL API [25] or the Jena Ontology API [26]. For our example, we have chosen *the Jena Ontology API* library. In order to add this library to our project, it is enough to add the following lines to the dependency file, if we are using *Maven*:

```
<dependency>
  <groupId>org.apache.jena</groupId>
  <artifactId>apache-jena-libs</artifactId>
  <type>pom</type>
  <version>X.Y.Z</version>
</dependency>
```

Next, use the API of this library to create an OWL file. The program code may look like this:

```
OntModel m = OntModelFactory.createModel(OntSpecification.OWL2_FULL_MEM);
m.createObjectProperty(NS + propertie);
OntClass article = m.createOntClass( NS + "Статья" );
article.addSubClass(m.createOntClass(NS + cls));
…
```

As a result of the above steps, an OWL file with the required entities is obtained. The above describes the step-by-step creation of an OWL file from a PDF file, as well as a detailed analysis of each step of the algorithm.

As a result of the aforementioned steps, an OWL file containing the required entities is obtained. The presented approach outlines the step-by-step generation of an OWL file from a PDF file, along with a detailed analysis of each stage of the algorithm.

The *UML* diagram of the *activity* is shown in Fig. 4.

3. Conclusion

In this paper, we address the problem of automating the processing of scientific publications using Semantic Web technologies. The proposed Knowledge-Related System for processing scientific

publications leverages knowledge graphs for the structured representation of scientific information. The paper describes the processes and procedures of two main modifications that enable the system to operate both in a local network and via remote access. The automation of OWL file generation from PDF documents is divided into three key stages: text extraction and cleaning, entity selection, and OWL file generation according to the predefined structure. The results of the study confirm the effectiveness of utilizing knowledge graphs and the OWL format for structuring scientific publications. Future research may focus on enhancing the mechanisms for automatic PDF processing, expanding the range of supported formats, and optimizing the generation of SPARQL queries to improve access to scientific information.

Several challenges remain in the development of automated knowledge processing, particularly in the role of ontology engineering in shaping the knowledge industry. The structured organization and semantic relationships within knowledge graphs facilitate intelligent information retrieval and integration, supporting advancements in various scientific domains. Additionally, the design of a modern circuitry-type processor distinguishes itself through its ability to optimize processing efficiency while maintaining flexibility for evolving computational demands. Another important consideration is the transfer of an information structures interpreter to PLD-implementation, which necessitates careful adaptation of algorithmic methods to ensure efficient and scalable hardware execution [27, 28].

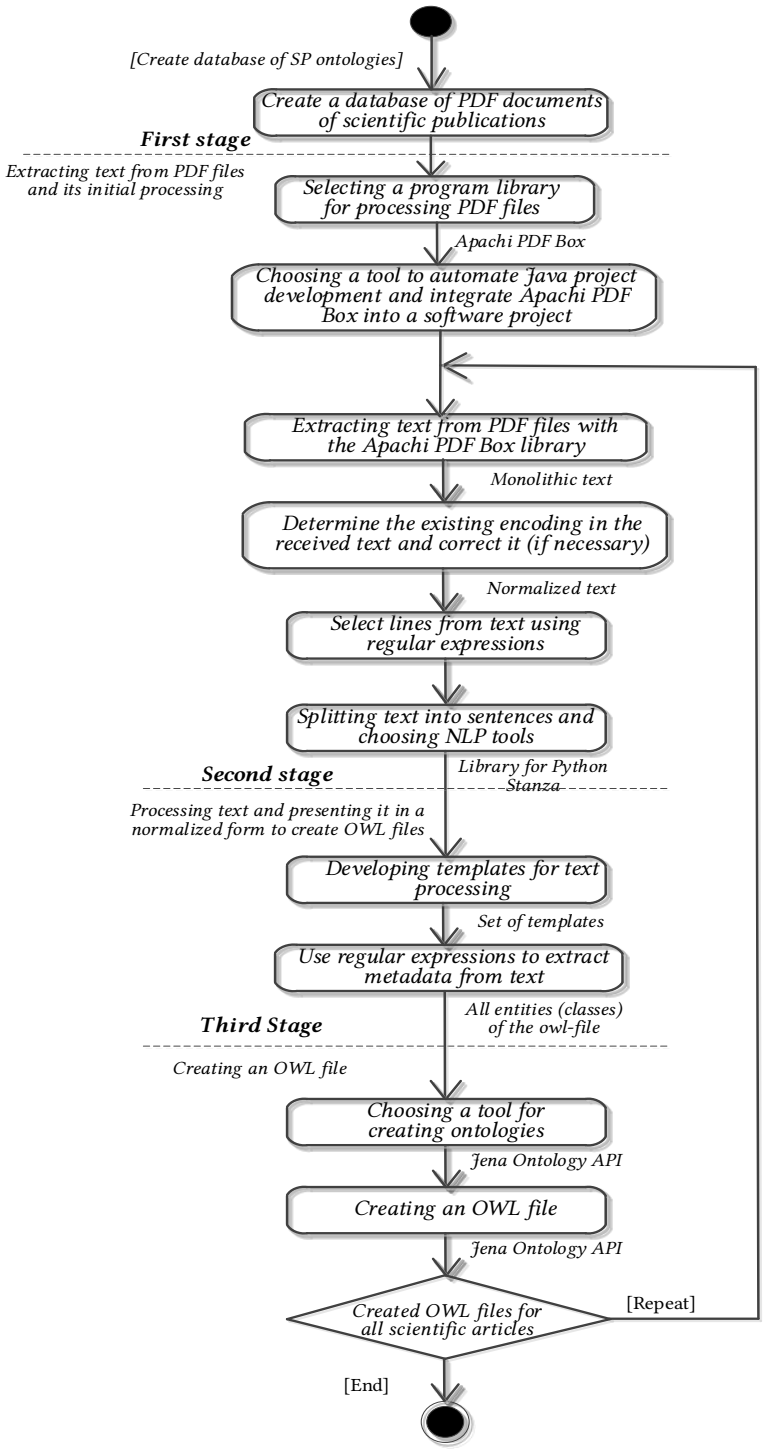

Figure 4: Diagram of the activity of creating SP ontologies

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] "OpenAI: GPT-4 Technical Report," arXiv:2303.08774 [cs.CL] 15 Mar (2023). https://arxiv.org/html/2303.08774v6

[2] A. Hogan, C. Gutierrez, M. Cochez, G.D. Melo, S. Kirrane, A. Polleres, R. Navigli, A.-C.N. Ngomo, S. M. Rashid, L. Schmelzeisen, S. Staab, E. Blomqvist, C. d'Amato, J. E .L. Gayo, S.

Neumaier, A. Rula, J. Sequeda, A. Zimmermann, Knowledge Graphs. Springer International Publishing, 2022. doi.org/10.2200/s01125ed1v01y202109dsk022.
[3] O. Palagin, M. Petrenko, K. Malakhov, Challenges and Role of Ontology Engineering in Creating the Knowledge Industry: A Research-Related Design Perspective, Cybern. Syst. Anal. (2024). doi.org/10.1007/s10559-024-00702-6
[4] K. S. Malakhov, Insight into the Digital Health System of Ukraine (eHealth): Trends, Definitions, Standards, and Legislative Revisions. Int. J. Telerehabilitation 15(2), (2023) 1–21. doi.org/10.5195/ijt.2023.6599.
[5] K. S. Malakhov, Letter to the Editor – Update from Ukraine: Development of the Cloud-based Platform for Patient-centered Telerehabilitation of Oncology Patients with Mathematical-related Modeling. Int. J. Telerehabilitation 15(1), (2023) 1–3. doi.org/10.5195/ijt.2023.6562.
[6] K. S. Malakhov, Editorial for the Special Issue of the International Journal of Telerehabilitation. Int. J. Telerehabilitation, Special Issue: Research Status Report – Ukraine in 2024, (2024) 1–2. doi.org/10.5195/ijt.2024.6639.
[7] K. S. Malakhov, Innovative Hybrid Cloud Solutions for Physical Medicine and Telerehabilitation Research. Int. J. Telerehabilitation 16(1), (2024) 1–19. doi.org/10.5195/ijt.2024.6635.
[8] K. S. Malakhov, Letter to the Editor: Advancements in Digital Health Technologies. S. Afr. Comput. J. 36(1), Article 1 (2024). doi.org/10.18489/sacj.v36i1.18942.
[9] K. S. Malakhov, Letter to the Editor – Update from Ukraine: Project Results in Oncology Telerehabilitation Approved at the National Cancer Institute and Showcased at the 4th National PM&R Congress. Int. J. Telerehabilitation 16(2) (2025). doi.org/10.5195/ijt.2024.6686.
[10] M. Petrenko, O. Palagin, M. Boyko, S. Matveyshyn, Knowledge-oriented tool complex for developing databases of scientific publications and taking into account Semantic Web technology. Control Syst. and Comput. 3.299, (2022) 11–28. doi.org/10.15407/csc.2022.03.011.
[11] O. Palagin, M. Petrenko, Knowledge-oriented tool complex processing databases of scientific publications. Control Syst. and Comput. 5(289), (2020) 17–33. doi.org/10.15407/csc.2020.05.017.
[12] O. V. Palagin, M. H. Petrenko, M. O. Boyko, Ontology-related Complex for Semantic Processing of Scientific Data, in: Proceedings of the 13th International Scientific and Practical Programming Conference UkrProg 2022. Kyiv, Ukraine, 2022. URL: http://ceur-ws.org/Vol-3501/s26.pdf.
[13] Simon, Brown. The C4 model for visualizing software architecture. http://leanpub.com/visualising-software-architecture
[14] https://app.diagrams.net/#DConcept_KrS.drawio#%7B%22pageId%22%3A%22IVSPsZGNb81UvTkcKvhU%22%7D
[15] https://app.diagrams.net/#G1kx0Zpd8UXUTFv7CeKhGxrUoTM_lfxTb1#%7B%22pageId%22%3A%22QJheQnxjksa_kTp4ZTIs%22%7D
[16] G., Wierda, MasteringArchiMateEdition3.2. Published by R&A, The Netherlands, 2024. https://ea.rna.nl/the-book-edition-iii/
[17] ArchiMate User Guide. https://www.archimatetool.com/downloads/archi/Archi%20User%20Guide.pdf
[18] https://pdfbox.apache.org/
[19] O. Palagin, M. Petrenko, A. Litvin, M. Boyko, Method of Developing an Ontological System with Automatic Formation of a Knowledge Base and User Queries, Proceedings of the 14th International Scientific and Practical Programming Conference UkrPROG 2024. Kyiv, Ukraine, May 14–15, 2024. URL: https://ceur-ws.org/Vol-3806/S_2_Palagin.pdf
[20] https://undatas.io/blog/posts/hands-on-comparison-of-open-source-pdf-parsing-tools-apache-pdfbox-pypdf2-pdfminer-camelot-etc./
[21] B. Lundell, C. Brax, Maintaining Interoperability in Open Source Software: A Case Study of the Apache PDFBox Project, Journal of Systems and Software, October, 2019. URL: https://www.researchgate.net/publication/336731884_Maintaining_Interoperability_in_Open_Source_SoftwareA_Case_Study_of_the_Apache_PDFBox_Project

[22] D. Maynard, O. Hamza, T. Mcenery T. P. Bakert, A Unicode-based Environment for Creation and Use of Language Resources, August, 2002. URL:https://www.researchgate.net/publication/2524865_A_Unicode-based_Environment_for_Creation_and_Use_of_Language_Resources
[23] M. Erwig, R. Gopinath, Explanations for Regular Expressions, Lecture Notes in Computer Science 7212:394-408, March, 2012. URL:https://www.researchgate.net/publication/262402240_Explanations_for_Regular_Expressions
[24] https://stanfordnlp.github.io/stanza/
[25] http://owlcs.github.io/owlapi/
[26] https://jena.apache.org/documentation/ontology/
[27] M. Petrenko, O. Kurgaev, Distinguishing features of design of a modern circuitry type processor. Upravlyayushchie Sistemy i Mashiny 5 (2003) 16–19. URL: https://www.scopus.com/record/display.uri?eid=2-s2.0-0347622333&origin=resultslist
[28] M. Petrenko, A. Sofiyuk, On one approach to the transfer of an information structures interpreter to PLD-implementation, Upravlyayushchie Sistemy i Mashiny 6 (2003) 48–57. URL: https://www.scopus.com/record/display.uri?eid=2-s2.0-0442276898&origin=resultslist