Documentation for Decision Support System in Software Management

Nicolas Pinto - Student ID: 807348

A.Y. 2023-2024

Abstract

This project presents the development of a decision support system tailored for software management within a software house. The system leverages ontologies to structure and represent knowledge, utilizing GraphBRAIN for ontology translation and Neo4j for data storage. A custom software ontology was created, integrating with existing ontologies and populated using data extracted from Wikidata via SPARQL queries. The knowledge base was further translated into Prolog, enabling the creation of rules to facilitate decision-making. This documentation outlines the methodology, tools, and processes used, and highlights the system's potential applications in real-world software management scenarios.
Link to the project repository: GitHub repo.

1 Introduction

1.1 Project Overview

In the landscape of software development, effective management of software and its associated knowledge is paramount for any software house aiming to maintain competitiveness. This project seeks to address this need by developing a *decision support system* tailored for software management, leveraging tools and methodologies in the field of artificial intelligence and knowledge representation.

The core of this project revolves around the creation of a custom ontology designed to model and manage the complexities of software engineering. The ontology is built using Gr@phBRAIN, a platform for managing interconnected knowledge, and is integrated with existing ontologies to ensure comprehensive coverage and reuse of established knowledge. To store and query this ontology efficiently, Neo4j[1], a graph database, is employed, facilitating complex relationships and data retrieval tasks. Additionally, the knowledge base is translated into Prolog, enabling the creation of intelligent rules and queries that can support decision-making processes within a software house.

1.2 Objectives

The primary objective of this project is to develop a robust decision support system that enhances the ability of software houses to manage and leverage software-related knowledge effectively. Specifically, the project aims to:

- **Create and Integrate a Custom Ontology**: Develop a software management ontology that integrates with existing ontologies in GraphBRAIN, ensuring the reuse of established knowledge and extending it with new, relevant concepts.
- **Populate the Ontology with Relevant Data**: Use external data sources, such as Wikidata[2], to populate the ontology with real-world software instances and their associated attributes and relationships.
- **Translate the Knowledge Base into Prolog**: Convert the ontology into a Prolog-compatible format and create useful rules that simulate realistic decision-making scenarios within the context of software management.

[1] https://neo4j.com/
[2] https://www.wikidata.org/wiki/Wikidata:Main_Page

- **Provide Useful Decision-Making Tools**: Develop a set of Prolog rules and queries that are specifically designed to address common decision-making needs within a software house, ensuring that the system is both practical and applicable to real-world scenarios.

This documentation provides a detailed account of the methodologies, tools, and processes used to achieve these objectives, along with the results and implications of the project.

2 Background and Related Sources

2.1 Ontology in Software Engineering

Ontologies have become a fundamental tool in software engineering, especially in the context of knowledge representation, decision support, and automated reasoning. An ontology, in the realm of artificial intelligence and knowledge engineering, is a formal representation of a set of concepts within a domain and the relationships between those concepts. In software engineering, ontologies help in structuring complex information, facilitating interoperability between systems, and enabling advanced querying and reasoning capabilities.

An ontology can serve as a structured knowledge base that encapsulates various aspects of software, such as development processes, programming languages, software types, and the relationships between these entities. This structured approach not only enhances the ability to manage and query software-related knowledge but also supports decision-making processes by providing a clear, organized framework for reasoning about software entities and their interactions.

2.2 Tools and Related Sources

2.2.1 GraphBRAIN

GraphBRAIN is a comprehensive framework for knowledge graph (KG) management and exploitation, integrating the efficiency of leading graph database technologies with the semantic rigor of ontologies. Unlike traditional Semantic Web approaches, which rely on the atomic RDF triple model, GraphBRAIN employs a Labelled Property Graph (LPG) model [1], allowing attributes on both nodes and edges. This enhances expressiveness and compactness, while separating the ontology schema from the instance data stored in a Neo4j graph database. GraphBRAIN supports the import and integration of existing ontologies, semantic reasoning, and advanced graph analysis, making it highly versatile for applications requiring complex knowledge representation and automated reasoning.

2.2.2 Neo4j

Neo4j is central to storing and processing instances in GraphBRAIN, using the Labelled Property Graph (LPG) model[1]. It offers native graph storage and processing, enhanced by index-free adjacency for efficient traversal. While schema-optional, Neo4j supports UNIQUE constraints and is ACID compliant, ensuring data integrity.

Cypher, Neo4j's SQL-like query language, and its strong UI, the Neo4j Data Browser, make data interaction intuitive. The platform provides broad API access, including REST, Java, and JavaScript options, and supports sharding for distributed data management. Neo4j also enables exporting data to JSON and Excel, enhancing flexibility and interoperability within the project.

2.2.3 Prolog

Prolog is a logic programming language associated with artificial intelligence and computational linguistics. It is particularly well-suited for tasks that involve pattern matching, tree-based data structuring, and automated reasoning. By translating the ontology into Prolog, the project leverages Prolog's strengths in rule-based reasoning, enabling the creation of intelligent queries and rules that can assist in decision-making within a software house.

2.2.4 Related Sources

The conceptualization of the software ontology in this project was heavily informed by existing research and frameworks in software engineering. Two key resources were particularly influential:

- **A Conceptual Model of Software Development** (Chapter 2)[3]. This work provided a comprehensive model of software development, offering insights into the various stages, roles, and artifacts involved in software projects. The model's emphasis on the interconnectedness of these elements was instrumental in shaping the structure of the custom ontology.
- **SEON: Software Engineering Ontology Network**[4]. SEON is a collection of ontologies specifically designed for software engineering, covering various subdomains such as requirements engineering, design, and maintenance. SEON's approach to structuring software engineering knowledge provided valuable guidance in the development of the custom ontology, ensuring that it aligned with established best practices in the field.

Together, these tools and resources provided a solid foundation for the creation of a robust, integrated ontology for software management.

3 Methodology

The development of the decision support system for software management involved several key steps, including knowledge engineering, ontology creation, and data management. Below is a detailed account of the methodology used in this project.

3.1 Ontology Translation and Creation

The initial phase of the project involved translating existing XML ontologies into a format compatible with GraphBRAIN, using the specifications outlined in the GraphBRAIN Scheme (GBS) Specifications v1.0 document. This translation process was critical in ensuring that the ontologies could be effectively integrated and used within the GraphBRAIN framework.

Subsequently, I conducted a comprehensive study of the software engineering domain, drawing on key sources such as *A Conceptual Model of Software Development* and the *SEON*. These resources provided foundational concepts that informed the development of a new ontology tailored for software management. This ontology, named `software_Pinto.gbs`, was designed to be fully compatible with existing ontologies in GraphBRAIN, specifically importing and integrating with `general.gbs` and `retrocomputing.gbs`. A conceptual schema, which visually represents the ontology with a legend indicating the origin of entities and relationships - particularly those inherited from other ontologies - was also developed and is and is shown in Figure 1 for reference (clicking the caption will take you to the readable file).

A list of the newly introduced entities and their descriptions is given in the Appendix A.

3.2 Ontology Integration and Inconsistency Resolution

After creating the `software_Pinto.gbs` ontology, it was necessary to integrate it with the existing knowledge base in GraphBRAIN. This process involved uploading the ontology to Neo4j using the `BatchUploadTest.java` script. During this phase, several inconsistencies were identified and resolved. One notable example was the reclassification of the entity *ProgrammingLanguage* from a specialization of *Software* to a specialization of *IntellectualWork*, aligning with the existing structure in general.gbs.

3.3 Data Population and Instance Creation

With the ontology structure in place, the next step was to populate it with relevant data. This was achieved by querying Wikidata, a vast open-source database, using SPARQL query in Listing 1 via the SPARQLWrapper library in Python. The query returned a JSON file containing detailed

[3] https://philippe.kruchten.com/wp-content/uploads/2012/07/kruchten-2007-conceptual_model.pdf
[4] https://dev.nemo.inf.ufes.br/seon/

information about 3027 software entities, including attributes such as developer, publication date, and programming language. An example of result is shown in Listing 2.

```
SELECT DISTINCT ?softwareLabel
                ?subInstanceLabel
                ?developerLabel
                ?subclassOfLabel
                ?partOfLabel
                ?publicationDate
                ?distributedByLabel
                ?operatingSystemLabel
                ?programmedInLabel
                ?copyrightLicenseLabel
                ?editionLabel
                ?editionPartOfLabel
                ?editionPublicationDate
                ?followsLabel
                ?editionProgrammedInLabel
                ?editionCopyrightLicenseLabel
WHERE {
  # Trova tutte le istanze di software di produttivita'
  ?software wdt:P31 wd:Q17155032.

  VALUES ?developer { wd:Q2283 wd:Q11463 } # Microsoft e Adobe

  # Trova le sotto-istanze di ciascun software di produttivita' e sviluppatore
  ?subInstance wdt:P31 ?software;
               wdt:P178 ?developer.

  # Ottieni gli attributi aggiuntivi
  OPTIONAL { ?subInstance wdt:P279 ?subclassOf. }
  OPTIONAL { ?subInstance wdt:P361 ?partOf. }
  OPTIONAL { ?subInstance wdt:P577 ?publicationDate. }
  OPTIONAL { ?subInstance wdt:P750 ?distributedBy. }
  OPTIONAL { ?subInstance wdt:P306 ?operatingSystem. }
  OPTIONAL { ?subInstance wdt:P277 ?programmedIn. }
  OPTIONAL { ?subInstance wdt:P275 ?copyrightLicense. }

  # Per ogni edizione o traduzione ottieni data di pubblicazione e follows
  OPTIONAL {
    ?subInstance wdt:P747 ?edition.
    OPTIONAL { ?edition wdt:P577 ?editionPublicationDate. }
    OPTIONAL { ?edition wdt:P155 ?follows. }
    OPTIONAL { ?edition wdt:P361 ?editionPartOf. }
    OPTIONAL { ?edition wdt:P277 ?editionProgrammedIn. }
    OPTIONAL { ?edition wdt:P275 ?editionCopyrightLicense. }
    ?edition rdfs:label ?editionLabel.
    FILTER(LANG(?editionLabel) = en)
  }

  SERVICE wikibase:label { bd:serviceParam wikibase:language en. }
}
ORDER BY ?editionLabel
```

Listing 1: SPARQL query for obtaining data from WidiData


```
{
    "softwareLabel": "application",
    "subInstanceLabel": "Microsoft Word",
    "developerLabel": "Microsoft",
    "subclassOfLabel": "word processor program",
    "partOfLabel": "Microsoft Office",
    "publicationDate": "1983-10-01T00:00:00Z",
    "distributedByLabel": "Microsoft Store",
    "operatingSystemLabel": "Mac OS operating systems",
    "programmedInLabel": "C++",
    "copyrightLicenseLabel": "end-user license agreement",
    "editionLabel": "Microsoft Office Word 2007",
    "editionPartOfLabel": "Microsoft Office 2007",
    "editionPublicationDate": "2007-01-30T00:00:00Z",
    "followsLabel": "Microsoft Word 2003"
```

```
16   },
```

Listing 2: An example of a result obtained from WikiData. Not all results come with all required attributes.

To integrate this data into the Neo4j database, a custom script (`pinto_script_kb.ipynb` in the project folder) was developed to transform the JSON results into the format required by `TestTree.java`, which is the script provided by the professor for uploading instances. The transformation process involved **mapping** each attribute from the Wikidata results to the creation of the corresponding entity attributes and relationships in the ontology:

- `?softwareLabel` maps to the **softwareType attribute** of the **Software** entity. This represents the type of software (e.g., productivity software, video editor software, etc.).
- `?subInstanceLabel` maps to the **name attribute** of the **Software** entity. This attribute represents the specific name of the software.
- `?developerLabel` maps to the **name attribute** of the **Stakeholder** entity and the **developedBy relationship** between the **Software** and **Stakeholder** entities. This attribute identifies the name of the developer (e.g., Microsoft, Adobe) who developed the software.
- `?subclassOfLabel` maps to the **softwareCategory attribute** of the **Software** entity. This represents the category to which the software belongs.
- `?partOfLabel` maps to the **name attribute** of the **Software2** entity and the **partOf relationship** between the **Software** and **Software2** entities. This represents the fact that the software is part of another software or suite.
- `?publicationDate` maps to the **publicationDate attribute** of the **Software** entity. This represents the publication date of the software.
- `?distributedByLabel` maps to the **name attribute** of the **Software3** entity and the **distributedBy relationship** between the **Software** and **Software3** entities. This represents the entity responsible for distributing the software.
- `?operatingSystemLabel` maps to the **name attribute** of the **OperatingSystem** entity and the **executableOn relationship** between the **Software** and **OperatingSystem** entities. This represents the operating system on which the software can be executed.
- `?programmedInLabel` maps to the **name attribute** of the **ProgrammingLanguage** entity and the **writtenIn relationship** between the **Software** and **ProgrammingLanguage** entities. This represents the programming language used to develop the software.
- `?copyrightLicenseLabel` maps to the **License attribute** of the **Software** entity. This represents the license under which the software is released.
- `?editionLabel` maps to the **name attribute** of the **Software4** entity and the **hasEdition relationship** between the **Software** and **Software4** entities. This represents the name of the software edition or version.
- `?editionPartOfLabel` maps to the **name attribute** of the **Software5** entity and the **partOf relationship** between the **Software4** and **Software5** entities. This represents the fact that an edition of the software is part of another software.
- `?editionPublicationDate` maps to the **presentationDate attribute** of the **Software4** entity. This represents the publication date of the specific software edition.
- `?followsLabel` maps to the **name attribute** of the **Software6** entity and the **follows relationship** between the **Software4** and **Software6** entities. This represents the software or edition that precedes the current one.

- `?editionProgrammedInLabel` maps to the **name attribute** of the **ProgrammingLanguage2** entity and the **writtenIn relationship** between the **Software4** and **ProgrammingLanguage2** entities. This represents the programming language used to develop the specific edition of the software.

- `?editionCopyrightLicenseLabel` maps to the **license attribute** of the **Software4** entity. This represents the license under which the specific software edition is released.

As anticipated, the result of this transformation is a file containing entities and relationships defined according to the formalism required by the `TestTree.java` script. For the example in Listing 2, therefore, we get the result in Listing 3 for the entities and in Listing 4 for the relationships involved.

Once all the data were uploaded into the Neo4j database, the graph in the Figure 2 was obtained.

```
{"jtype": "node", "identity": 0, "label": "Software", "properties": {"name": "
    Microsoft Word", "license":  "end-user license agreement", "softwarType":  "
    application", "softwarCategory":  "word processor program", "presentationDate": "
    1983-10-01"}}
{"jtype": "node", "identity": 1, "label": "Stakeholder", "properties": {"name": "
    Microsoft"}}
{"jtype": "node", "identity": 2, "label": "Software", "properties": {"name": "
    Microsoft Office"}}
{"jtype": "node", "identity": 3, "label": "Software", "properties": {"name": "
    Microsoft Store"}}
{"jtype": "node", "identity": 4, "label": "OperatingSystem", "properties": {"name": "
    MacOS operating system"}}
{"jtype": "node", "identity": 5, "label": "ProgrammingLanguage", "properties": {"name"
    : "C++"}}
{"jtype": "node", "identity": 6, "label": "Software", "properties": {"name": "
    Microsoft Office World 2007", "presentationDate":"2007-01-30"}}
{"jtype": "node", "identity": 7, "label": "Software", "properties": {"name": "
    Microsoft Office 2007"}}
{"jtype": "node", "identity": 8, "label": "Software", "properties": {"name": "
    Microsoft Word 2003"}}
```

Listing 3: Entities created from the result In Listing 2

```
{"jtype": "relationship", "subject": 0, "object": 1, "name": "developedBy", "
    properties": {}}
{"jtype": "relationship", "subject": 0, "object": 2, "name": "partOf", "properties": {
    }}
{"jtype": "relationship", "subject": 0, "object": 3, "name": "distributedBy", "
    properties": {}}
{"jtype": "relationship", "subject": 0, "object": 4, "name": "executableOn", "
    properties": {}}
{"jtype": "relationship", "subject": 0, "object": 5, "name": "writtenIn", "properties"
    : {}}
{"jtype": "relationship", "subject": 0, "object": 6, "name": "hasEdition", "properties
    ": {}}
{"jtype": "relationship", "subject": 6, "object": 7, "name": "partOf", "properties": {
    }}
{"jtype": "relationship", "subject": 66, "object": 8, "name": "follows", "properties":
     {}}
```

Listing 4: Relationships created from the result In Listing 2

3.4 Knowledge Base Translation and Rule Creation

Following the population of the ontology in Neo4j, the knowledge base was translated into Prolog using the `TestTree.java` script. This translation enabled the application of logic-based reasoning to the data, which is crucial for the decision support system.

The final step in the methodology involved creating Prolog rules tailored to the specific needs of the case study. These rules were designed not only to demonstrate coverage of key topics discussed during the course but also to provide practical, user-oriented queries that could support decision-making in software management. The rules were thoroughly documented to ensure clarity and ease of use for future users of the system.

4 Results

4.1 Ontology Structure

The `software_Pinto.gbs` ontology is a comprehensive conceptual model aimed at capturing the complexities of the software engineering domain, particularly focusing on software as a product. This ontology integrates multiple entities and relationships to provide a structured representation of the software lifecycle, stakeholder roles, and software types, making it a powerful tool for knowledge management and decision support.

One of the main challenges in designing this ontology was balancing the need for detailed classification with the practical limitations of available data, particularly from sources like Wikidata. To address this, the ontology was designed to be both extensible and flexible, allowing for the integration of additional data and the refinement of existing entities and relationships.

4.2 Prolog Rules and Queries

To enhance the functionality of the decision support system, several Prolog rules and queries were implemented to enable more sophisticated reasoning over the knowledge graph. These rules were designed to extract meaningful insights from the structured data and are documented below.

- `software_of_type/2`
  - **Description:** This predicate checks or retrieves the type of a software entity.
  - **Key Features:**
    - **Built-in Predicates:** `member/2` is used to check if a software's `name` and `softwareType` are present in the node's properties.
  - **Example Usage:**

```
?- software_of_type('Windows Live Messenger', 'instant messaging client').
```

    Expected Output: `true` if the software matches the type.

- `software_of_single_type/2`
  - **Description:** A variant of `software_of_type/2` that returns only the first matching result.
  - **Key Features:**
    - **Cut Operator (`!`):** Used to limit the query to the first solution, preventing backtracking after a match is found.
  - **Example Usage:**

```
?- software_of_single_type('Windows Live Messenger', Type).
```

    Expected Output: `Type = 'instant messaging client'` (or the first matching type).

- `software_with_license/2`
  - **Description:** This predicate checks or retrieves the license type associated with a software entity.
  - **Key Features:**
    - **Built-in Predicates:** `member/2` is used to verify the presence of `name` and `license` properties within the software's node properties.
  - **Example Usage:**

```
?- software_with_license('Windows Live Messenger', 'freeware').
```

    Expected Output: `true` if "Windows Live Messenger" is associated with the "freeware" license.

- `software_of_type_with_license/3`

- **Description:** This predicate checks whether a software has a specific type and license by combining `software_of_type/2` and `software_with_license/2`.
- **Key Features:**
  - **Predicate Composition:** Combines the results of two existing predicates to achieve the desired result.
- **Example Usage:**

```
?- software_of_type_with_license('Windows Live Messenger', 'instant messaging
    client', 'freeware').
```

Expected Output: `true` if the software matches both the type and license.

- `release_year_before_2000/1`
  - **Description:** This predicate checks if a given release date is before the year 2000.
  - **Key Features:**
    - **Built-in Predicates:** Uses `split_string/4` to extract the year and `atom_number/2` to convert it to a number.
    - **Simple Arithmetic:** Compares the year to 2000.
  - **Example Usage:**

```
?- release_year_before_2000('1999-07-22').
```

Expected Output: `true` if the release year is before 2000.

- `software_related_to_stakeholder/2`
  - **Description:** This predicate checks if a software is related to a specific stakeholder by verifying the `developedBy` relationship.
  - **Key Features:**
    - **Graph Traversal:** Uses the `arc/4` predicate to traverse relationships in the graph.
  - **Example Usage:**

```
?- software_related_to_stakeholder('Adobe Photoshop', 'Adobe').
```

Expected Output: `true` if the software is developed by the stakeholder.

- `software_related_to_single_stakeholder/2`
  - **Description:** This variant of `software_related_to_stakeholder/2` retrieves only the first matching stakeholder.
  - **Key Features:**
    - **Cut Operator (`!`):** Limits the result to the first matching relationship, preventing further backtracking.
  - **Example Usage:**

```
?- software_related_to_single_stakeholder('Adobe Photoshop', Stakeholder).
```

Expected Output: The first stakeholder related to the software (e.g., `Stakeholder = 'Adobe'`).

- `obsolete_adobe_software/1`
  - **Description:** Finds Adobe software released before the year 2000.
  - **Key Features:**
    - **Setof/2:** Collects all software names meeting the criteria.
    - **Predicate Composition:** Combines `software_related_to_stakeholder/2` and `release_year_before_20`
  - **Example Usage:**

```
?- obsolete_adobe_software(Software).
```

Expected Output: The names of obsolete Adobe software.

- `non_obsolete_software/1`
  - **Description:** Checks if software was released in or after the year 2000.
  - **Key Features:**
    - **Negation as Failure (\+):** Uses negation to filter out software released before 2000.
  - **Example Usage:**

```
?- non_obsolete_software('Clipchamp').
```

Expected Output: `true` if the software is not obsolete.

- `software_executable_on/2`
  - **Description:** Checks if a software can be executed on a specific operating system.
  - **Key Features:**
    - **Graph Traversal:** Uses the `arc/4` predicate to find the operating system the software can run on.
  - **Example Usage:**

```
?- software_executable_on('Adobe Photoshop', 'macOS').
```

Expected Output: `true` if the software can run on the specified OS.

- `software_executable_on_single/2`
  - **Description:** A variant of `software_executable_on/2` that retrieves only the first matching result.
  - **Key Features:**
    - **Cut Operator (!):** Used to limit the query to the first matching operating system, preventing further backtracking.
  - **Example Usage:**

```
?- software_executable_on_single('Adobe Photoshop', OSName).
```

Expected Output: The first operating system that the software can run on (e.g., `OSName = 'Windows'`).

- `select_software_for_project/4`
  - **Description:** Selects software for a project based on its type, license, and compatibility with macOS.
  - **Key Features:**
    - **Conditional Execution:** Uses `->/2` (if-then) to check if the software needs to be compatible with macOS.
  - **Example Usage:**

```
?- select_software_for_project('Visual Studio Code', 'source code editor', '
    proprietary license', true).
```

Expected Output: `true` if the software meets all the criteria.

- `software_edition_of_same_language/2`
  - **Description:** Checks if two software editions are written in the same programming language.
  - **Key Features:**
    - **Setof/2:** Collects all common programming languages between the two software editions.

    - **List Comparison:** Ensures that there is at least one common language.
  - **Example Usage:**

```
?- software_edition_of_same_language('Microsoft Office', 'Microsoft Office 2021').
```

  Expected Output: `true` if both editions are written in at least one common language.

- `compatible_software_bundle/3`
  - **Description:** Checks if two software products are compatible as a bundle for a given operating system.
  - **Key Features:**
    - **Predicate Composition:** Combines multiple checks including stakeholder consistency, OS compatibility, and software type diversity.
  - **Example Usage:**

```
?- compatible_software_bundle('Adobe Authoware', 'Adobe LiveMotion', 'macOS').
```

  Expected Output: `true` if the software bundle is compatible for the specified OS.

- `compatible_software_list/2`
  - **Description:** Checks if a list of software products is compatible as a bundle for a given operating system.
  - **Key Features:**
    - **Recursive Predicates:** Uses recursion to check multiple conditions such as non-obsolescence, stakeholder consistency, OS compatibility, and type diversity across the entire list.
  - **Example Usage:**

```
?- compatible_software_list(['Microsoft Office 2001','Microsoft Office 2019','Visual Studio Code'], 'macOS').
```

  Expected Output: `true` if the software list forms a compatible bundle for the specified OS.

- `software_by_stakeholder/2`
  - **Description:** Finds all software products developed by a specified stakeholder.
  - **Key Features:**
    - **Findall/3:** Used to collect all software names associated with a stakeholder into a list.
  - **Example Usage:**

```
?- software_by_stakeholder('Microsoft', SoftwareList).
```

  Expected Output: `SoftwareList = ['Windows Smart Screen', 'Microsoft Device Emulator', ...]`.

- `languages_of_software/2`
  - **Description:** Retrieves the programming languages of the specified software products.
  - **Key Features:**
    - **Findall/3:** Used to gather all programming languages used in the provided list of software into a list.
  - **Example Usage:**

```
?- languages_of_software(['Microsoft Messaging Passing Interface', 'Adobe ImageReady'], LanguageList).
```

  Expected Output: `LanguageList = ['C++', 'C', 'C++', 'C++']`..

- `software_count_by_language_for_stakeholder/2`
  - **Description:** Counts the number of software products developed by a stakeholder for each programming language.
  - **Key Features:**
    - **Recursive Counting:** Uses recursive predicates `count_languages/3` and `update_count/3` to count occurrences of each language.
  - **Example Usage:**

```
?- software_count_by_language_for_stakeholder('Microsoft', LanguageCountList)
    .
```

Expected Output: `LanguageCountList = ['C++'-89, 'C'-33, 'Python'-14, ...]`.

5 Discussion

5.1 Challenges and Solutions

The development of the software engineering ontology presented several significant challenges, primarily due to the complexity of the domain and the limitations of the available data sources, such as Wikidata. Below, I discuss these challenges and the solutions I implemented to address them.

5.1.1 Complexity of Software Engineering Domain

The software engineering domain is inherently complex, encompassing a vast array of concepts, entities, and relationships. Many of these elements are difficult to conceptualize and represent, particularly when relying on external data sources like Wikidata, which may lack the depth or specificity required for certain aspects of the domain. To manage this complexity, the ontology primarily focuses on entities directly related to software as a product, omitting more management-oriented aspects that require information not readily available in Wikidata. This selective focus ensures the ontology remains functional and coherent, even if it sacrifices some breadth.

5.1.2 Redundancy in Instances

Due to the structure and limitations of Wikidata, there is a redundancy in how certain instances are represented in the ontology. For example, operating systems are stored both as instances of the `Software` entity with the attribute **Type** set to "operating system" and as instances of a more specific `OperatingSystem` entity. This duplication arose because Wikidata itself distinguishes these as separate entities, despite the initial conceptualization of the ontology, which viewed `OperatingSystem` as a specialization of `Software`. This redundancy was accepted to maintain compatibility with Wikidata, with the understanding that it reflects the data source's structure rather than an optimal conceptual model.

5.1.3 Entities Without Attributes

Some entities within the ontology, such as Stakeholder, have been specialized into sub-entities that currently lack any attributes. These could have been more straightforwardly represented as attribute values, but I chose to maintain them as separate entities. This decision is justified by the potential real-world application of the support system, where a company, for example, might have attributes that need to be captured. A similar rationale applies to the specializations of the Software entity, where maintaining distinct entities allows for future expansion and attribute assignment as the ontology evolves.

5.1.4 Handling Unanticipated Software Types

During the integration of data from Wikidata, it became evident that the ontology's initial design did not account for all the types of software identified in Wikidata. Specifically, Wikidata returned 110 different types of software, far exceeding the specializations initially foreseen. To address this, I

temporarily added a `Type` attribute to the Software entity to accommodate all the types provided by Wikidata. This solution allows for immediate data integration without altering the planned hierarchy, with the intention of refining and potentially reifying these types in future iterations of the ontology.

5.1.5 Distinct Entity Instances Based on Attribute Variation

The ontology treats instances as distinct entities whenever there is any variation in their attributes. For example, the knowledge base includes four different instances of Visual Studio Code, differentiated by license type or software type, despite sharing the same name and publication date. This approach ensures that all relevant variations are captured, reflecting the diverse nature of software products and their attributes. While this results in multiple instances for what might be considered the same product, it allows for a more precise representation of the data as it exists in Wikidata.

5.2 Usefulness of the Decision Support System

The Decision Support System (DSS) built using the software engineering ontology in GraphBRAIN significantly enhances the way users manage and analyze software-related data.

The system consolidates diverse information into a single, unified framework, making it easier to access and cross-reference software entities from multiple sources, such as Wikidata. This integration improves the depth and reliability of decision-making by providing a broader base of knowledge. Furthermore, by using ontologies, the DSS offers advanced semantic analysis, allowing users to uncover relationships and patterns that might otherwise go unnoticed. This leads to more informed decisions, especially in complex scenarios where understanding the connections between software products, stakeholders, and processes is crucial.

The DSS is highly adaptable, allowing users to tailor the system to their specific needs. Whether it's applying different ontologies to the same dataset or customizing queries, the system supports a wide range of use cases without requiring extensive reconfiguration. In fact, with its querying capabilities, the DSS enables users to retrieve specific information quickly and efficiently.

Designed for collaboration, the DSS allows multiple stakeholders to contribute to decision-making processes. By facilitating knowledge sharing across teams, it ensures that decisions are well-informed and consider various perspectives.

5.3 Limitations and Future Work

One of the primary limitations lies in the handling of data downloaded from Wikidata. The current approach can lead to redundancy and inconsistencies, particularly with the treatment of software types and operating systems. Future work should focus on refining the data integration process to ensure a more accurate and streamlined dataset.

The current model handles operating systems both as a type of software and as distinct entities, leading to potential overlap and redundancy. Future improvements should aim to unify these concepts, ensuring that the relationship `exeOn` is strictly maintained between software and operating system entities, avoiding duplication.

At present, the `type` attribute in the `Software` entity is used to manage various software categories, many of which are directly obtained from Wikidata. A more sophisticated approach would involve reifying these types into sub-entities within the Software entity. This would not only reduce redundancy but also allow for the integration of existing entities that may overlap with these types.

Currently, the verification of SPARQL query results is performed manually, with sample checks via GUI and rule-based methods. A more rigorous and automated formal verification process should be developed to ensure the accuracy and reliability of the data extracted from queries.

6 Conclusion

In this work, I developed a knowledge-based decision support system tailored to the software engineering domain. By leveraging ontologies and integrating data from Wikidata, we established a robust framework for managing and querying complex software-related information. The system effectively organizes software entities and their relationships, providing valuable insights for various stakeholders.

Despite its strengths, the system has limitations, particularly in data handling, redundancy, and the current approach to managing software types and operating systems. Future work will focus on refining these aspects, including better data management, reification of software types, and the implementation of formal verification processes for query results.

Overall, the system demonstrates significant potential as a tool for decision-making in software management, offering a structured approach to navigate the complexities of the field. With further improvements, it can become an even more powerful resource for supporting decisions in software engineering and related domains.

References

[1] D. Di Pierro and D. Redavid. The graphbrain framework for knowledge graph management and its applications to cultural heritage.

A Ontology Entities and Descriptions

1. **Investor**
   Stakeholder that commits money to a venture with the expectation of generating a profit or some other form of benefit.

2. **Vendor**
   Stakeholder that provides goods or services essential for the development, deployment, or maintenance of the software.

3. **ExecutiveSponsor**
   Stakeholder that acts as a champion for the project within the organization, providing various levels of support and guidance.

4. **RequirementsReviewer**
   Stakeholder responsible for conducting reviews in requirements artifacts.

5. **RequirementsStakeholder**
   Stakeholder that provides needs and expectations for the product.

6. **Customer**
   Stakeholder that purchases the software or pays for its development. They may or may not be the end users themselves.

7. **EndUser**
   Stakeholder who will ultimately interact with and use the software on a regular basis.

8. **RequirementsEngineer**
   Stakeholder responsible for conducting the requirements development activities.

9. **Programmer**
   Stakeholder with programming skills responsible for producing and documenting the software code.

10. **CodeReviewer**
    Stakeholder responsible for reviewing the code according to the requirements document and design document.

11. **Software**
    One or more computer programs together with any accompanying auxiliary items, such as documentation, delivered under a single name, ready for use.

12. **Development**
    Specialization of the *Software* entity, focusing on tools and environments used for software development.

13. **Educational**
    Specialization of the *Software* entity, representing software aimed at educational purposes.

14. **OfficeAutomation**
    Specialization of the *Software* entity, focusing on software designed for office tasks, such as document processing and spreadsheets.

15. **OperatingSystem**
    Specialization of the *Software* entity, representing software designed to manage and coordinate hardware resources and applications.

16. **Videogame**
    Specialization of the *Software* entity, representing software products designed for gaming and entertainment purposes.

17. **Server**
    Entity representing a specialized software or system component responsible for managing network resources and services.

18. **SoftwareComponent**
    Piece of software produced during the software process, not considered a complete Software Product, but an intermediary result.

19. **SoftwareSystem**
    Entity representing an integrated set of software components designed to perform a complex function or set of functions.

20. **Program**
    A sequence of instructions written to perform a specified task with a computer.

21. **Code**
    Entity representing the actual lines of code written in a programming language.

22. **SourceCode**
    A well-formed sequence of computer instructions and data definitions expressed in a programming language, in a form suitable for input to an assembler, compiler, or other translator.

23. **MachineCode**
    Computer instructions and data definitions expressed in a form output by an assembler, compiler, or other translator, recognizable by the processing unit of a computer machine.

24. **InformationItem**
    Relevant information for human use, particularly within the context of software documentation or reporting.

25. **Documentation**
    Entity representing documents that provide information or instruction about a software product.

26. **ComponentDescription**
    Detailed information about a software component within a system.

27. **Pseudocode**
    Informal way of describing the logic of a computer program, using a mixture of natural language and programming language syntax.

28. **IntentDocumentation**
    Documents that capture the underlying goals and purposes of a software project or component.

29. **SpecificationDocument**
    Documents that detail the specifications, including functional and non-functional requirements, of a software project.

30. **SystemSpecification**
    Detailed document specifying the architecture, components, and interactions within a software system.

31. **ProgramSpecification**
    Document specifying the details of a particular program, including its expected behavior and constraints.

32. **BugReport**
    Document that captures information about an identified issue or malfunction within a software system.

33. **UserStory**
    Document that captures a specific feature or functionality of a software system from the end user's perspective.

34. **UseCase**
    Document that captures a collection of user stories for a software project.

35. **TestCase**
    Document containing the input data, expected results, steps, and general conditions for testing some situation regarding a code.

36. **AgreementEmail**
    Document that serves as a formal agreement reached between two or more stakeholders.

37. **DesignDocument**
    Comprehensive document that outlines the technical plan for building a software project.

38. **CodeReviewReport**
    Document pointing out problems identified in the code during review activities.

39. **RequirementsDocument**
    Document reporting requirements and related information.

40. **RequirementsEvaluationDocument**
    Document pointing out problems identified in documented requirements and conceptual models registered in a requirements document.

41. **RequirementsAgreement**
    Represents the agreement achieved by the stakeholders, regarding the requirements for the product.

42. **Intent**
    Captures what the project aims to achieve, reflecting the desires and expectations of key stakeholders.

43. **AdvertisingMaterial**
    Entity representing promotional content created to advertise a software product or service.

44. **SlideDeck**
    Collection of slides used for presentations, typically for conveying information about a software project.

45. **Slide**
    Individual element within a SlideDeck, representing a single page or screen of content.

46. **RequirementArtifact**
    Information Item describing a requirement, often used in documentation and modeling.

47. **Model**
    A representation (abstraction) of a process or system from a particular perspective.

48. **DigitalItem**
    Any item that exists in a digital form, including files, software, and other digital resources.

49. **FileSystemItem**
    Any item within a file system, including files and directories.

50. **Directory**
    A type of file system entity that can contain other files or directories.

51. **PerformedActivity**
    Action performed by a Stakeholder as part of the software process.

52. **PerformedSimpleActivity**
    Performed Activity that is not further decomposed into other Performed Activities.

53. **CodeDevelopment**
    Activity executed for developing the software code.

54. **CodeDocumentation**
    Activity executed for documenting the software Code with useful information.

55. **CodeReview**
    Activity executed for reviewing the software code to identify errors and non-conformances.

56. **PerformedCompositeActivity**
    Performed Activity composed of other Performed Activities.

57. **RequirementsElicitation**
    Identification of requirements from the stakeholders and other sources, and documenting them.

58. **ConceptualModeling**
    Activity executed for modeling requirements, producing conceptual models.

59. **RequirementsDocumentation**
    Activity executed for recording and managing requirements during and after the project.

60. **RequirementsVerification**
    Activity executed for evaluating the requirements and for recording the identified problems.

61. **RequirementsNegotiation**
    Activity executed to solve problems in the requirements and to reach an agreement on the set of requirements to be considered in the project.

62. **Coding**
    Activity involving the creation of code, typically as part of software development.

63. **Requirement**
    Goal to be achieved, representing a condition or capacity needed for the system users to solve a problem.

64. **FunctionalRequirement**
    Requirement defining a function to be available in the product being built.

65. **NonFunctionalRequirement**
    Requirement defining criteria or capabilities for the product.

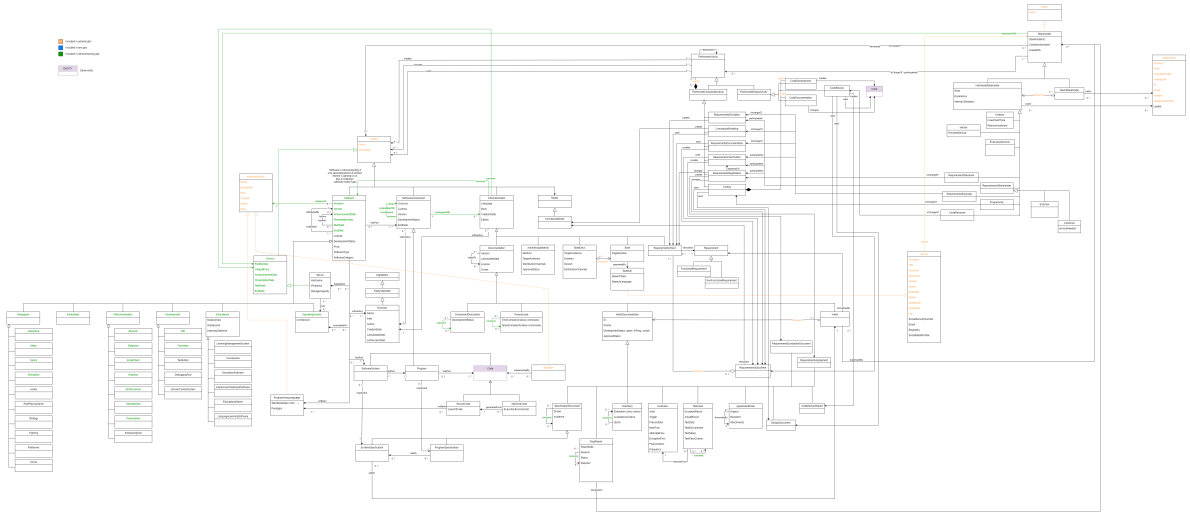

Figure 1: Link to SVG Readable Copy

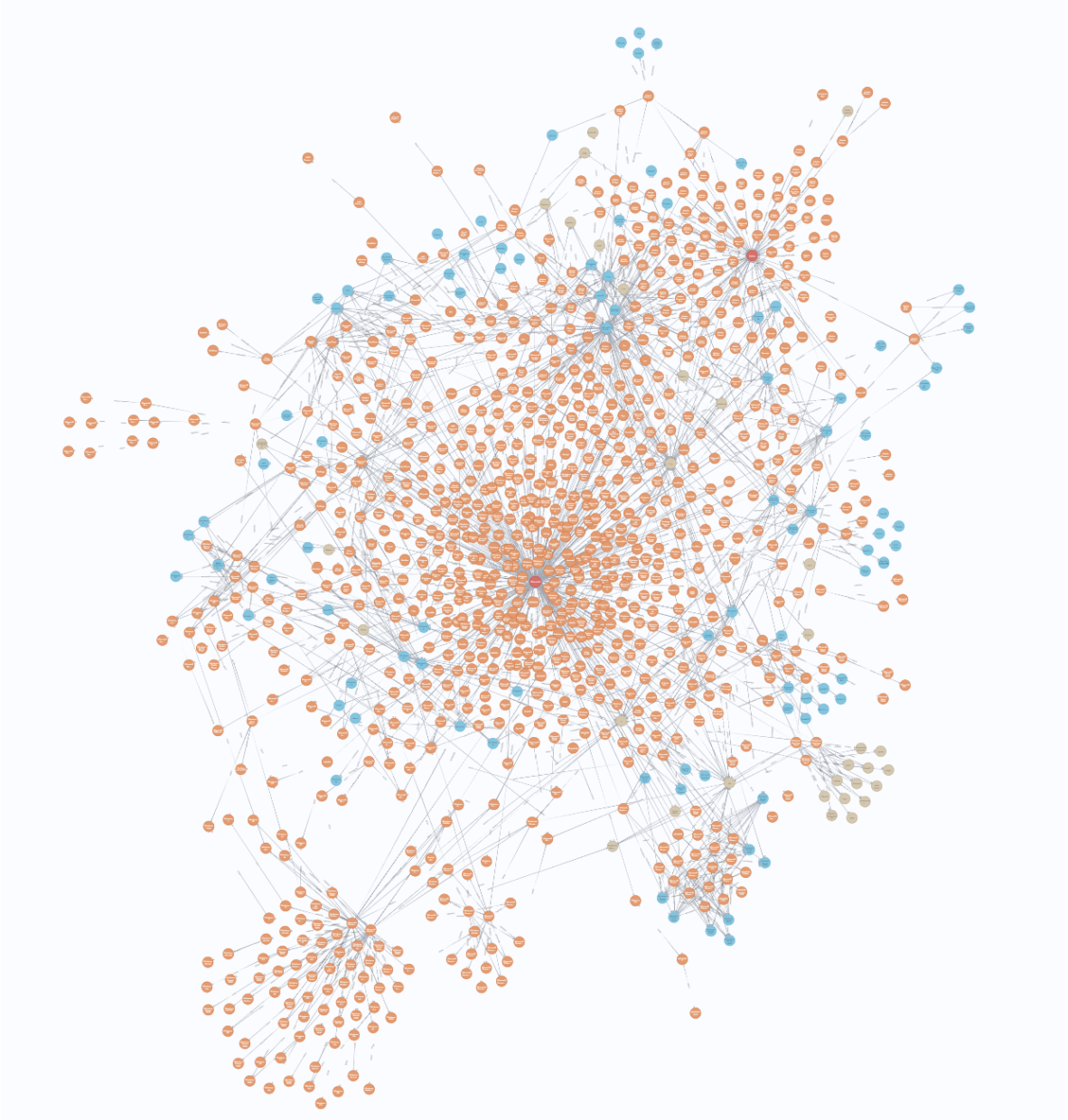

Figure 2: The graph provided by the Neo4j GUI after loading the instances.