НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ННК "Інститут прикладного системного аналізу"
(повна назва інституту/факультету)

Кафедра Системного проектування
(повна назва кафедри)

«До захисту допущено»
Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)
"___"_____2016 р.

Дипломна робота

першого (бакалаврського) рівня вищої освіти
(першого (бакалаврського), другого (магістерського))

зі спеціальності 7.05010102, 8.05010102 Інформаційні технології проектування
7.05010103, 8.05010103 Системне проектування
(код та назва напряму підготовки)

на тему: Алгоритми логічного виведення фактів в OWL-онтологіях

Виконала: студентка 4 курсу, групи ДА-21
(шифр групи)

Слухай Яна Олександрівна _____
(прізвище, ім'я, по батькові) (підпис)

Керівник ст. викладач, к.т.н. Булах Б.В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант Економічний професор, д.е.н. Семенченко Н.В _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Нормоконтроль ст. викладач Бритов О.А. _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2016 року

Національний технічний університет України
«Київський політехнічний інститут»

Факультет (інститут) ННК "Інститут прикладного системного аналізу"
(повна назва)

Кафедра Системного проектування
(повна назва)

Рівень вищої освіти Перший(Бакалаврський)
(перший (бакалаврський), другий (магістерський) або спеціаліста)

Спеціальність 7.05010102, 8.05010102 Інформаційні технології проектування
7.05010103, 8.05010103 Системне проектування
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)

«___»_____2016 р.

ЗАВДАННЯ
на дипломний проект (роботу) студенту
Слухай Яні Олександрівні
(прізвище, ім'я, по батькові)

1. Тема проекту Алгоритми логічного виведення фактів в OWL-онтологіях

керівник проекту (роботи) Булах Богдан Вікторович, к.т.н., ст. викладач
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 12 травня 2016 р. № 50-ст

2. Строк подання студентом проекту (роботи) 08.06.2016
2. Вихідні дані до проекту (роботи)
Зразки онтологій для тестування додатку, що є у відкритому доступі

3. Зміст розрахунково-пояснювальної записки (перелік завдань, які потрібно розробити)
   1. Опис предметної області.
   2. Аналіз семантичних різонерів.

3. Додаток для аналізу різонерів.
4. Економічна частина.

4. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників, плакатів тощо)
   1. Результати порівняння різонерів – плакат.
   2. Тестові онтології– плакат.
   3. UML-діаграма класів розробленого додатку – плакат.

5. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний розділ	Семенченко Н.В., професор		
Основна частина			

6. Дата видачі завдання 01.02.2016

Календарний план

№ з/п	Назва етапів виконання дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Отримання завдання	01.02.2016	
2	Збір інформації	15.02.2016	
3	Дослідження структури OWL-онтологій	28.02.2016	
4	Дослідження структури різонерів	10.03.2016	
5	Аналіз основних алгоритмів	15.03.2016	
6	Розробка програмного додатку	25.03.2016	
7	Вибір та аналіз тестових онтологій	25.04.2016	
8	Тестування програмного додатку	30.04.2016	
9	Оформлення дипломної роботи	31.05.2016	
10	Отримання допуску до захисту та подача роботи в ДЕК	08.06.2016	

Студент _____ Я.О. Слухай
(підпис) (ініціали, прізвище)

Керівник проекту (роботи) _____ Б.В. Булах
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

Бакалаврської роботи Слухай Яни Олександрівни на тему «Алгоритми логічного виведення фактів в OWL-онтологіях».

Дипломна робота присвячена детальному аналізу семантичних різонерів та алгоритмів, за допомогою яких вони проектуються.

Метою дипломного проекту є розробка програмного додатку, що аналізує швидкодію та якість роботи основних функцій вибраних семантичних різонерів (Structural reasoner, Pellet, Hermit), виконуючи їх для вибраної користувачем онтології.

Значна увага приділена аналізу роботи програми, для чого було підібрано та охарактеризовано шість тестових онтологій.

Окремий розділ присвячений функціонально-вартісному аналізу розробленого програмного продукту.

Загальний обсяг роботи – 92 сторінки, 24 рисунки, 9 таблиць, 21 посилання, 1 додаток на 13 сторінок.

Ключові слова: OWL, OWL API, Семантична павутина, онтологія, семантична таблиця, гіпертаблиця, пряме виведення, зворотне виведення, Structural reasoner, Pellet, Hermit.

АННОТАЦИЯ

Бакалаврской работы Слухай Яны Александровны на тему «Алгоритмы логического вывода фактов в OWL-онтологиях».

Дипломная работа посвящена детальному анализу семантических ризонеров и алгоритмов, с помощью которых они проектируются.

Целью дипломного проекта является разработка программного приложения, которое анализирует быстродействие и качество работы основных функций выбранных семантических ризонеров (Structural reasoner, Pellet, Hermit), выполняя их для выбранной пользователем онтологии.

Значительное внимание уделено анализу работы программы, для чего было подобрано и охарактеризовано шесть тестовых онтологий.

Отдельный раздел посвящен функционально-стоимостному анализу разработанного программного продукта.

Общий объем работы - 92 страниц, 24 рисунка, 9 таблиц, 21 ссылка, 1 приложение на 13 страниц.

Ключевые слова: OWL, OWL API, Семантическая паутина, онтология, семантическая таблица, гипертаблица, прямой вывод, обратный вывод, Structural reasoner, Pellet, Hermit.

ANNOTATION

For the bachelor's degree work of Slukhai Yana Olexandrivna on "Reasoning algorithms for OWL onthologies".

Diploma project is devoted to a detailed analysis of semantic reasoners and algorithms they are based on.

The aim of the diploma project is to develop a software application that analyzes the performance and quality of the main functions of the selected semantic reasoners (Structural reasoner, Pellet, Hermit), performing them for the selected ontology.

A lot of attention is paid to the program analysis, as there were six test ontologies chosen and described.

A separate section is devoted to software value engineering of developed application.

The total amount of work - 92 pages, 24 drawings, 9 tables, 21 references, 1 addition on 13 pages.

Keywords: OWL, OWL API, Semantic Web, ontologies, semantic table, hypertableau, forward chaining, backward chaining, Structural reasoner, Pellet, Hermit.

ЗМІСТ

ВСТУП

Все швидше і динамічніше розвивається IT-індустрія у сфері автоматичного пошуку, збору і обробки інформації. Одним з провідних напрямків цієї галузі є технологія Semantic Web . Серед останніх значних досягнень даної технології специфікація мови опису онтологій OWL .

У загальних рисах під онтологією розуміється система понять деякої предметної області, яка представляється як набір сутностей, з'єднаних різними відношеннями. Онтології використовуються для формальної специфікації понять і відношень, які характеризують певну область знань. Перевагою онтологій як способу представлення знань є їх формальна структура, яка спрощує їх комп'ютерну обробку.

Можна говорити про неявне застосування онтологій як системи понять в природничих науках (біологія, медицина, геологія та інші), де вони служать свого роду фундаментом для побудови теорій. Оскільки класифікаційна структура (таксономія) є невід'ємною частиною будь-якої онтології, можна говорити про присутність елементів онтологій в спеціальних класифікаціях і системах індексації (наприклад, в бібліотечних класифікаційних кодах).

У явному вигляді онтології використовуються як джерела даних для багатьох комп'ютерних програм (для інформаційного пошуку, аналізу текстів, вилучення знань і в інших інформаційних технологіях), дозволяючи більш ефективно обробляти складну і різноманітну інформацію[1]. Цей спосіб представлення знань дозволяє додаткам розпізнавати ті семантичні відмінності, які є самі по собі зрозумілими для людей, але не є відомими для комп'ютера. Саме поняття онтології відоме давно, але, будучи переосмисленим, воно стало застосовуватися в комп'ютерних технологіях лише недавно. Повноцінна розробка онтологій в новому розумінні цього терміна почалася лише в кінці 90-х. Це досить нова і мало досліджена галузь прикладної лінгвістики.

Особливу увагу привертає можливість виведення нових логічних фактів, що забезпечується різонерами, ключовими компонентами для забезпечення роботи з OWL-онтологіями [2]. Практично всі запити до OWL-онтології виконуються, так чи інакше використовуючи їх. Це спричинено тим, що знання у онтологіях можуть зберігатися у неявному вигляді, а для отримання коректних результатів запитів необхідно вивести їх явно.

Дана робота присвячена огляду алгоритмів, які найчастіше лягають в основу побудови різонерів, їх аналізу та порівнянню швидкодії та точності виконання основних функцій різонерів у контексті різних за вибраними показниками онтологій.

Метою дипломного проекту є розробка програмного додатку, що аналізує швидкодію та якість роботи основних функцій вибраних семантичних різонерів (Structural reasoner, Pellet, Hermit), виконуючи їх над вибраною користувачем онтологією.

1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Онтології

1.1.1 Визначення поняття онтології

Онтології отримали досить широке поширення в задачах представлення знань та інженерії знань, семантичної інтеграції інформаційних ресурсів та інформаційного пошуку. У контексті комп’ютерних наук онтологія – це "специфікація концептуалізації предметної області", або спрощено, документ або файл, що формально задає відносини між термінами. Це свого роду словник понять предметної області і сукупність явно виражених припущень щодо змісту цих понять.

Інакше кажучи, онтологія — представлення деякою мовою знань певної предметної області (середовище, світ). Онтологію неодмінно супроводжує деяка концепція цієї області інтересів. Найчастіше ця концепція виражається за допомогою визначення базових об’єктів (індивідуумів, атрибутів, процесів) і відношень між ними. Визначення цих об'єктів і відношень між ними зазвичай називають концептуалізацією.

У філософії онтологією називають теорію про природу буття і видах сутностей. Онтологічний рівень формалізує накопичені знання, визначаючи і об'єднуючи термінологію різних предметних областей.

Хоча термін «онтологія» споконвічно філософський, в інформатиці він набув самостійного значення. Тут є дві істотні відмінності:

- онтологія в інформатиці повинна мати формат, який комп'ютер зможе легко обробити;
- інформаційні онтології створюються завжди з конкретними цілями — рішення конструкторських задач; вони оцінюються більше з погляду застосовності, ніж повноти.

У якості робочого визначення, найбільш пристосованою для цілей комп'ютерної лінгвістики, можна розглядати дефініцію, запропоновану Едвардом Хові [3]: «Онтологія - це структура даних з заданими в ній символами, що дозволяють представляти концептуалізації для обробки комп'ютерними програмами ».

Онтології нагадують ієрархії класів в об'єктно-орієнтованому програмуванні, але є кілька важливих відмінностей. Ієрархії класів призначені для представлення структури, що використовуються у вихідному коді, які розвиваються досить повільно, тоді як онтології призначені для представлення інформації в Інтернеті і, як очікується, будуть оновлюватись практично постійно.

Онтології, як правило, набагато гнучкіші, оскільки вони призначені для представлення інформації в Інтернеті, що надходить з різнорідних джерел даних. Класи ієрархій в свою чергу, є більш статичними і розраховані на меншу різноманітність типів даних і більш структуровані джерела даних, таких як корпоративні бази даних

1.1.2 Структура онтологій

У загальному вигляді структура онтології являє собою набір елементів чотирьох категорій:

- поняття;
- відношення;
- аксіоми;
- окремі екземпляри;

Поняття розглядаються як концептуалізації класу всіх представників якоїсь сутності або явища (наприклад, Тварина, Почуття). Класи (або поняття) є загальними категоріями, які можуть бути впорядковані ієрархічно. Кожен клас описує групу індивідуальних сутностей, які об'єднані на підставі наявності загальних властивостей.

Поняття можуть бути пов'язані різного роду відношеннями (наприклад, Довжина, Місцезнаходження), які описують їх. Найпоширенішим типом відношень, що використовується у всіх онтологіях, є відношення категоризації, тобто віднесення до певної категорії. Цей тип відношення має ряд інших назв [3], що зустрічається в різних дослідженнях:

- таксономическое відношення;
- відношення IS-A;
- клас - підклас;
- лінгвістика: гіпонім - гіперонім;
- родовидовое відношення;
- відношення a-kind-of.

Аксіоми задають умови співвіднесення категорій і відношень, вони виражають очевидні твердження, що зв'язують поняття і відношення. Під аксіомою можна розуміти твердження, що вводиться в онтологію в готовому вигляді, з якого можуть бути виведені інші твердження. Вони дозволяють висловити ту інформацію, що не може бути відображена в онтології за допомогою побудови ієрархії понять і установки різних відношень між поняттями. Прикладом аксіоми може послужити таке висловлювання: «Якщо X смертний, то X колись помре ». Аксіоми дозволяють надалі здійснювати умовиводи в рамках онтології. Вони можуть постачати дослідників інформацією про правила, які дозволяють автоматично додавати інформацію. Аксіоми можуть також являти собою обмеження, що накладаються на будь-які відношення, які роблять можливим виведення умовиводів. Наведемо кілька прикладів таких обмежень. Понятійні обмеження вказують на те, який тип понять може виражати дане відношення (наприклад, властивість Колір може виражатися тільки поняттями категорії Колір). Прикладом числових обмежень є твердження того, що для Людини кількість біологічних батьків рівна двом. Кількість і ступінь деталізації аксіом зазвичай залежать від типу онтології.

1.1.3 Класифікація онтологій

Онтології сильно розрізняються по ряду параметрів, і дослідники виділяють різні підстави для їх класифікації. Так Е. Хові [3] говорить, що онтології різняться залежно від набору елементів, що містяться в них, а також від типів введених відношень. Він виділяє так звані «термінологічні онтології» і «справжні онтології». Під першими Е.Хові [3] розуміє онтології, що включають суті, явища, властивості, зв'язки предметної області і об'єднують їх структурні відношення. «Справжні» ж онтології включають в себе також дефініційні відношення і відношення додаткової інформації. Поряд з цим в них входять аксіоми, що визначають взаємозалежності між відношеннями і поняттями.

Е.Хові вибудовує детальну класифікацію різних характеристик онтологій. За його думкою основними параметрами можуть бути: форма (те, як формується онтологія), зміст, а також засоби використання онтології. Існує розбиття онтологій за кількістю і якістю понять, що наявні в них.

Онтології верхньої зони зазвичай нараховують приблизно 100-500 концептів. У них включені найбільш абстрактні категорії, що володіють властивістю універсальності. Вони є базовою розбивкою дійсності на категорії. Зазвичай вони будуються теоретиками, філософами. Перевагою таких онтологій є можливість їх використання в багатьох областях і навіть у багатьох мовах. Для даного роду онтологій характерний обмежений набір узагальнених відношень, які можна віднести до базових (таких як родовидові відносини, відносини частина-ціле і асоціативні відносини). У таких онтологіях типовими на верхньому рівні розбиття є такі поняття, як:

- сутність;
- явище;
- об'єкт;
- процес;

- роль .

Іншим типом є онтології середньої зони, тут елементів зазвичай вже більше (500 - 100000 концептів). Вони представляють світ в цілому і в загальному випадку це неаксіоматизована область. Складність полягає в тому, що для даного виду онтологій потрібно виводити занадто велику кількість аксіом. Зазвичай виходом є використання методів автоматизованого виведення аксіом з уже існуючих онтологій. Побудовою онтологій цього рівня найчастіше займаються когнітологи і лінгвісти.

Онтології нижньої зони або так звані онтології предметної області є найбільш обширними, зазвичай вони налічують близько 200 - 2 000 концептів. Описують конкретні предметні області з їх специфікою. При цьому коло вирішуваних завдань і питань, на які онтологія відповідає, обмежене обраною областю. Для даного типу онтологій характерна наявність відношень, специфічних для конкретної області [3]. Це високоаксіоматизована зона, тобто для неї можлива побудова великої кількості аксіом і правил. В більшості випадків цей тип онтологій будується експертами галузі знання або за їх сприяння. В зв'язку з великою специфікою кожної окремої предметної онтології її повторне використання найчастіше можливо тільки в рамках предметної області.

Поряд з описаним розподілом всі онтології можуть бути розділені на глибинні і поверхневі. Поверхневі онтології будуються на поверхневій семантиці, вони визначають поняття через значення слів. Однак тут виникає проблема, яку кількість сенсів виділяти для кожного слова. Глибинні ж онтології використовують глибинну семантику.

З точки зору предмета концептуалізації дослідники виділяють прикладні онтології, онтології області знання, загальні (родові) онтології і репрезентаційні онтології (мова йде про онтології метарівня, що включають в себе репрезентаційні першоелементи) [4].

Онтології можуть бути також розділені на одномовні і багатомовні. Уже існує ряд онтологій, орієнтованих на подання знань на декількох мовах, наприклад, EuroWordNet, MikroKosmos і деякі інші. Складність створення таких онтологій зазвичай полягає в тому, що можлива наявність відмінностей в понятійних системах різних мов[5].

С. А. Коваль [6] пропонує розрізняти безекземплярні і екземплярні онтології. Як зрозуміло з назви даного типу онтологій, безекземплярні онтології відрізняються відсутністю конкретних екземплярів. На нижніх рівнях ієрархії таких онтологій знаходяться не конкретні екземпляри, а поняття. Ця особливість онтології накладає певний відбиток і на відношення, що вводяться у даній онтології.

Таким чином, існує безліч підрозділів онтологій, але ці класифікації не завжди бувають досить чіткими і послідовними.

1.2 Концепція Семантичної павутини

Семантична павутина (англ. Semantic web) — нова концепція розвитку Всесвітньої павутини і мережі Інтернет, яка створена і впроваджується Консорціумом Всесвітньої павутини (англ. World Wide Web Consortium, W3C). Інші назви — семантичний веб, семантична мережа. Хоча поняття семантична мережа, яке виникло раніше, породило поняття семантична павутина, їх слід відокремлювати.

В свою чергу, семантична мережа — інформаційна модель предметної області, що має вигляд орієнтованого графа, вершини якого відповідають об'єктам предметної області, а ребра задають відносини між ними. Об'єктами можуть бути поняття, події, властивості, процеси[7]. Таким чином, семантична мережа є одним із способів представлення знань.

У назві сполучені терміни з двох наук: семантика у мовознавстві вивчає сенс одиниць мови, а мережа в математиці є різновидом графу — набору вершин,

сполучених дугами (ребрами). У семантичній мережі роль вершин виконують поняття бази знань, а дуги (причому направлені) задають відношення між ними. Таким чином, семантична мережа відображає семантику предметної області у вигляді понять і відношень між ними.

1.2.1 Семантичні відношення

Кількість типів відношень в семантичній мережі визначається її розробником, виходячи з конкретних цілей. В реальному світі їхня кількість прямує до нескінченності. Кожне відношення є, по суті, предикатом, простим або складним. Швидкість роботи з базою знань залежить від того, наскільки ефективно зроблені програми обробки потрібних відношень.

Найчастіше виникає потреба в описі відносин між елементами, множинами і частинами об'єктів. Відношення між об'єктом і множиною, що позначає, що об'єкт належить цій множині, називається відношенням класифікації (ISA). Говорять, що множина (клас) класифікує свої екземпляри[3]. Назва походить від англійського «IS A». Іноді це відношення іменують також MemberOf або якось подібно. Зв'язок ISA припускає, що властивості об'єкта успадковуються від множини. Зворотне до ISA відношення використовується для позначення прикладів, тому так і називається — «Example», або українською «Наприклад».

Відношення між надмножиною і підмножиною називається AKO — «A Kind Of» («різновид»). Елемент підмножини називається гипонімом, а надмножини — гиперонімом, а саме відношення називається відношенням гипонімії. Альтернативні назви — «SubsetOf» і «Підмножина». Це відношення визначає, що кожен елемент першої множини входить і в друге (виконується ISA для кожного елементу), а також логічний зв'язок між самими підмножинами: що перше не більше другого і властивості першої множини успадковуються другою.

Об'єкт, зазвичай, складається з декількох частин, або елементів. Наприклад, комп'ютер складається з системного блоку, монітора, клавіатури, миші і т. д.

Важливим відношенням є HasPart, що описує частини/складові об'єкти (відношення меронімії). Меронім — це об'єкт, що є частиною іншого. Двигун — це меронім для автомобіля. Холонім — це об'єкт, який влючає в себе інше. Наприклад, біля будинку є дах. Будинок — холонім для даху. Комп'ютер — холонім для монітора. Меронім і холонім — протилежні поняття.

Часто в семантичних мережах потрібно визначити відносини синонімії і антонімії. Ці зв'язки або дублюються явно в самій мережі, або в алгоритмічній складовій.

Спроба створення семантичної мережі на основі всесвітньої павутини отримала назву семантичної павутини. Ця концепція має на увазі використання мови RDF (мови розмітки на основі XML) і покликана додати посиланням якийсь сенс, зрозумілий комп'ютерним системам. Це дозволить перетворити Інтернет на розподілену базу знань глобального масштабу.

Основна ідея створення семантичної павутини полягає в тому, що, на відмінну від існуючої гіпертекстової мережі, яку обробляє людина, семантична глобальна мережа майбутнього повинна б була представити інформацію таким чином, щоб її можна було обробляти автоматично без втручання людини. Як відомо існуюча на сьогоднішній день глобальна павутина базується на інформації представленій гіпертекстовою мовою розмітки і представляється людині через спеціально розроблені для цього засоби, зокрема браузери. На зміну цьому старому підходу, семантична павутина покликана представити наявну на сьогоднішній день інформацію у формі семантичних мереж, використовуючи при цьому, як сказано у статті, формалізацію областей знань з допомогою концептуальних схем, що в свою чергу складається із структур даних (об'єктів, зв'язків між ними і правил), прийнятих у даній області.

Першовідкривачем та засновником поняття семантичної павутини був Тім Бернерс-Лі (Berners-Lee T). Так у 2001 році була опублікована його стаття, що

носила назву «Наступний крок у розвитку Всесвітньої павутини». Тім Бернерс-Лі запропонував створити надбудову до існуючої глобальної мережі, яка б перетворила існуючі дані у такий спосіб, щоб ці дані стали зрозумілими комп'ютерам. Як засіб вирішення даної задачі було запропоновано стандарт RDF (Resource Definition Framework) та RDFS (RDF Schema).

RDF специфікація передбачає створення деякої множини ресурсів, для яких визначаються зв'язки значення — властивість. Ці ресурси ідентифікуються у Web за допомогою спеціальних ідентифікаторів URI. По великому рахунку RDF семантика повинна визначати отологію конкретної предметної області. Саме онтологія з кожним роком завойовує все ширше і ширше застосування у розв'язку задачі представлення знань, семантичної адаптації інформаційних ресурсів та їх пошуку. Дана віха у розвитку специфікації концептуалізації предметної області є досить перспективною, також з її допомогою здійснюються спроби представлення ієрархічних понять.

1.3 Web Ontology Language

OWL (англ. Web Ontology Language) — мова опису онтологій для семантичної павутини. Мова OWL дозволяє описувати класи і відношення між ними, властиві для веб-документів і додатків[8]. OWL заснована на більш ранніх мовах OIL і DAML OIL і в наш час є рекомендованою консорціумом Всесвітньої павутини. В основі мови — уявлення дійсності в моделі даних «об'єкт — властивість». OWL придатна для опису не тільки веб-сторінок, але і будь-яких об'єктів дійсності. Кожному елементу опису в цій мові (в тому числі властивостями, що зв'язує об'єкти) ставиться у відповідність URI.

1.3.1 Різновиди мови

- OWL Lite призначена для користувачів, які потребують передусім класифікаційної ієрархії і простих обмежень. Наприклад, при тому, що вона підтримує обмеження кардинальності (кількості елементів), допускаються значення кардинальності тільки 0 або 1. Для розробників повинно бути простіше в своїх продуктах забезпечити підтримку OWL Lite, чим виразніших варіантів OWL. Зокрема, OWL Lite дозволяє швидко перенести існуючі тезауруси і інші таксономії. OWL Lite також має нижчу формальну складність, ніж OWL DL.
- OWL DL призначена для користувачів, яким потрібна максимальна виразність при збереженні повноти обчислень (всі логічні висновки, що припускаються тією чи іншою онтологією, будуть гарантовано обчислюваними) і розв'язуваності (всі обчислення завершаться за певний час). OWL DL включає всі мовні конструкції OWL, але вони можуть використовуватися лише згідно з певним обмеженням (наприклад, клас може бути підкласом багатьох класів, але не може сам бути представником іншого класу). OWL DL так названий з-за його відповідності дескрипційній логіці — дисципліні, в якій розроблені логіки, що складають формальну основу OWL.
- OWL Full призначена для користувачів, яким потрібна максимальна виразність і синтаксична свобода RDF без гарантій обчислення. Наприклад, у OWL Full клас може розглядатися одночасно як сукупність індивідів і як один індивід у своєму власному значенні. OWL Full дозволяє будувати такі онтології, які розширюють склад зумовленого (RDF або OWL) словника. Малоймовірно, що будь-яке програмне забезпечення зможе здійснювати повну підтримку кожної особливості OWL Full.

1.3.2 Структура мови

Онтології OWL складаються з окремих компонентів, таких як індивіди, класи, властивості об'єкта.

Індивіди(Individuals). Являють собою конкретні об'єкти певної предметної області. У OWL не використовується припущення про унікальність імен (Unique Name Assumption - UNA). Це означає, що два різних імені можуть фактично посилатися на один і той же індивід. В OWL два індивіда можуть позначати один і той же об'єкт предметної області якщо *явно* не вказано інше.

Властивості (Properties). Це бінарні відношення на індивідах. Іншими словами, відношення по'єднують двох індивідів.

Класи (Classes). Це множини, елементами яких є індивіди. Вони описуються, використовуючи формальні (математичні) конструкції, які декларують вимоги для членства в класі. Класи можуть бути організовані в ієрархію відношень виду «підкласс-суперклас», яка також відома як таксономія. Підкласи є підмножинами свого суперкласу. Одна з головних особливостей OWL-DL - те, що ці відношення підкласу і суперкласу (відношення категоризації) можуть бути обчислені автоматично за допомогою різонерів(детальніше в розділі 2). В OWL класи мають описи, які визначають умови, яким повинен задовольняти індивід для того, щоб бути членом класу. Всі класи – підкласи класу owl:Thing, клас-корінь онтології, у DL позначається як $\top$. Найнижчий у ієрархії клас owl:Nothing є підкласом усіх класів онтології. Позначається як $\bot$. Їх використовують для отримання певної інформації про всі об’єкти онтології або про жоден[8].

1.3.2 Синтаксиси мови

Нехай ми маємо онтологію квітів, що буде основана на класі Flower. Кожна OWL онтологія повинна мати свій URI. Нехай він буде таким: http://www.example.org/flower.owl.

Розглянемо об’явлення класу, використовуючи різні доступні синтаксиси мови.

OWL2 Functional

```
Ontology(<http://www.example.org/flower.owl>
 Declaration( Class( :Flower) )
)
```

OWL2 XML

```
<Ontology ontologyIRI="http://www.example.org/flower.owl" ...>
 <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
 <Declaration>
  <Class IRI="Flower"/>
 </Declaration>
</Ontology>
```

Манчестерський

```
Ontology: < http://www.example.org/flower.owl>

Class: Flower
```

RDF/XML

```
<rdf:RDF ...>

   <owl:Ontology rdf:about=""/>

   <owl:Class rdf:about="#Flower"/>

</rdf:RDF>
```

RDF/Turtle

```
< http://www.example.org/flower.owl> rdf:type owl:Ontology .

:Flower  rdf:type owl:Class
```

1.4 Висновки

У даному розділі було розглянуто основні принципи предметної області. Проведено аналіз концепцій Семантичної павутини, семантичних відношень, особливості визначення та структури онтологій у контексті комп’ютерних наук, їх класифікацій.

Було проаналізовано мову OWL для створення та редагування онтологій семантичної павутини, яка дозволяє описувати класи і відношення між ними, властиві для веб-документів і додатків. Охарактеризовано компоненти, з яких зазвичай складаються онтології, описані за допомогою OWL, їхні особливості та суть застосування.

Розглянуто основні різновиди мови OWL(OWL Lite , OWL DL, OWL Full), проаналізовано можливості кодного з них. Також продемонстровано особливості синтаксисів, які підтримує мова (OWL2 Functional, OWL2 XML, Манчестерський, RDF/XML, RDF/Turtle).

2 АНАЛІЗ СЕМАНТИЧНИХ РІЗОНЕРІВ

2.1 Поняття різонера

Семантичний різонер - частина програмного забезпечення, що здатна виводити логічні висновки з набору вибраних фактів та аксіом, а також надає можливість автоматичної підтримки таких завдань як: класифікація, налагодження, формування запитів. Правила виведення зазвичай задаються засобами мови онтологій, і часто засобами мов описової логіки. Використання різонерів для автоматичного виведення ієрархії класів є одним з основних переваг побудови онтології з використанням мови OWL DL [9]. При проектуванні дуже великих онтологій (понад декілька тисяч класів) без різонера дуже важко обслуговувати великі, складні онтології, і зберігати в коректному, належному вигляді.

В OWL реазонери працюють на основі концепції OWR – «Open World Reasoning». Open World Reasoning буквально перекладається з англійської як «міркування про відкритість світу». В OWL DL - це категорія, що позначає відкритість баз знань, заснована на припущенні про відкритість світу (open world assumption), що принципово відрізняє їх від баз даних, заснованих на припущення про замкнутість світу (closed world assumption) [10]. По суті це означає неповноту знань по відношенню до фактів, які можна вивести на їх основі шляхом логічних міркувань. Тобто побудова повної класифікації за описами окремих класів, які здавалося б, не дають повного опису онтології. Це можна розуміти так: якщо «щось» не було сказано, для того щоб бути фактом, не можна припустити його хибним - вважається, що «знання просто не було додано до бази знань». Створення явних припущень у предметній області, що лежать в основі реалізації, дає можливість легко змінити ці припущення при зміні наших знань про предметну область. Жорстке кодування припущень про світ на мові програмування призводить

до того, що ці припущення не тільки складно знайти і зрозуміти, але і також складно змінити, особливо непрограмістам [11]. Без розуміння принципів побудови онтологій, робота реазонера може здаватися непомітною.

2.2 Основні задачі різонерів

Серед основних задач різонерів можна виділити такі:

1. Проводити класифікацію і виводити ієрархію класів.
2. Перевіряти консистентність онтології.
3. Визначати тип індивіда - належність до певного класу.
4. Визначати класи, що не перетинаються з заданим класом.
5. Визначати підкласи вибраних класів.

2.3 Основні алгоритми побудови різонерів

2.2.1 Загальні відомості про правила виведення

У логіці правило виведення, правило виводу або правило перетворення — це логічна форма, що складається з функції, яка отримує передумови, аналізує їхній синтаксис і повертає висновок (або висновки). Наприклад, правило виведення, що називається modus ponens, отримує дві передумови, одну у формі «Якщо p тоді q», а другу у формі «p», і повертає висновок «q». Це правило є чинним відносно семантики класичної логіки (як і відносно семантик багатьох інших некласичних логік), у тому сенсі що якщо передумови є істинними (в межах інтерпретації), то істинним є і висновок.

Зазвичай правило виведення зберігає істинність, семантичну властивість. У багатозначній логіці воно зберігає узагальнене значення. Але дія правила виведення є винятково синтаксичною, і не потребує зберігання ніякої семантичної властивості: будь-яка функція з множин формул до формул вважається правилом виведення [12].

Modus ponens (латиною: метод що підтверджує) - коректна, проста форма аргументації (інколи використовується скорочення MP):

Якщо P, то Q.

P.

Звідси Q.

або у логіко-операторному записі:

$$\begin{array}{l} p \rightarrow q, \\ p \\ \vdash q, \end{array} \tag{2.1}$$

де $\vdash$ означає логічний висновок. Аргумент має два вихідних твердження. Перше це умова «якщо-то» або «умовне» твердження, а саме що із P слідує Q. Друге твердження це те що P, умовна частина першого твердження, є істиною. Із цих двох умов логічно слідує що Q, висновок першого твердження, мусить бути істиною також [12].

Приклад твердження у формі modus ponens:

- Якщо демократія є найкращою системою урядування, то кожен повинен голосувати.
- Демократія є найкращою системою урядування.
- Отже, кожен повинен голосувати.

Той факт, що висновок є коректним не ґарантує істиності вихідних тверджень. Коректність modus ponens каже нам тільки те, що висновок є істинним тоді і тільки тоді, коли всі вихідні твердження є істинними. Слід нагадати, що коректний логічний висновок, у якому одне або більше вихідних тверджень не є істинними, називають необґрунтованим, інакше, якщо усі твердження є істинними, такий аргумент називають обґрунтованим. У більшості логічних систем Modus ponens вважається коректним, хоча кожен окремий випадок застосування може

бути або обґрунтованим або ні. Висновок логіки висловлювань із використанням modus ponens є дедуктивним.

Modus tollens (латиною: спосіб, що заперечує) це формальна назва для доведення від супротивного. Вживається також скорочення MT.

Modus tollens є простою, часто вживаною формою аргументації:

Якщо P, то Q.

Q є фальш.

Тому P є фальш.

Використовуючи логіко-операторну нотацію:

$$\begin{array}{l} p \rightarrow q, \\ \neg q \\ \vdash \neg p. \end{array} \tag{2.2}$$

Аргумент має два посилання. Перше посилання це умовне твердження «якщо - то», а саме, що із P слідує Q. Другим посиланням є те, що Q є фальш. Із цих двох посилань слідує, що P є фальш. (Якщо P істинне, то Q також істинне з першого посилання, але це суперечить другому посиланню.) Важливо зауважити, що в достовірному судженні, якщо посилання істинні, то висновок обов'язково слідує.

2.2.2 Пряме виведення

Пряме виведення **(**forward chaining) є одним з двох основних методів логічного виведення. Є популярною стратегією впровадження експертних систем, бізнесових та продукційних моделей, базованих на правилах [12]. Алгоритм починається з формування ланцюжка з наявними даними і використовує правила виведення для вилучення більшої кількості даних (від кінцевого користувача, наприклад), поки мета не буде досягнута. У системах з прямим виведенням за відомимми фактами відшукується факт, який з них випливає. Якщо такий факт вдається знайти, то він записується в базу фактів. Пряме виведення називають

також виведенням, керованим даними або виведенням, керованим посиланнями правил.

Наприклад, припустимо, що мета полягає в тому, щоб визначити чи є колір домашньої тварини на ім'я Том зеленим, враховуючи, що він квакає і їсть мух, і що база правил містить наступні чотири правила:

1. Якщо X квакає і X їсть мух - Тоді X жаба
2. Якщо X щебече і X співає - Тоді X є канаркою
3. Якщо X жаба - Тоді X зелений
4. Якщо X є канаркою - Тоді X жовтий

Проілюструємо хід логіки машини виведення, що керується прямим логічним виведенням. Приймемо наступні факти:

Том квакає

Том їсть мух

Машина логічного виводу може визначити, що Том зелений серією кроків:

1. Оскільки базові факти вказують на те, що "Том квакає" і "Том їсть мух", антецедент правила № 1 виконується шляхом підстановки Тома на місце X, і укладається механізм логічного висновку:

   Том – жаба

2. Антецедент правила № 3 задовольняється шляхом заміни Тома для X, і укладає механізм логічного висновку:

   Том зелений

У механізму прямого виведення архітектура складається з таких частин:

- Робоча пам’ять, у якій зберігаються наведені факти
- Правила, яким задовольняють ті чи інші факти за певних умов

- Дії, які включають в себе додавання або видалення фактів з робочої пам’яті

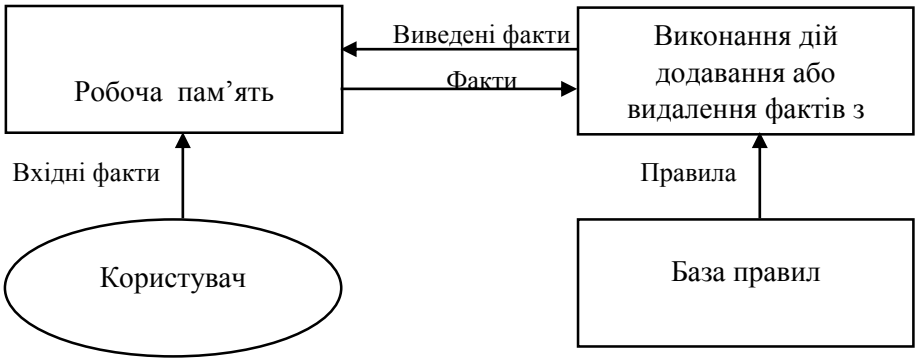


Рисунок 2.1 – Архітектура механізму прямого виведення

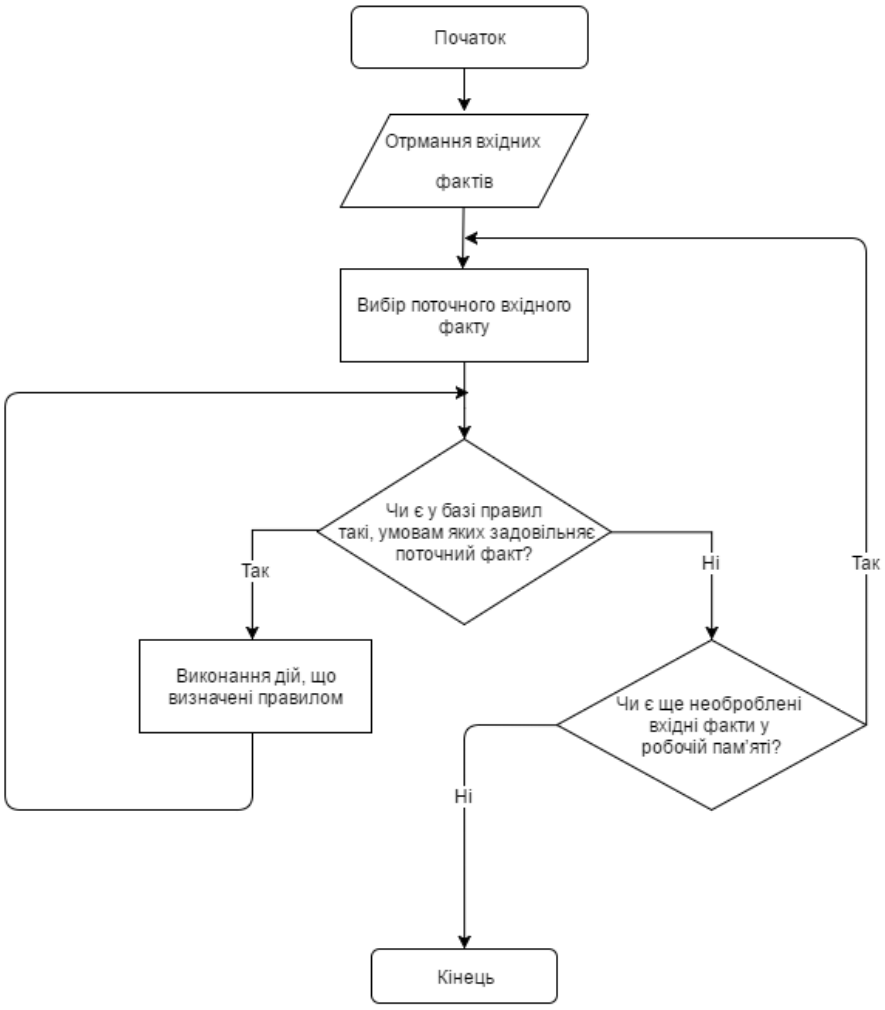


Рисунок 2.2 – Блок-схема алгоритму прямого виведення

Однією з переваг прямого логічного виведення є те, що отримання нових даних може спровокувати виникнення нових висновків, що робить програмний юніт більш гнучким для систем, які динамічно змінюються [13].

2.2.3 Зворотне виведення

Зворотний вивід (або зворотне міркування) - це метод отримання висновку, який працює в зворотному напрямку від мети. Він використовується в автоматичному доведенні теорем, машинному виведенні та інших напрямках штучного інтелекту [12].

Вирішимо ту ж задачу(визначення кольору Тома), використовуючи логіку зворотного виведення.

Для початку, запит формулюється в якості твердження, яке повинно бути доведено: «Том зелений».

Після підстановки Тома замість X у правилі №3 маємо:

Якшо Том жаба – Тоді Том зелений

Оскільки консеквент відповідає поставленому твердженню «Том зелений», машина логічного виведення потребує доведення антецедента «Якщо Том жаба». Це твердженя стає поточною метою:

Том – жаба

Після підстановки Тома на місце X у правилі №1 маємо:

Якщо Том квакає і Том їсть мух - Тоді Том жаба

Оскільки консеквент відповідає поточній меті, машина логічного виведення потребує доведення антецедента «Якщо Том квакає і Том їсть мух» , ці твердження стають новою метою логічного виводу. Оскільки ця мета - це кон’юнкція двох виразів, машина логічного виводу розуміє, що вони обоє були задані як початкові умови. Отже, твердження «Том квакає і Том їсть мух» правдиве. Значить і антецедент правила №1 є правдивим, а тоді і антецедент правила №3 є правдивим теж. А звідси випливає, що і консеквент теж правдивий, а значить

Том зелений

Цілям завжди відповідають висновки, у яких пізніше антецеденти розглядаються в якості нової мети. В кінцевому рахунку антецедентам повинні відповідати відомі факти (вони, як правило, визначаються як висновки, у яких завжди істинний антецедент). Таким чином, правилом виведення, яке використовується є modus ponens.

Оскільки саме список цілей визначає, які правила вибирати і використовувати, цей метод називається методом, керованим метою, на відміну від прямого виводу, що є методом, керованим даними. Зворотний вивід часто використовується в експертних системах [13].

Мови програмування, такі як Пролог, Knowledge Machine і ECLiPSe підтримують метод зворотного виводу в своїх механізмах виводу. Очевидно, що такі системи завичай використовують стратегію пошуку в глибину.

Алгоритм зворотного виведення найчастіше використовують у додатках, метою яких є визначення відповідності тих чи інших тверджень певним критеріям. Прикладом такого комерційного додатку може бути програма, що знаходить відповідність між страховими полісами та контрактами перестрахування, до яких вони відносяться.

Недоліком алгоритму є те, що мета повинна бути точно вказаною, що унеможливлює його використання для обширних запитів. Тому зазвичай для реалізації різонера на базі алгоритмів виведення використовують і зворотне, і пряме виведення.

Детальніше роботу алгоритму зворотного виведення можна розглянути на рисунку 2.3, де зображена його блок-схема.

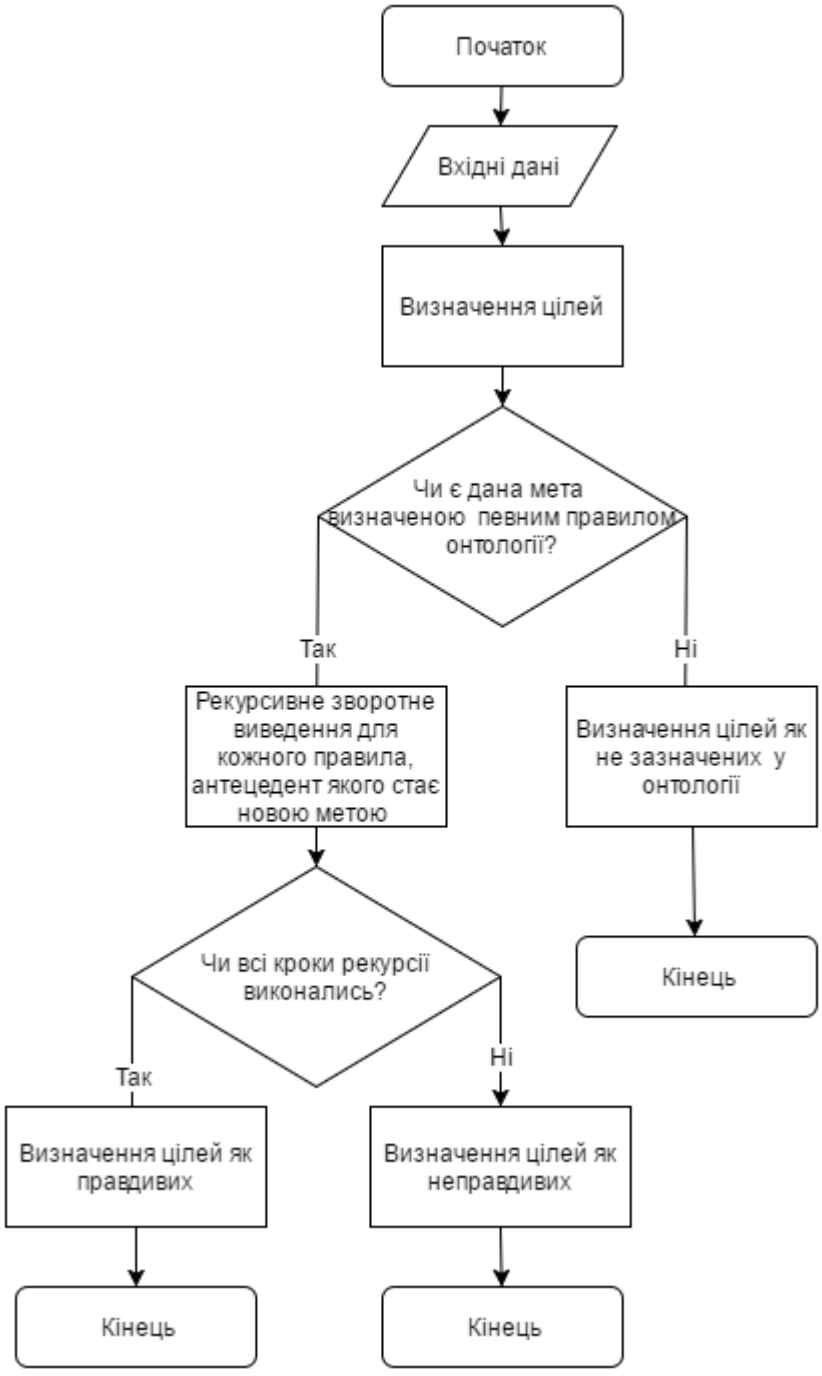


Рисунок 2.3 – Блок-схема алгоритму зворотного виведення

2.2.4 Алгоритм семантичних таблиць

Метод семантичних таблиць - це формальна роздільна процедура для формул логіки висловлювань і логіки предикатів, що дозволяє чисто синтаксичними засобами вирішувати семантичні проблеми формалізованих обчислень. Метод семантичних таблиць розробив голландський філософ і логік Е. В. Бет (E. W. Beth) в 1955 році.

Семантична таблиця - це дерево, вершинами якого є досліджувана формула і все її подформули. Вершина називається особливою, якщо вона відмінна від атома, тобто її логічне значення залежить від останньої виконуваної логічної зв'язки. Вершина називається звичайною, якщо в ній знаходиться атом [14]. Останні вершини кожної гілки - це обов'язково атоми, і такі вершини називаються листками. Семантична таблиця складного висловлювання *K* будується індуктивно, виходячи з семантичних таблиць підформул, що входять в висловлювання *K*. Кожній логічній зв'язці, що виконується у відповідній даній вершині подформулі, зіставляється елементарна семантична таблиця у вигляді дерева, що розкриває логічну інтерпретацію зв'язки. Приклади найбільш вживаних в логіці висловлювань зв'язок наведені на рисунку 2.1

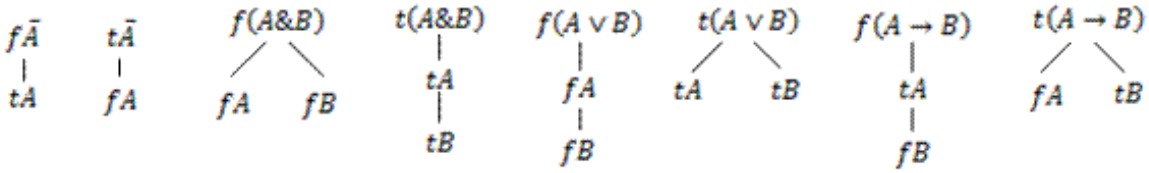


Рисунок 2.4 - Елементарні семантичні таблиці [14]

Ідея методу грунтується на побудові дерева, вершинами якого є всі підформули даної формули. Логічне значення всіх підформул досліджується аж до атомів, які називаються листками, поки не буде виявлено протиріччя в даній вітці або в даній вітці не залишиться особливих вершин. Таке дерево відтворює таблицю істинності

формули. Однак часто немає необхідності будувати все дерево до листків: в деякому вузлі значення функції може бути визначено відразу для всіх листків, які можуть бути побудовані з цього вузла.

Нехай *K* - формула мови логіки висловлювань. Позначимо *tK* як твердження «К - істинно» і *fK* як твердження: «К-брехливо». При цьому *tK* і *fK* називаються поміченими формулами.

Семантична таблиця - це дерево, вершинами якого є позначена формула і її помічені підформули [15].

Семантична таблиця складного висловлювання *K* - це граф, який не має циклів (дерево), з єдиним коренем, в який поміщається позначена формула *K*. Кожна гілка дерева інтерпретується як кон'юнкція всіх підформул, записаних в вершинах цієї гілки. Семантична таблиця будується зверху вниз, виходячи з індуктивного правила визначення формули: неподільні висловлювання (літерали) - це атоми; з атомів будуються формули з використанням логічних зв'язок. Кожна формула є результатом застосування одно- або двомісної логічної зв'язки до підформул, що її утворюють [16].

Процедура побудови семантичної таблиці полягає в наступному [14]:

Побудову семантичної таблиці для складного висловлювання починаємо з того, що записуємо позначену формулу *tK* або *fK* в корінь семантичної таблиці.

Якщо якась частина таблиці вже побудована, знаходимо в безлічі висячих вершин (тобто вершин графа, яким інцидентне єдине ребро) особливі вершини і продовжуємо побудову атомарними семантичними таблицями для кожної з підформул. Вершина семантичної таблиці називається особливою, якщо вона зустрічається як корінь деякої атомарної семантичної таблиці. Якщо вершина позначена поміченим атомом, вона називається звичайною.

Гілка семантичної таблиці називається суперечливою, якщо в ній зустрічаються звичайні вершини, що містять помічені атоми *tP* і *fP*, які означають,

що кон'юнкція всіх підформул, що входять в дану вітку дорівнює нулю. Тобто ця гілка містить протиріччя, обумовлене атомом Р на деякому наборі значень пропозиціональних змінних розв'язуваної задачі.

Семантична таблиця називається замкнутою, якщо жодна з її гілок не містить особливих вершин, які не були б продовжені відповідно до атомарних семантичних таблиць її підформул. В іншому випадку семантична таблиця називається незамкненою.

Семантична таблиця суперечлива, якщо кожна її гілка суперечлива. Якщо замкнута семантична таблиця з коренем *fK* суперечлива, а це означає, що ми намагалися всіма можливими способами зробити висловлювання *K* хибним і не зуміли, то *K* - тавтологія.

Доведенням або висновком по Бету висловлювання *K* називається замкнута суперечлива семантична таблиця, в корені якої поміщена позначена формула *fK*.

Замкнута суперечлива таблиця, що має в якості кореня *tK*, називається спростуванням по Бету висловлювання *K*.

Кажуть, що висловлювання *K* може виводитись по Бету, якщо воно має доведення по Бету. Висловлення називається заперечним по Бету, якщо існує спростування *K* за Бетом [14].

Розглянемо приклад побудови семантичної таблиці Бета для доведення істинності висловлювання

$$(A\rightarrow(B\rightarrow (A\&B))) \qquad (2.3)$$

В корінь дерева помістимо формулу, позначивши її як неправдиву . Розкриваючи послідовно значення всіх її підформул за допомогою елементарних семантичних таблиць для логічних зв'язок «→» і «&», будемо будувати гілки дерева зверху вниз до тих пір, поки в кожній гілці не виявиться суперечність у вигляді входження до гілки одного і того ж атома, позначеного одного разу як

істинний і інший раз як неправдивий. Як тільки протиріччя виявиться, подальша побудова гілки припиняється.

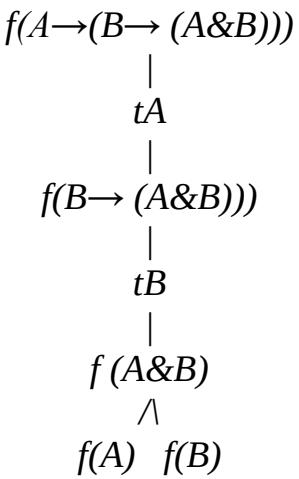


Рисунок 2.5 – Доведення методом семантичних таблиць Бета

2.2.5 Алгоритм гіпертаблиць

Алгоритм побудови гіпертаблиць базується на особливостях побудови аналітичних таблиць, але з використанням переваг головних ідей алгоритму гіпер-резолюцій [17].

Оскільки метод є розширенням табличних методів, він забезпечує багату структуру для всього процесу виведення; важливі частини історії виведення зберігаються в таблиці і можуть бути використані для подальших логічних виведень

За допомогою використання правил алгоритму гіпер-резолюцій було наслідувано можливість за один крок виведення отримати всі заперечення поточної вершини [18].

Головною відмінністю алгоритму від попереднього є наявність наступних правил виведення [18]. Нехай ми маємо вирази *A, B, C* і для них справедливі вирази

$$(\mathrm{A} = 0) \vee (\mathrm{A} = 1) \tag{2.4}$$

$$\overline{((\mathrm{A} = 0) \wedge (\mathrm{B} = 0))} \tag{2.5}$$

$$\overline{((\mathrm{A} = 1) \wedge (\mathrm{B} = 1))} \tag{2.6}$$

Тоді з цих виразів витікає наступний:

$$\overline{((\mathrm{B} = 1) \wedge (\mathrm{B} = 1))} \tag{2.7}$$

Нехай є такі рівності

$$(\mathrm{A} = 0) \vee (\mathrm{A} = 1) \tag{2.6}$$

$$\overline{((\mathrm{A} = 0) \wedge (\mathrm{B} = 0))} \tag{2.7}$$

$$\overline{((\mathrm{A} = 1) \wedge (\mathrm{C} = 1))} \tag{2.8}$$

Тоді справедливо сказати, що

$$\overline{((B = 1) \wedge (\mathrm{C} = 1))} \tag{2.9}$$

2.4 Огляд існуючих різонерів

В даний час існує величезна кількість різонерів, проте зародження їх почалося в період з 1975 року по 2009 рік. Можна аналізувати існуючі різонери за чотирма ознаками:

- Підтримка інтерфейсу
- Вид алгоритму і завершеність
- Мова розробки
- Підтримка мов розробки семантичних мереж

Існує, щонайменше, дев'ятнадцять різонерів. Серед них можна виділити ті, які є досить популярними, а саме: Pellet, RACER, FACT ++, Snorocket, HermiT, CEL, ELK, SWRL-IQ і TrOWL, Structural Reasoner [2].

2.4.1 Pellet

Pellet є вільно поширюваним OWL-DL різонером, реалізований на Java.

Розробники, The Mind Swag Group, в основу даного reasoner-а поклали табличні алгоритми і підтримку виразного опису логіки. Це перший в світі різонер, який підтримує всі OWL DL SHOIN і був розширений до OWL2 . Імплементує OWL API інтерфейс. Також є підтримка пояснення помилок. В основі роботи лежить алгоритм семантичних таблиць.

2.4.2 RACER

Renamed ABoxes and Concept Expression Reasoner – це один з популярних різонерів, який був розроблений Яном Хоррокс. RACER, так само відомий як RacerPro є першим в історії reasner-ом. Підтримка оптимізаційних технологій FACT, а так само підтримка нових методів для роботи з чисельним обмеженням і ABoxes. Racer реалізує TBOX і ABOX різонери для описової логіки SHIQ.

2.4.3 FACT ++

Ян Хоррокс представив різонер, що відомий під назвою FACT ++.
Даний блок міркувань може бути використаний як класифікатор описової логіки і тестування досяжності модальної логіки. Система FACT озвучена і повністю оснащена підтримкою алгоритмів для описової логіки. Після випущених оновлень FaCT перетворився в FaCT ++. Відмінність між ними полягає у внутрішній структурі, яка написана на C ++. Перша версія FaCT ++ підтримувала тільки SHOIQ, OWL-DL. Однак, в останніх версіях з'явилася підтримка OWL цілком і за основу взята описова логіка SROIQ.

2.4.4 SnoRocket

SnoRocket являє собою високопродуктивний поліноміальний алгоритм класифікації для описової логіки EL +. Реалізований даний алгоритм на мові Java. Його розроблено як частину «CSIRO's Health Informatics» і «Clinical Terminologies research program».

2.4.5 SWRL-IQ

Розшифровується як Semantic Web Rule Language Inference and Query tool, і є плагіном для Protégé 3.5, який дозволяє користувачеві змінювати, зберігати, записувати запити та передавати їх до базового механізму логічного висновку, заснованого на XSB Prolog.

2.4.6 ELK

ELK є у вільному доступі. Reasoner для мови онтологій OWL 2 EL. ELK реазонер написаний на Java і контролюється за допомогою OWL API. Доступний тільки під Apache License 2.0. Кросплатформність здійснюється за рахунок підтримки Java 1.5 або вище.

2.4.7 Hermit

Hermit - це перший загальнодоступний OWL-різонер, який написаний за допомогою OWL API. З його допомогою можна перевірити правильність складання онтологій в OWL-файлах і скласти ієрархічну структуру класів. Обчислення будуються за допомогою гіпертабличного числення .

2.4.8 CEL

Даний реазонер був створений на мові програмування LISP. Володіючи простим інтерфейсом у вигляді командного рядка, CEL надає користувачам всі необхідні функції, включаючи прості інтерактивні команди допомоги. В основу різонера лягли алгоритми прямого і зворотного виводу.

2.4.9 TrOWL

TrOWL є загальним інтерфейсом для ряду різонерів, які розроблені в університеті Абердіна. Виділяють два основних види: TrOWL Quill і TrOWL REL. Quill надає зв'язок зі службами OWL 2 QL. TrOWL REL це свого роду оптимізований CEL алгоритм, який забезпечує зв'язок з OWL 2 EL. Для підтримки

повного OWL DL, TrOWL використовує інші різонери, такі як FACT ++, Pellet, Hermit, RacerPro і т.д.

2.4.10 Structural Reasoner

Це структурний різонер, який наразі не є повним, але достатньо часто використовується в контексті OWL-API, де являється одним зі стандартних різонерів фреймворку. Розроблений на Java, використовує алгоритми прямого та зворотного виведення.

Таблиця 2.1 – Порівняльний аналіз різонерів

Різонер / Критерій		Pellet	RACER	FACT++	Snorocket	SWRLIQ	Hermit	CEL	TrOWL	ELK
Алгоритм		Семантичних таблиць	Семантичних таблиць	Семантичних таблиць	Правил виведення	SWRL правид	Гіпертабличного числення	Правил виведення	Правил виведення	Правил виведення
Стійкість		Так	Так	Так	Так	Так	Так	Так	Так	Так
Повність		Так	Так	Так	Так	Ні	Так	Так	Так	Так
Виразність		SROIQ (D)	SHIQ	SROIQ (D)	EL+	-	SROIQ (D)	EL+	SHIQ	EL
Нативний профіль		DL, EL	DL	DL	EL	-	DL	EL	DL, EL	EL
Інкрементальна класифікація	Додавання	Так	Ні	Ні	Так	Ні	Ні	Так	Ні	Так
	Видалення	Так	Ні	Ні	Ні	Ні	Ні	Ні	Ні	Так
Підтримка правил		Так (SWRL)	Так (SWRL)	Ні	Ні	Так (SWRL)	Так (SWRL)	Ні	Ні	Так
Платформи		Всі	Всі	Всі	Всі	Всі	Всі	Linux	Всі	Всі
Обгрунтування		Так	Так	Ні	Ні	Так	Ні	Так	Ні	Ні
Виведення логічних висновків щодо індивідів		Так	Так	Так	Ні	Так	Так	Так	Так	Ні
Підтримка OWL API		Так	Так	Так	Так	Ні	Так	Так	Так	Так
Підтримка OWL Link API		Так	Так	Так	Ні	Ні	Так	Так	Ні	Ні
Підтримка Protégé		Так	Так	Так	Так	Так	Так	Так	Так	Так
Підтримка NeOn		Так	Ні	Ні	Ні	Ні	Так	Ні	Ні	Ні
Ліцензія		DULI: AGPL	Власна	GLGPL	Власна	Так/Ні	GLGPL	Apache License 2.0	DULI: AGPL	Apache License 2.0
Підтримка Jena		Так	Ні	Ні	Ні	Ні	Ні	Ні	Так	Ні
Мова написання		Java	Lisp	C++	Java	Prolog	Java	Lisp	Java	Java
Доступ		Відкритий	Комерційний	Відкритий	Комерційний	Комерційний	Відкритий	Відкритий	Комерційний	Відкритий

2.5 Висновки

У даному розділі було визначено поняття семантичного різонера, основну концепцію його побудови у вигляді Open World Reasoning - категорії, що позначає відкритість баз знань, яка заснована на припущенні про відкритість світу (open world assumption), що принципово відрізняє їх від баз даних, заснованих на припущення про замкнутість світу (closed world assumption).

Було виділено головні задачі різонерів (проведення класифікації і виведення ієрархію класів, перевірка консистентності онтології, визначення типу індивіда – належнісості його до певного класу, визначення класів, що не перетинаються з заданим класом, визначення підкласів вибраних класів)

Проаналізовано найчастіше використовувані при проектуванні різонерів алгоритми(прямого та зворотного виведення, побудови семантичних таблиць, побудови гіпертаблиць)

Було здійснено огляд існуючих різонерів, особливостей їх реалізації.

3 ДОДАТОК ДЛЯ АНАЛІЗУ РІЗОНЕРІВ

3.1 Архітектура та функціонал додатку

У рамках дипломного проекту було вирішено проаналізувати швидкодію та якість виконання головних функцій трьох різонерів, які демонструють роботу розглянутих алгоритмів.

Таблиця 3.1 – Різонери для аналізу в розробленому додатку

Різонер	**Алгоритм**
Structural Reasoner	Прямого та зворотнього виводу
Pellet	Семантичних таблиць
Hermit	Гіпертаблиць

Для цього було створено додаток на мові програмування Java, який для заданих онтологій дозволятиме аналізувати такі параметри роботи різонерів:

1. Час логічного виведення класової ієрархії.
2. Час логічного виведення ієрархії властивостей даних.
3. Час логічного виведення ієрархії властивостей об’єктів.
4. Кількість всіх зв’язків «клас - підклас» онтології.
5. Час отримання всіх зв’язків «клас - підклас» онтології.
6. Кількість всіх зв’язків «клас – непересічний клас» онтології.
7. Час отримання всіх зв’язків «клас – непересічний клас» онтології.
8. Кількість всіх зв’язків «індивідуал – клас» онтології.
9. Час отримання всіх зв’язків «індивідуал – клас» онтології.
10. Час перевірки онтології на консистентність.

11.Кількість всіх класів.

Даний додаток дає зрозуміти користувачеві, які різонери підходять для вирішення яких із перечислених задач найкраще, даючи змогу вибрати оптимальне рішення з їх використання для заданої онтології.

Для реалізації додатку для аналізу різонерів було вирішено скористатись фреймворком OWL API, оскільки OWL API включає в себе інтерфейс OWLReasoner для доступу до OWL різонерів, який імплементують всі вибрані для дослідження різонери.

OWL API є прикладним програмним інтерфейсом для створення, аналізу і обробки OWL онтологій. Версії 3.1 і новіші з них призначені для OWL 2 [19].

OWL API випущений у режимі подвійної ліцензії: LGPL і Apache. Обидві ліцензії у відкритому доступі [19].

Оскільки для роботи додатку необхідно було завантажити велику кількість необхідних бібліотек з реалізацією функцій даних різонерів, було створено Java Maven проект.

Maven - це засіб автоматизації роботи з програмними проектами, який спочатку використовувався для Java проектів [20]. Використовується для управління (management) та складання (build) програм. Створений Джейсоном ван Зилом (Jason van Zyl) у 2002 році. За принципами роботи кардинально відрізняється від Apache Ant, та має простіший вигляд щодо build-налаштувань, яке надається в форматі XML. XML-файл описує проект, його зв'язки з зовнішніми модулями і компонентами, порядок будування (build), папки та необхідні плагіни. Сервер із додатковими модулями та додатковими бібліотеками розміщується на серверах. Раніше Maven, де він був частиною Jakarta Project.

Для опису програмного проекту який потрібно побудувати (build), Maven використовує конструкцію відому як Project Object Model (POM), залежності від зовнішніх модулів, компонентів та порядку побудови. Виконання певних, чітко

визначених задач - таких, як компіляція коду та пакетування відбувається шляхом досягнення заздалегідь визначених цілей (targets).

Ключовою особливістю Maven є його мережева готовність (network-ready) [20].

Двигун ядра може динамічно завантажувати плагіни з репозиторію, того самого репозиторію, що забезпечує доступ до багатьох версій різних Java-проектів з відкритим кодом, від Apache та інших організацій та окремих розробників.

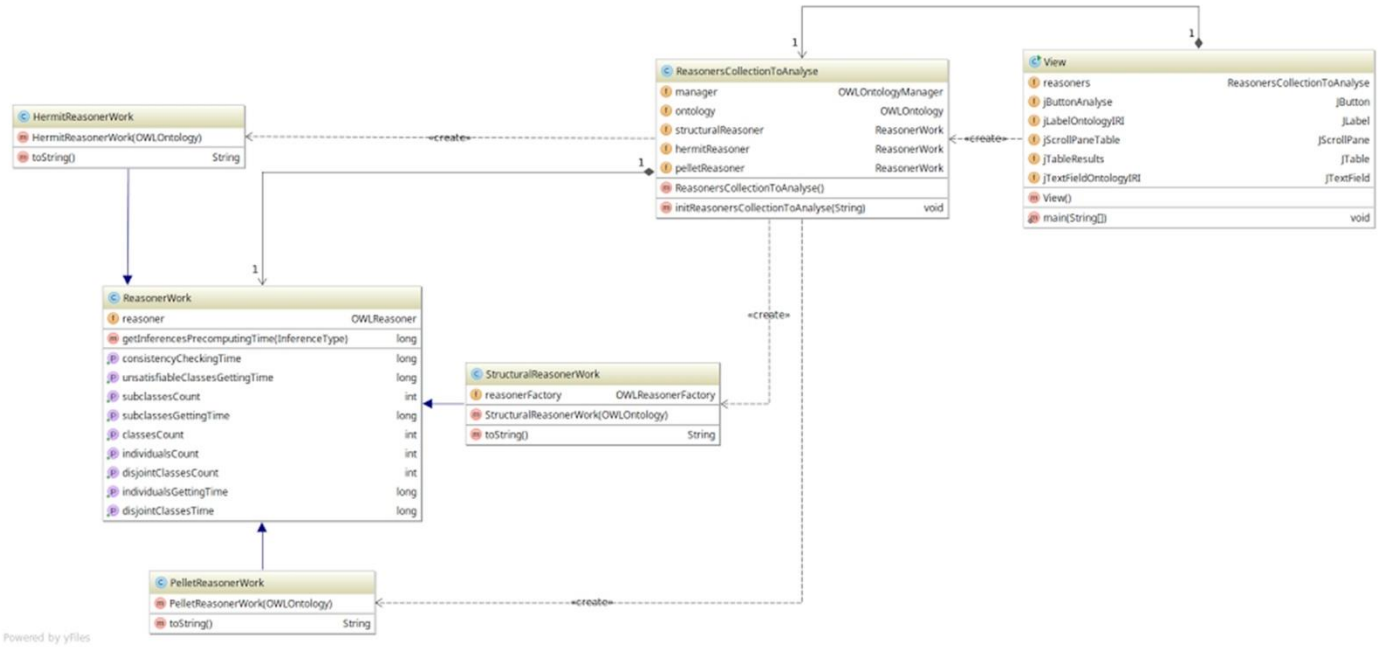


Рисунок 3.1 – UML-діаграма класів розробленого додатку

Графічний інтерфейс програми було реалізовано за допомогою інструментарію для створення графічного інтерфейсу користувача Swing.

Swing — платформо-незалежна бібліотека, що означає, що програму з використанням Swing можна запустити на всіх платформах, які підтримують JVM.

Swing — дуже розподілена архітектура, яка дозволяє «підключати» реалізації користувача вказаної інфраструктури інтерфейсів: користувачі можуть створити свою власну реалізацію цих компонентів, щоб замінити компоненти без обумовлення (за замовчуванням). Взагалі, користувачі Swing можуть розширити структуру, продовжуючи (з допомогою extends) існуючі класи і/або створюючи альтернативні реалізації основних компонентів.

У графічному інтерфейсі наявні

- поле для введення Internationalized Resource Identifier онтології;
- кнопка для запуску роботи різонерів;
- таблиця з результатами роботи різонерів.

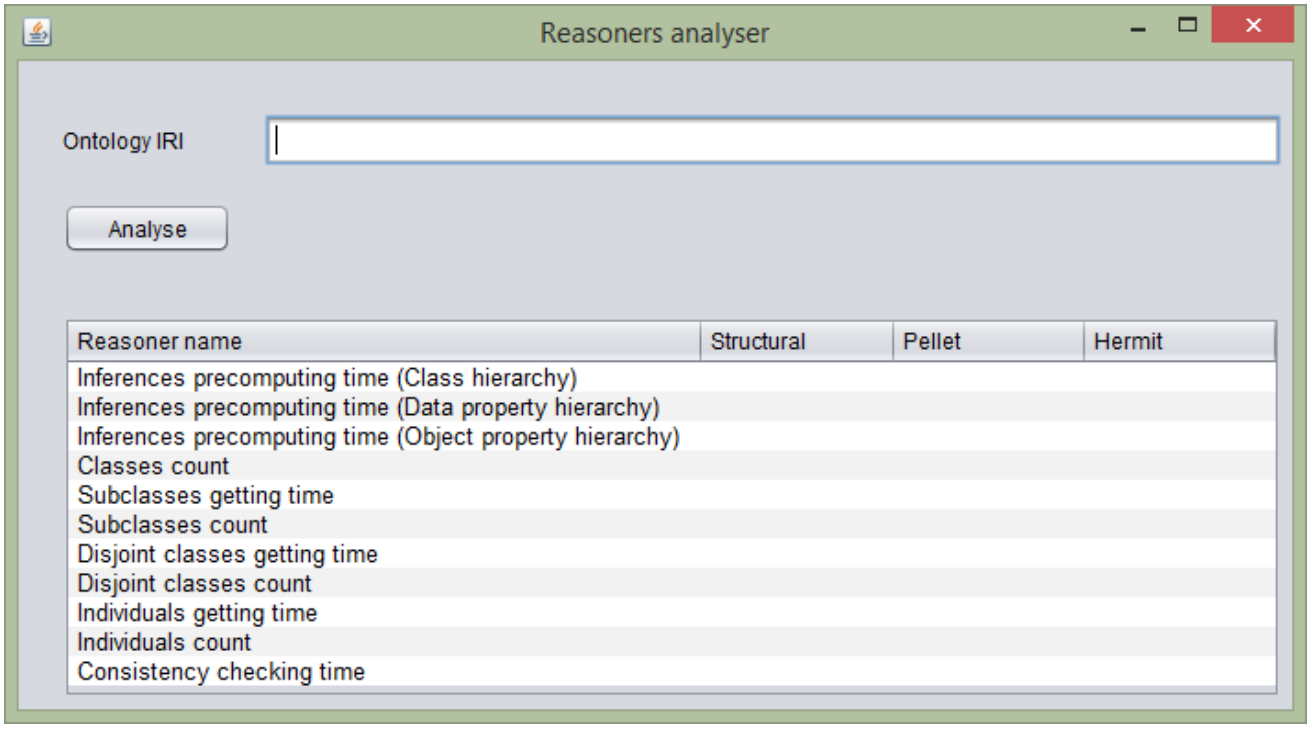


Рисунок 3.2 - Графічний інтерфейс розробленого додатку

Вихідний код програми наведений у Додатку А.

3.2 Тестування додатку

Для тестування додатку було вибрано шість онтологій, які описані детально нижче. Для кожної з них наведені:

- класифікаційний опис;
- ілюстрація фрагменту за допомогою редактора Protégé;
- результати роботи розробленого додатку.

3.2.1 Тестова онтологія №1

Першою тестовою онтологією вибрана термінологічна, одномовна, безекземплярна онтологія предметної області (молекулярна хімія) .

Доступна за адресою http://ontology.dumontierlab.com/molecule-complex

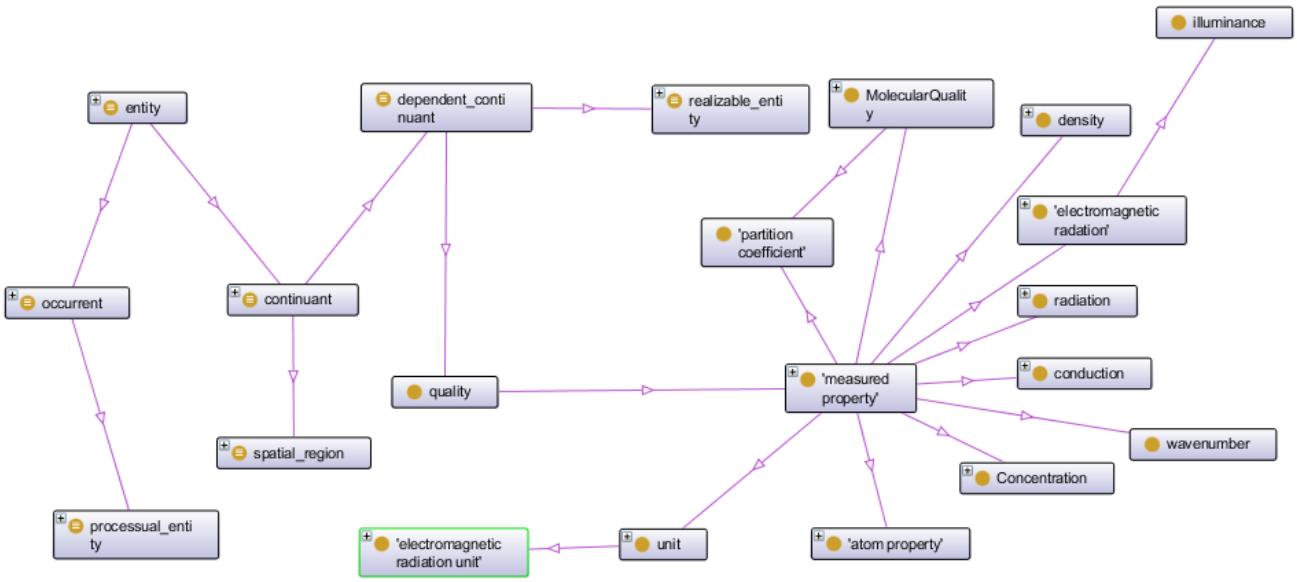
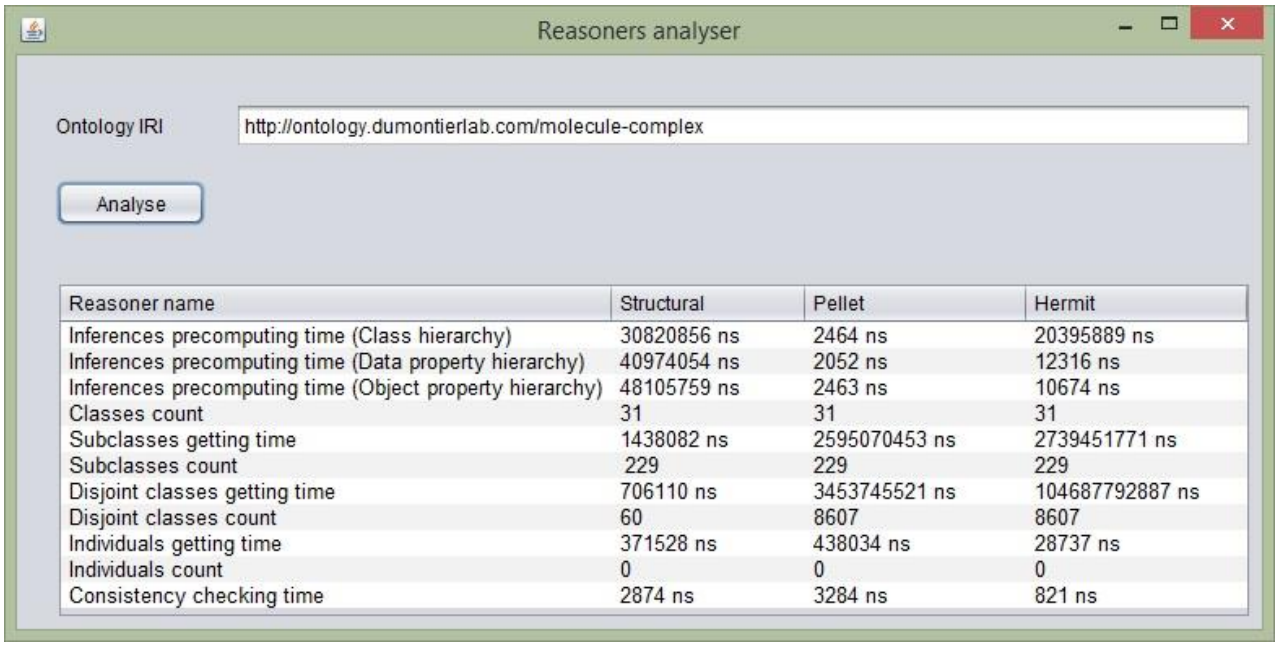


Рисунок 3.3 – Фрагмент тестової онтології №1



Reasoner name	Structural	Pellet	Hermit
Inferences precomputing time (Class hierarchy)	30820856 ns	2464 ns	20395889 ns
Inferences precomputing time (Data property hierarchy)	40974054 ns	2052 ns	12316 ns
Inferences precomputing time (Object property hierarchy)	48105759 ns	2463 ns	10674 ns
Classes count	31	31	31
Subclasses getting time	1438082 ns	2595070453 ns	2739451771 ns
Subclasses count	229	229	229
Disjoint classes getting time	706110 ns	3453745521 ns	104687792887 ns
Disjoint classes count	60	8607	8607
Individuals getting time	371528 ns	438034 ns	28737 ns
Individuals count	0	0	0
Consistency checking time	2874 ns	3284 ns	821 ns

Рисунок 3.4 – Результати роботи додатку для тестової онтології №1

3.2.2 Тестова онтологія №2

Другою тестовою онтологією вибрана термінологічна, одномовна, екземплярна онтологія предметної області (фізика) .

Доступна за адресою http://ontology.dumontierlab.com/unit-individuals

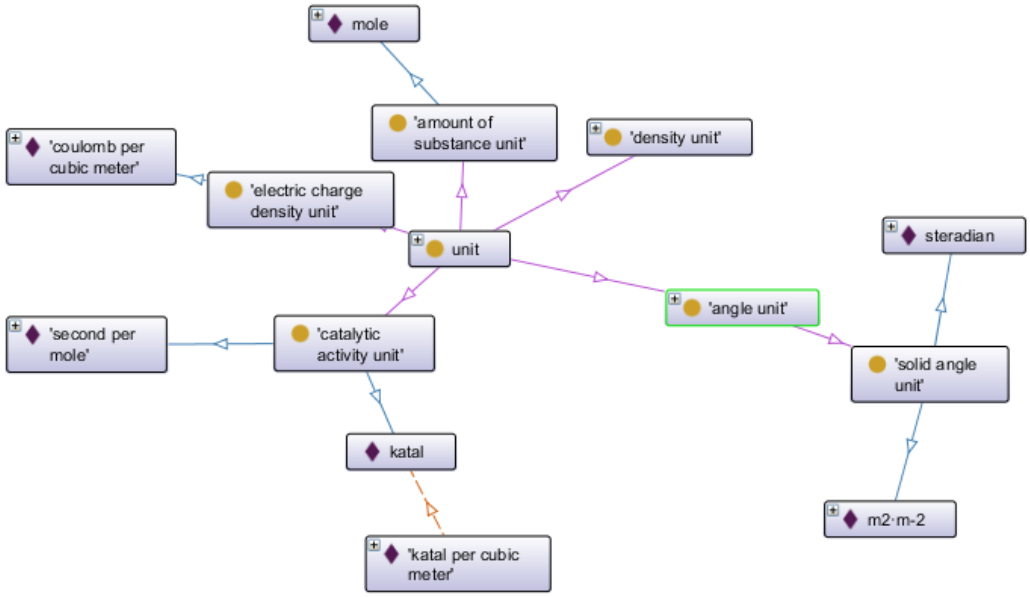
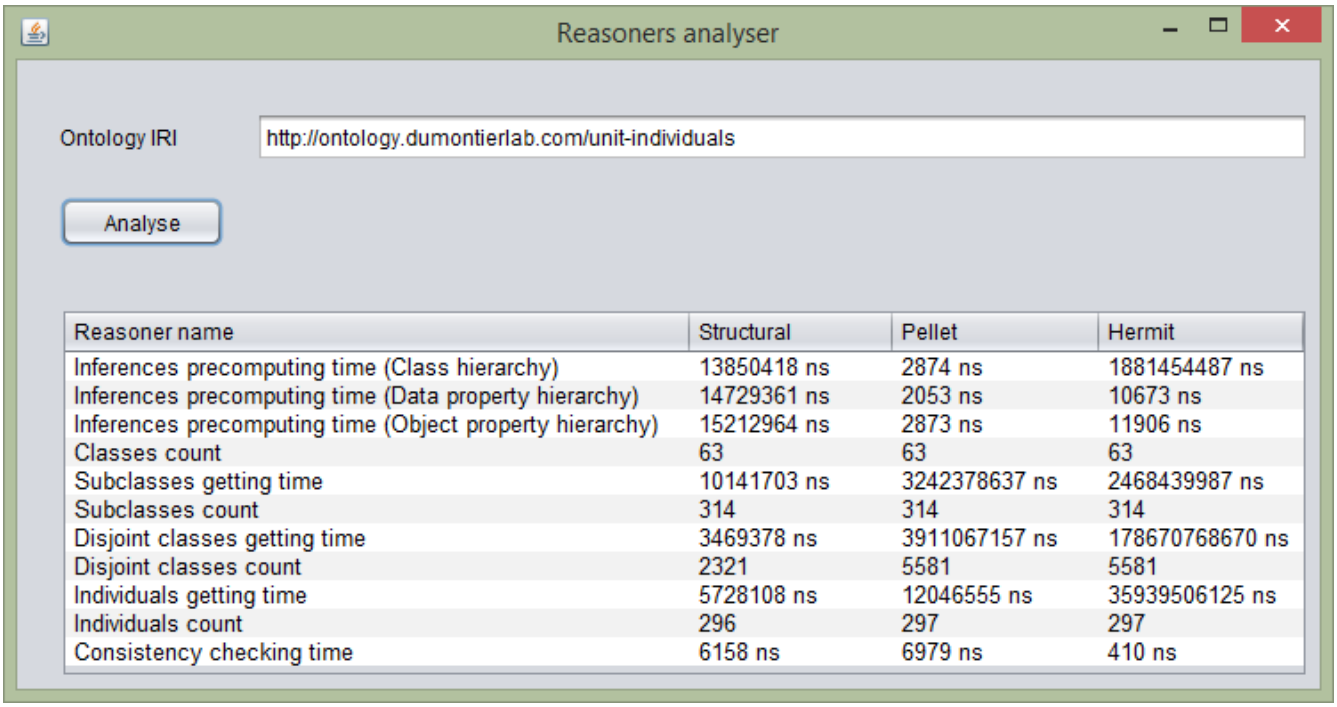


Рисунок 3.5 – Фрагмент тестової онтології №2



Reasoner name	Structural	Pellet	Hermit
Inferences precomputing time (Class hierarchy)	13850418 ns	2874 ns	1881454487 ns
Inferences precomputing time (Data property hierarchy)	14729361 ns	2053 ns	10673 ns
Inferences precomputing time (Object property hierarchy)	15212964 ns	2873 ns	11906 ns
Classes count	63	63	63
Subclasses getting time	10141703 ns	3242378637 ns	2468439987 ns
Subclasses count	314	314	314
Disjoint classes getting time	3469378 ns	3911067157 ns	178670768670 ns
Disjoint classes count	2321	5581	5581
Individuals getting time	5728108 ns	12046555 ns	35939506125 ns
Individuals count	296	297	297
Consistency checking time	6158 ns	6979 ns	410 ns

Рисунок 3.6 – Результати роботи додатку для тестової онтології №2

3.2.3 Тестова онтологія №3

Третьою тестовою онтологією вибрана термінологічна, одномовна, безекземплярна онтологія предметної області (одиниці виміру інтервалів часу) .

Доступна за адресою http://ontology.dumontierlab.com/time-interval-primitive

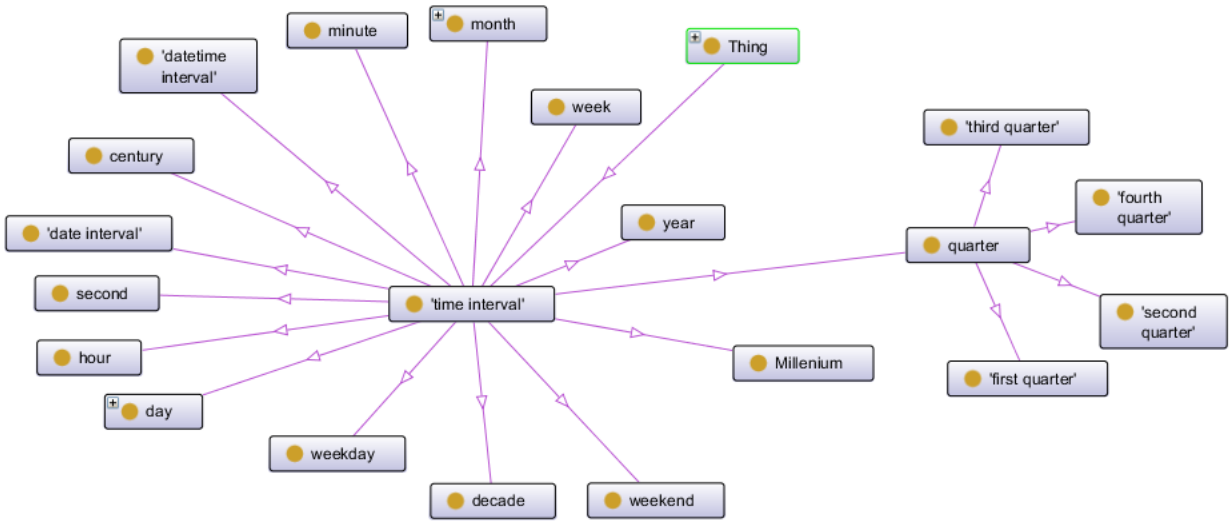
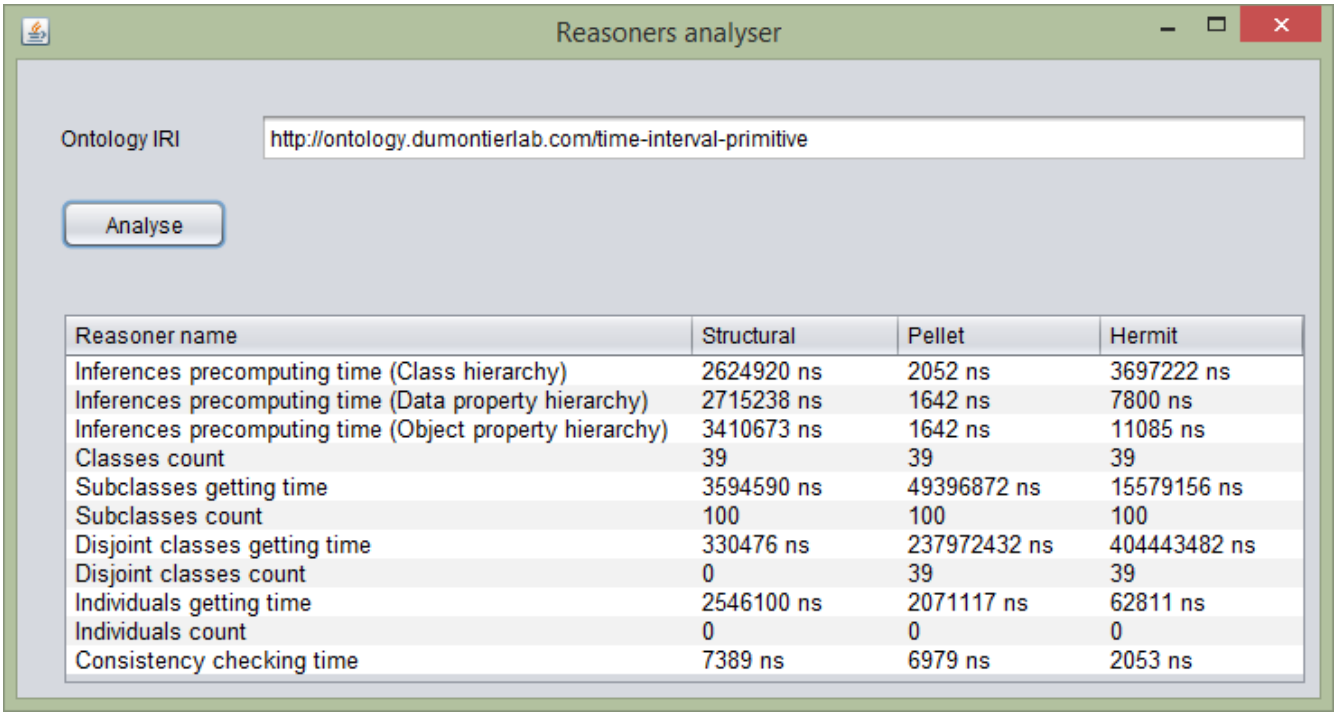


Рисунок 3.7 – Фрагмент тестової онтології №3



Reasoner name	Structural	Pellet	Hermit
Inferences precomputing time (Class hierarchy)	2624920 ns	2052 ns	3697222 ns
Inferences precomputing time (Data property hierarchy)	2715238 ns	1642 ns	7800 ns
Inferences precomputing time (Object property hierarchy)	3410673 ns	1642 ns	11085 ns
Classes count	39	39	39
Subclasses getting time	3594590 ns	49396872 ns	15579156 ns
Subclasses count	100	100	100
Disjoint classes getting time	330476 ns	237972432 ns	404443482 ns
Disjoint classes count	0	39	39
Individuals getting time	2546100 ns	2071117 ns	62811 ns
Individuals count	0	0	0
Consistency checking time	7389 ns	6979 ns	2053 ns

Рисунок 3.8 – Результати роботи додатку для тестової онтології №3

3.2.4 Тестова онтологія №4

Тетвертою тестовою онтологією вибрана «справжня», одномовна, екземплярна онтологія для опису навколишнього світу.

Доступна за адресою http://owl.man.ac.uk/2006/07/sssw/people.owl

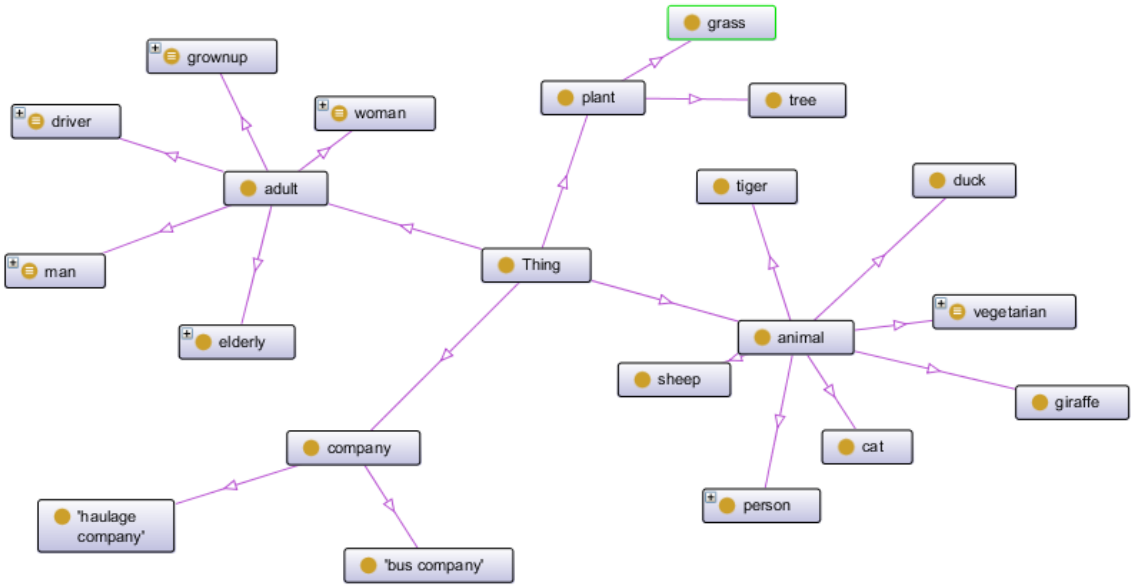
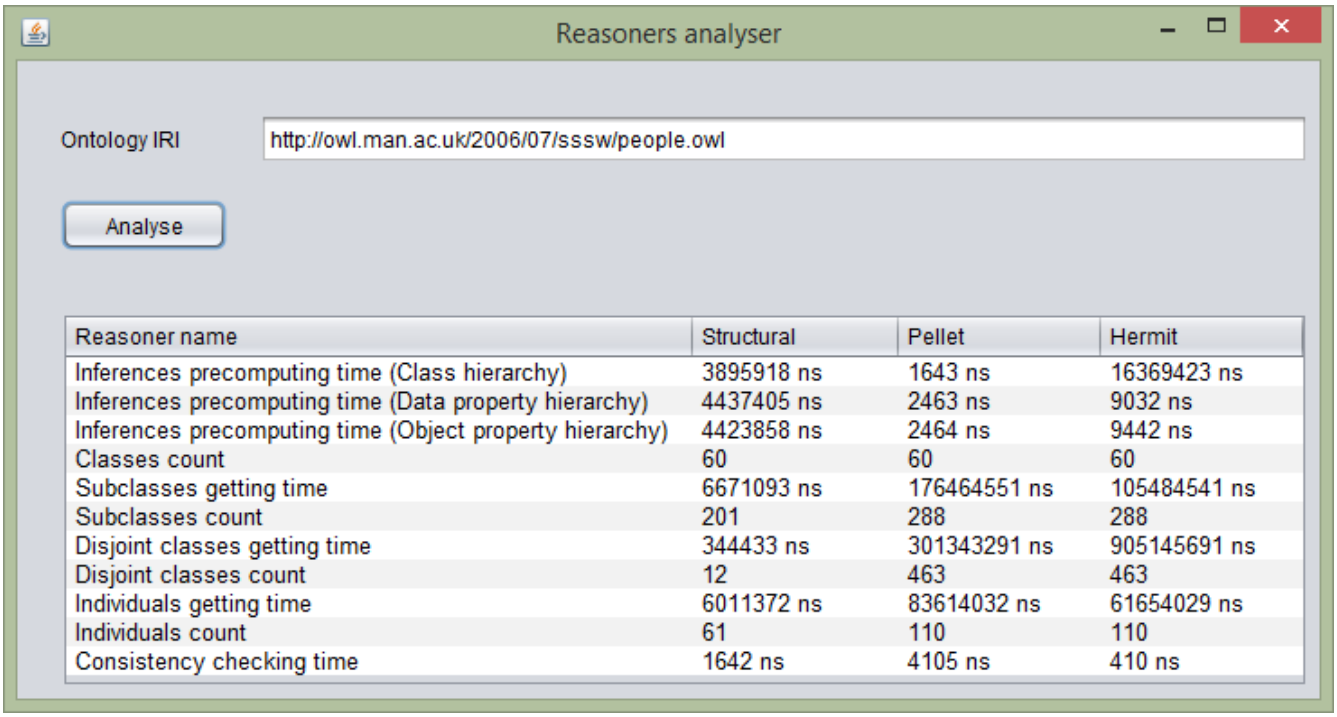


Рисунок 3.9 – Фрагмент тестової онтології №4



Reasoner name	Structural	Pellet	Hermit
Inferences precomputing time (Class hierarchy)	3895918 ns	1643 ns	16369423 ns
Inferences precomputing time (Data property hierarchy)	4437405 ns	2463 ns	9032 ns
Inferences precomputing time (Object property hierarchy)	4423858 ns	2464 ns	9442 ns
Classes count	60	60	60
Subclasses getting time	6671093 ns	176464551 ns	105484541 ns
Subclasses count	201	288	288
Disjoint classes getting time	344433 ns	301343291 ns	905145691 ns
Disjoint classes count	12	463	463
Individuals getting time	6011372 ns	83614032 ns	61654029 ns
Individuals count	61	110	110
Consistency checking time	1642 ns	4105 ns	410 ns

Рисунок 3.10 – Результати роботи додатку для тестової онтології №4

3.2.5 Тестова онтологія №5

П'ятою тестовою онтологією вибрана термінологічна, одномовна, безекземплярна онтологія предметної області (біохімія).

Доступна за адресою http://www.biopax.org/release/biopax-level1.owl

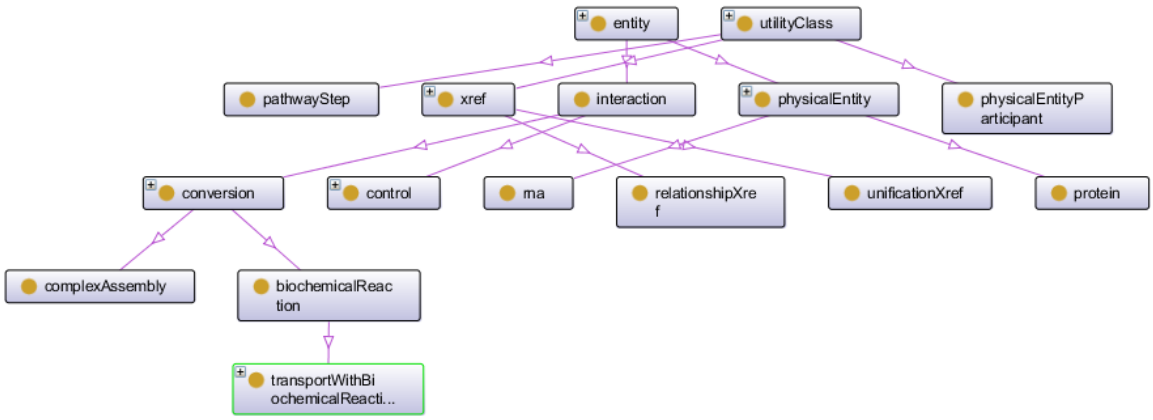
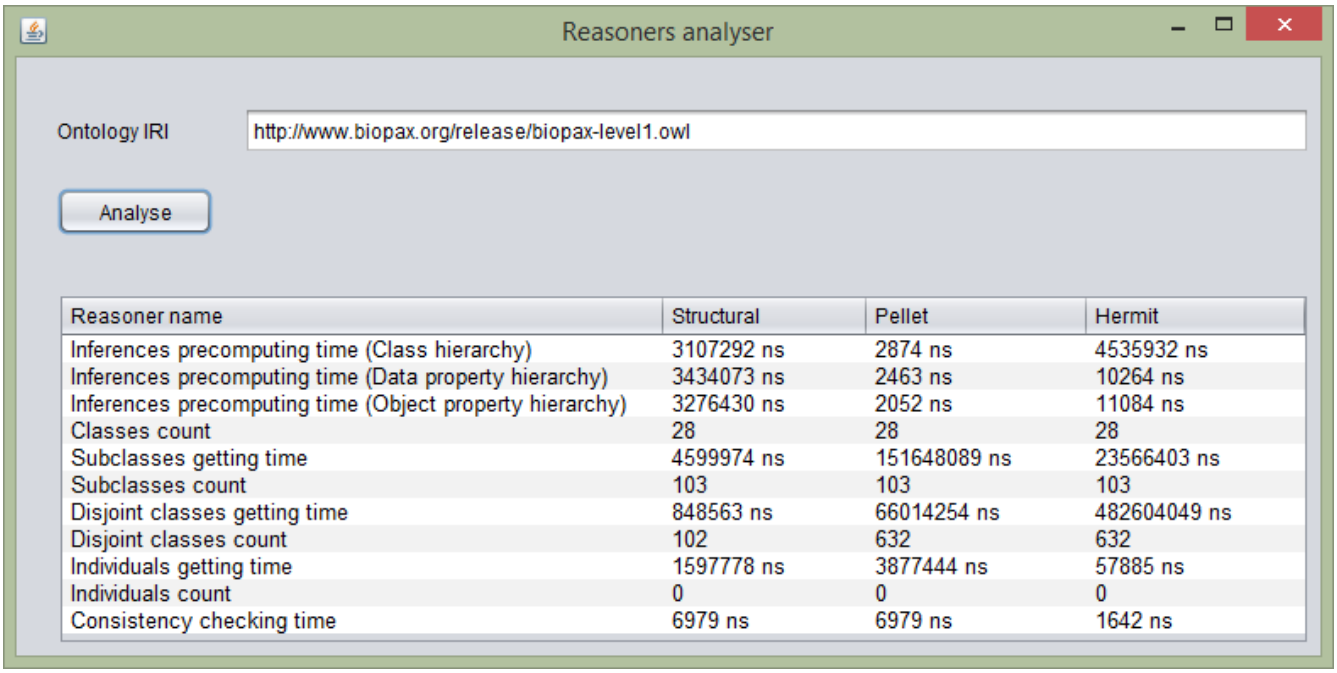


Рисунок 3.11 – Фрагмент тестової онтології №5



Reasoner name	Structural	Pellet	Hermit
Inferences precomputing time (Class hierarchy)	3107292 ns	2874 ns	4535932 ns
Inferences precomputing time (Data property hierarchy)	3434073 ns	2463 ns	10264 ns
Inferences precomputing time (Object property hierarchy)	3276430 ns	2052 ns	11084 ns
Classes count	28	28	28
Subclasses getting time	4599974 ns	151648089 ns	23566403 ns
Subclasses count	103	103	103
Disjoint classes getting time	848563 ns	66014254 ns	482604049 ns
Disjoint classes count	102	632	632
Individuals getting time	1597778 ns	3877444 ns	57885 ns
Individuals count	0	0	0
Consistency checking time	6979 ns	6979 ns	1642 ns

Рисунок 3.12 – Результати роботи додатку для тестової онтології №5

3.2.6 Тестова онтологія №6

Шостою тестовою онтологією вибрана термінологічна, одномовна, безекземплярна онтологія предметної області (фізика).

Доступна за адресою http://ontology.dumontierlab.com/physics-complex-1.0.owl

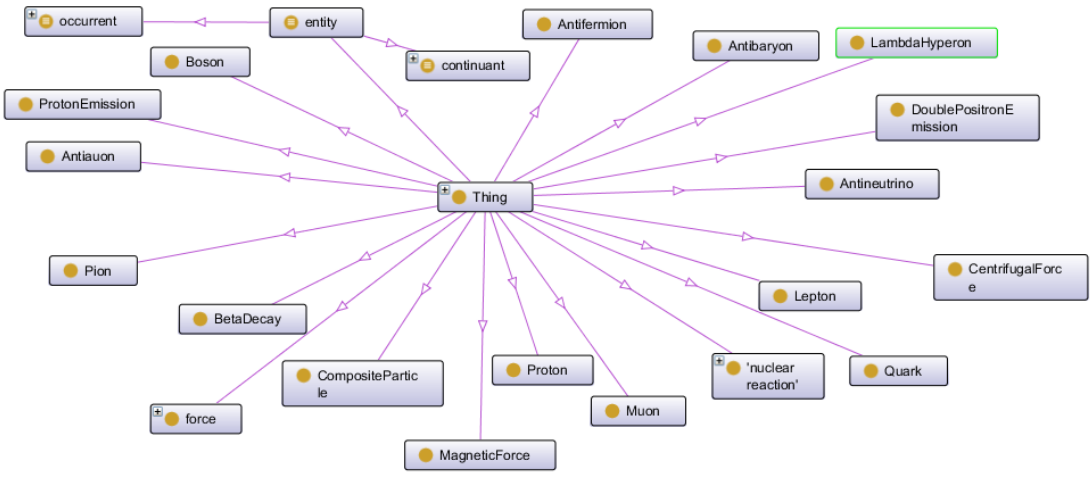
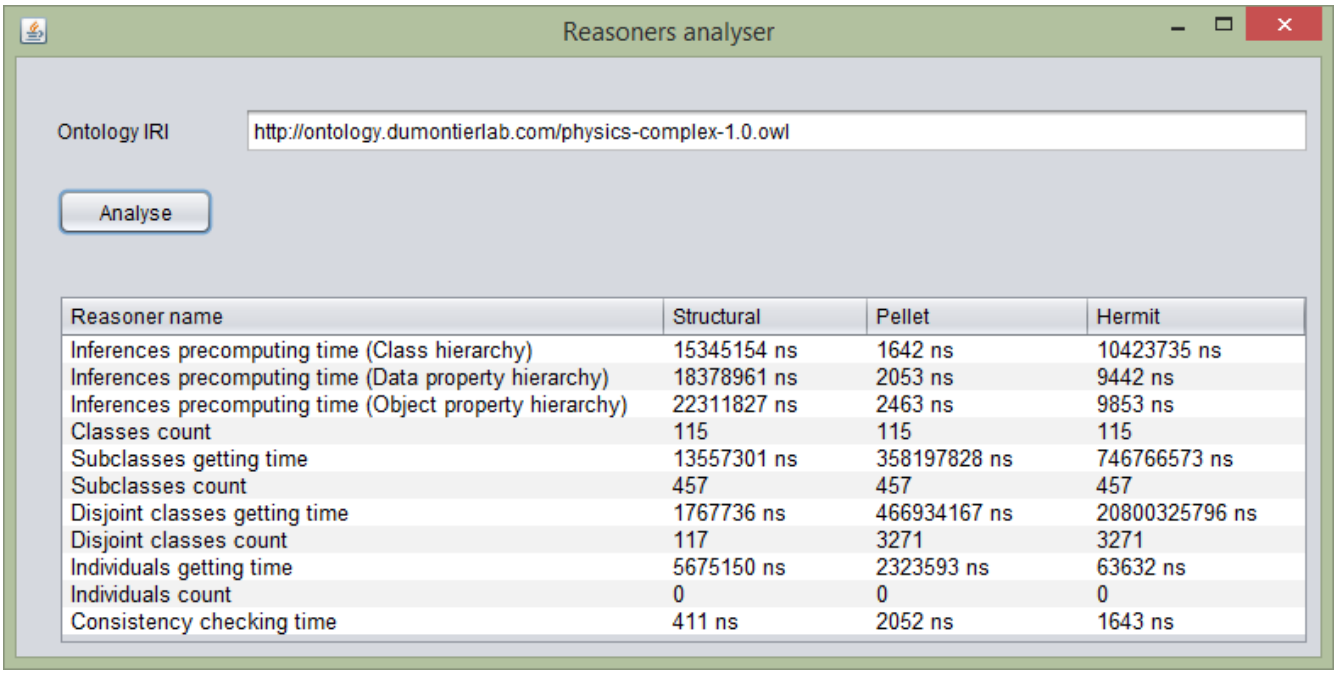


Рисунок 3.13 – Фрагмент тестової онтології №6



Reasoner name	Structural	Pellet	Hermit
Inferences precomputing time (Class hierarchy)	15345154 ns	1642 ns	10423735 ns
Inferences precomputing time (Data property hierarchy)	18378961 ns	2053 ns	9442 ns
Inferences precomputing time (Object property hierarchy)	22311827 ns	2463 ns	9853 ns
Classes count	115	115	115
Subclasses getting time	13557301 ns	358197828 ns	746766573 ns
Subclasses count	457	457	457
Disjoint classes getting time	1767736 ns	466934167 ns	20800325796 ns
Disjoint classes count	117	3271	3271
Individuals getting time	5675150 ns	2323593 ns	63632 ns
Individuals count	0	0	0
Consistency checking time	411 ns	2052 ns	1643 ns

Рисунок 3.14 – Результати роботи додатку для тестової онтології №6

3.2.7 Аналіз отриманих результатів

Отримані результати було зведено у таблицю, де рядки – це назви критеріїв тестування різонерів, стовпчики – назви тестових онтологій, для яких було проведено аналіз функцій різонерів.

На перетині стовпчиків та рядків названий різонер, що найшвидше (для часових критеріїв) або найкраще(для кількісних критеріїв) виконав ту чи іншу функцію.

Якщо ж для кількісних критеріїв були отримані однакові максимальні значення для різних різонерів, вони впорядковані у клітинці за часом виконання цих функцій(від меншого до більшого).

Таблиця 3.2 – Аналіз отриманих результатів роботи додатку

Найменування тестових онтологій / Найменування критеріїв оцінки різонерів	Тестова онтологія №1	Тестова онтологія №2	Тестова онтологія №3	Тестова онтологія №4	Тестова онтологія №5	Тестова онтологія №6
Час логічного виведення класової ієрархії	Pellet	Pellet	Pellet	Pellet	Pellet	Pellet
Час логічного виведення ієрархії властивостей даних	Pellet	Pellet	Pellet	Pellet	Pellet	Pellet
Час логічного виведення ієрархії властивостей об’єктів	Pellet	Pellet	Pellet	Pellet	Pellet	Pellet
Кількість всіх зв’язків онтології типу «клас- підклас»	Structural, Pellet, Hermit	Structural, Pellet, Hermit	Structural, Pellet, Hermit	Hermit, Pellet	Structural, Hermit, Pellet	Structural, Pellet, Hermit
Час отримання всіх зв’язків онтології типу «клас - підклас»	Structural	Structural	Structural	Structural	Structural	Structural
Кількість всіх зв’язків онтології типу «клас – непересічний клас»	Pellet, Hermit	Pellet, Hermit	Pellet, Hermit	Pellet, Hermit	Pellet, Hermit	Pellet, Hermit
Час отримання всіх зв’язків онтології типу «клас – непересічний клас»	Structural	Structural	Structural	Structural	Structural	Structural
Кількість всіх зв’язків онтології типу «клас – непересічний клас»	Structural, Pellet, Hermit	Pellet, Hermit	Structural, Pellet, Hermit	Hermit, Pellet	Structural, Pellet, Hermit	Hermit, Pellet, Structural
Час отримання всіх зв’язків онтології типу «індивідуал – клас»	Hermit	Structural	Hermit	Structural	Hermit	Hermit
Час перевірки онтології на консистентність	Hermit	Hermit	Hermit	Hermit	Hermit	Structural

Для всіх тестових онтологій найкоротший час виконання логічного виведення класової ієрархії, ієрархії властивостей даних та ієрархії властивостей об’єктів показав Pellet. Час отримання всіх всіх зв’язків «клас - підклас» онтології найкоротшим був для Structural різонера, але він для тестової онтології №4 виявив меншу кількість зв’язків, ніж Pellet та Hermit, що каже про менші можливості різонера у логічному виведенні класової ієрархії. Час отримання всіх зв’язків «клас – непересічний клас», найкоротшим також виявився для Structural, але кількість таких виявлених ним зв’язків у всіх шести випадках була меншою ніж у Pellet та Hermit. Серед останніх двох найкраще себе у плані часу виконання даної задачі виявив Pellet. Часу на те, щоб показати, що онтологія не має індивідуалів, найменше витратив Hermit, однак на вирахування їх кількості у екземплярних

онтологіях менше часу витратив Structural, але він знову ж таки, зв’язків «індивідуал – клас» він нарахував менше, ніж Pellet чи Hermit. Перевірка онтології на консистентність найменше часу у п’яти випадках з шести зайняла у Hermit-а, лише один раз у Structural.

Отже, бачимо, що оптимальним варіантом для виведення логічних фактів на прикладі вибраних тестових онтологій є Pellet – різонер, що реалізовує метод семантичних таблиць. Але перевірку на консистентність системи варто проводити, використовуючи Hermit. Structural reasoner виявився менш ефективним у виведенні зв’язків «індивідуал – клас», «клас - підклас», «клас – непересічний клас», адже у багатьох випадках кількість таких зв’язків була суттєво меншою за кількість зв’язків, яку вивели Pellet та Hermit.

3.3 Висновки

Було розроблено кросплатформний додаток на мові Java, який дає змогу перевірити швидкодію виконання основних функцій різонерів(Structural reasoner, Pellet, Hermit) на вибраній онтології.

У контексті даного розділу було проаналізовано архітектуру та функціонал даного додатку, наведено UML-діаграму класів, спроектованих для написання програмного коду, оглянуто основні технології та фреймворки, які було використано у ході розробки програми(OWL API, Java Maven, Java Swing).

Було вибрано шість онтологій для тестування додатку, зроблено їх огляд та аналіз результатів програми для кожної з них.

4 ЕКОНОМІЧНА ЧАСТИНА

У даному розділі проводиться оцінка основних характеристик програмного продукту, призначеного для аналізу основних функцій вибраних різонерів OWL-онтологій. Програмний продукт був розроблений за допомогою мови програмування Java.

Програмний продукт є кросплатформним та рекомендується для використання на персональних комп’ютерах під управлінням операційних систем Windows, Linux чи Mac. Для техніко-економічного аналізу програмного продукту буде використаний метод функціонально-вартісного аналізу (ФВА). Основою ФВА є функціональний підхід, згідно з яким об'єктом аналізу тобто не сам продукт, а функції, які він виконує. ФВА проводиться в два етапи: функціональний аналіз і вартісний аналіз.

Мета ФВА полягає у забезпеченні правильного розподілу ресурсів, виділених на виробництво продукції або надання послуг, на прямі та непрямі витрати. У даному випадку – аналізу функцій програмного продукту й виявлення усіх витрат на реалізацію цих функцій.

Фактично цей метод працює за таким алгоритмом:

– визначається послідовність функцій, необхідних для виробництва продукту. Спочатку – всі можливі, потім вони розподіляються по двом групам: ті, що впливають на вартість продукту і ті, що не впливають. На цьому ж етапі оптимізується сама послідовність скороченням кроків, що не впливають на цінність і відповідно витрат.

– для кожної функції визначаються повні річні витрати й кількість робочих часів.

– для кожної функції наоснові оцінок попереднього пункту визначається кількісна характеристика джерел витрат.

– після того, як для кожної функції будуть визначені їх джерела витрат, проводиться кінцевий розрахунок витрат на виробництво продукту.

4.1 Постановка задачі техніко-економічного обґрунтування

У роботі застосовується метод ФВА для проведення техніко-економічного обґрунтування розробки системи аналізу роботи reasoner-ів у OWL-онтологіях. Оскільки основні проектні рішення стосуються всієї системи, кожна окрема підсистема має їм задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для проведення аналізу роботи вибраних алгоритмів.

Відповідно цьому варто обирати і систему показників якості програмного продукту.

Технічні вимоги до продукту наступні:

– програмний продукт повинен функціонувати на персональних комп’ютерах із стандартним набором компонент;

– програмний продукт повинен бути кросплатформним;

– забезпечувати зручність і простоту взаємодії з користувачем або з розробником програмного забезпечення у випадку використовування його як модуля;

– передбачати мінімальні витрати на впровадження програмного продукту.

4.1.1 Обґрунтування функцій програмного продукту

Головна функція F_0– розробка програмного продукту, який аналізує процес за вхідними даними та будує його модель для подальшого прогнозування. Виходячи з конкретної мети, можна виділити наступні основні функції ПП:

F_1 – вибір мови програмування;

F_2 – використання готових бібліотек;

F_3 – інтерфейс користувача.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

а) мова програмування C#;

б) мова програмування Java;

Функція F_2:

а) написання алгоритмів вручну;

б) використання готових бібліотек;

Функція F_3:

а) інтерфейс користувача, створений за технологією Windows Forms;

б) інтерфейс користувача, створений за технологією Java Swing;

4.1.2 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1). На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (таблиця 4.1).

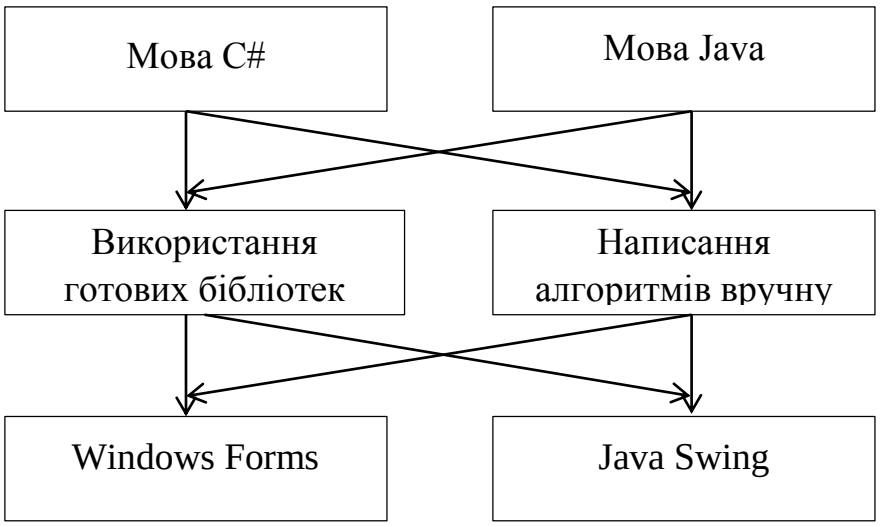


Рисунок 4.1 – Морфологічна карта

Морфологічна карта відображує всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

На основі аналізу позитивно-негативної матриці(табл. 4.1) робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам.

Таблиця 4.1 – Позитивно-негативна матриця

Основні функції	**Варіанти реалізації**	**Переваги**	**Недоліки**
F1	*А*	Займає менше часу при написанні коду	Не кросплатформна
	Б	Код швидко виконується, кросплатформна	Займає більше часу при написанні коду
F2	А	Забезпечує більшу гнучність у використанні	Займає більше часу при написанні коду
	Б	Займає менше часу при написанні коду	Забезпечує меншу гнучкість у використанні
F3	*А*	Простота створення графічних об’єктів	Відсутня можливість використання на інших платформах
	Б	Простота створення графічних об’єктів	Відсутня можливість використання на інших платформах

Функція *F1:*

Оскільки продукт повинен бути кросплатформним, варіант а) має бути відкинутим.

Функція *F2:*

Оскільки методи написані вручну та за допомогою готових бібліотек будуть давати однакові результати, вважаємо варіанти а) та б) гідними розгляду.

Функція *F3:*

Оскільки з даних варіантів для платформи Java підходить лише б), то варіант а) необхідно відкинути.

Таким чином, будемо розглядати такі варіанти реалізації ПП:

1. F1б – F2а – F3б
2. F1б – F2б – F3б

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.2 Обґрунтування системи параметрів ПП

4.2.1 Опис параметрів

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього, визначаються основні параметри виробу, що будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

– *X1* – швидкодія мови програмування;

– *X2* – об’єм пам’яті для збереження даних;

– *X3* – час обробки даних;

– *X4* – потенційний об’єм програмного коду.

X1: Відображає швидкодію операцій залежно від обраної мови програмування.

X2: Відображає об’єм пам’яті в оперативній пам’яті персонального комп’ютера, необхідний для збереження та обробки даних під час виконання програми.

X3: Відображає час, який витрачається на дії.

X4: Показує розмір програмного коду який необхідно створити безпосередньо розробнику.

4.2.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у табл. 4.2.

Таблиця 4.2 - Основні параметри ПП

Назва Параметра	**Умовні позначення**	**Одиниці виміру**	**Значення параметра**		
			гірші	**середні**	**кращі**
Швидкодія мови програмування	X1	Оп/мс	2000	11000	19000
Об’єм пам’яті для збереження даних	X2	Мб	32	16	8
Час обробки даних алгоритмом	X3	мс	800	420	60
Потенційний об’єм програмного коду	X4	кількість строк коду	2000	1500	1000

За даними таблиці 4.2 будуються графічні характеристики параметрів – рис. 4.2 – рис. 4.5.

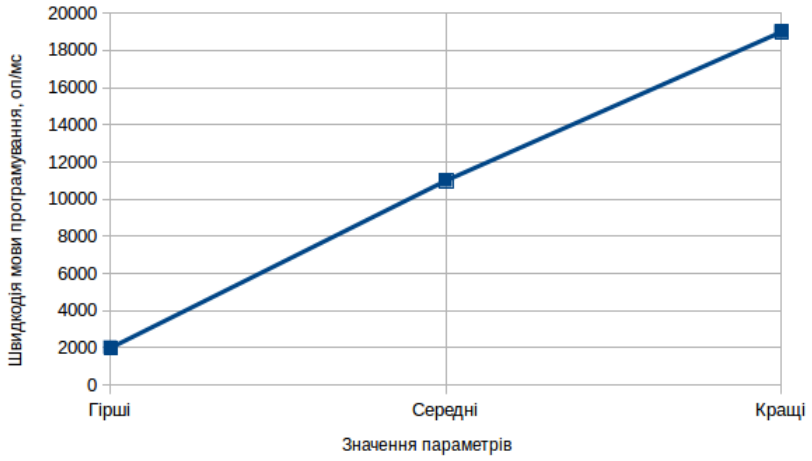


Рисунок 4.2 – X1, швидкодія мови програмування

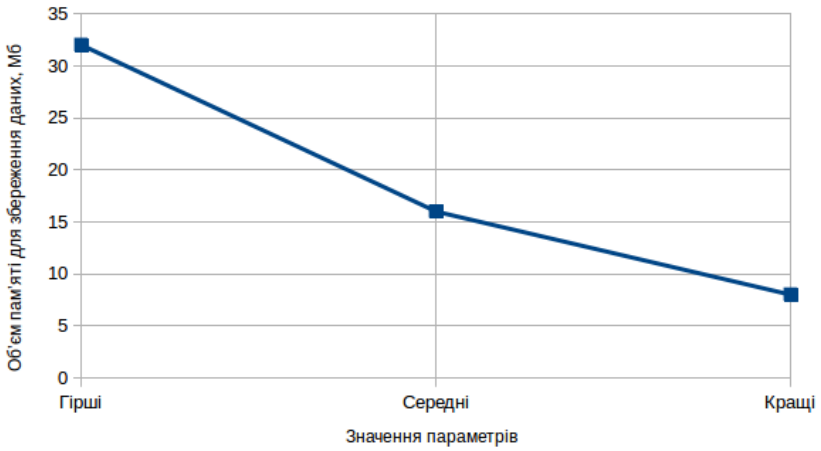


Рисунок 4.3 – X2, об’єм пам’яті для збереження даних

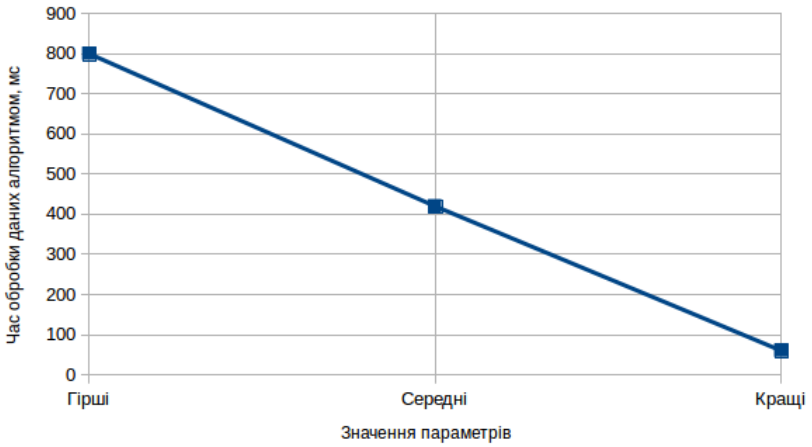


Рисунок 4.4 – X3, час обробки даних алгоритмом

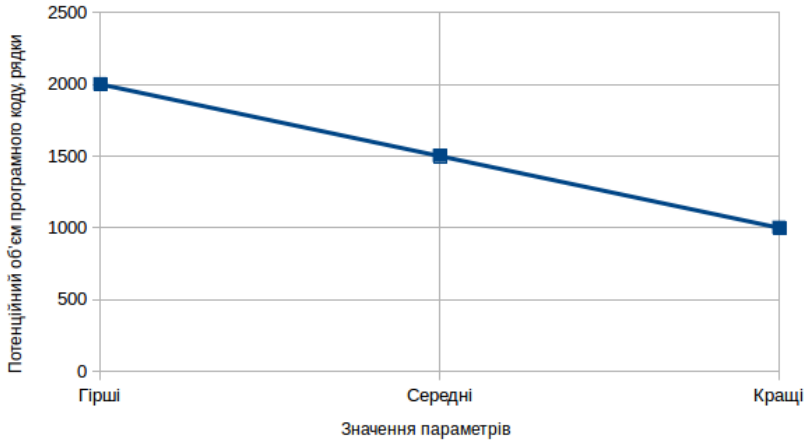


Рисунок 4.5 – X4, потенційний об’єм програмного коду

4.2.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Таблиця 4.3 – Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія мови програмування	Оп/мс	4	3	4	4	4	4	4	27	0.75	0.56
X2	Об’єм пам’яті для збереження даних	Мб	4	4	4	3	4	3	3	25	-1.25	1.56
X3	Час обробки даних алгоритмом	Мс	2	2	1	2	1	2	2	12	-14.25	203.06
X4	Потенційний об’єм програмного коду	кількість строк коду	5	6	6	6	6	6	6	41	14.75	217.56
	Разом		15	15	15	15	15	15	15	105	0	420.75

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

– визначення рівня значимості параметра шляхом присвоєння різних рангів;

– перевірку придатності експертних оцінок для подальшого використання;

– визначення оцінки попарного пріоритету параметрів;

– обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.3.

Для перевірки ступеню достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 105, \quad (4.1)$$

де N – число експертів, n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 26.25 \quad (4.2)$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T \quad (4.3)$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 420.75 \quad (4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3-n)} = \frac{12 \cdot 420.75}{7^2(5^3-5)} = 1{,}03 > W_k = 0.67 \quad (4.5)$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0.67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 4.4.

Таблиця 4.4 – Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	=	>	=	<	=	<	<	<	0.5
X1 і X3	<	<	<	<	<	<	<	<	0.5
X1 і X4	>	>	>	>	>	>	>	>	1.5
X2 і X3	<	<	<	<	<	<	<	<	0.5
X2 і X4	>	>	>	>	>	>	>	>	1.5
X3 і X4	>	>	>	>	>	>	>	>	1.5

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases} \quad (4.6)$$

З отриманих числових оцінок переваги складемо матрицю $A = \| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{ві}$ за наступними формулами:

$$K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i}, \text{де } b_i = \sum_{i=1}^{N} a_{ij} \quad (4.7)$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%).На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{ві}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \text{де } b_i' = \sum_{i=1}^{N} a_{ij} b_j \quad (4.8)$$

Як видно з таблиці 4.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.5 – Розрахунок вагомості параметрів

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$	b_i^2	$K_{\text{ві}}^2$
X1	1.0	0.5	0.5	1.5	3.5	0.219	22.25	0.216	100	0.215
X2	1.5	1.0	0.5	1.5	4.5	0.281	27.25	0.282	124.25	0.283
X3	1.5	1.5	1.0	1.5	5.5	0.344	34.25	0.347	156	0.348
X4	0.5	0.5	0.5	1.0	2.5	0.156	14.25	0.155	64.75	0.154
Всього:					16	1	98	1	445	1

4.3 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X2*(об’єм пам’яті для збереження даних) *X1* (швидкодія мови програмування) та *X3* відповідають технічним вимогам умов функціонування даного ПП.1

Абсолютне значення параметра *X4 (*потенційний об’єм програмного коду) обрано не найгіршим (не максимальним), тобто це значення відповідає або варіанту а) 1800 або варіанту б) 1200.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 4.6):

$$K_K(j) = \sum_{i=1}^{n} K_{вi,j} B_{i,j}\,, \qquad (4.9)$$

де n – кількість параметрів;$K_{вi}$– коефіцієнт вагомості i–го параметра;B_i – оцінка i–го параметра в балах.

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1(X1)	А	11000	3.6	0.215	0.774
F2(X3)	А, Б	800	2.4	0.348	0.835
F2(X4)	А	1800	2	0.154	0.308
	Б	1200	8	0.154	1.232
F3(X2)	Б	16	3.4	0.283	0.962

За даними з таблиці 4.6 за формулою

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}], \qquad (4.10)$$

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 0.774 + 0.835 + 0.308 + 0.962 = 2.879$$

$$K_{K2} = 0.774 + 0.835 + 1.232 + 0.962 = 3.803$$

Як видно з розрахунків, кращим є другий варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.4 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

При цьому варіант 3 має додаткове завдання:

3. Реалізація методів аналізу;

А варіант 4 має інше додаткове завдання:

4. Обробка інтерфейсу готових бібліотек.

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 2.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється як

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}, \quad (4.11)$$

де T_P – трудомісткість розробки ПП;$K_П$ – поправочний коефіцієнт;$K_{СК}$ – коефіцієнт на складність вхідної інформації; K_M – коефіцієнт рівня мови програмування;$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: T_P =90 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $K_П$ = 1.7. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для

всіх завдань рівний 1: $К_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $К_{СТ}$ = 0.8. Тоді, за формулою 5.1, загальна трудомісткість програмування першого завдання дорівнює:

$$Т_1 = 90 \cdot 1.7 \cdot 0.8 = 122.4 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм другої групи складності, ступінь новизни Б), тобто $Т_Р$ =27 людино-днів, $К_П$ =0.9,$К_{СК}$ = 1,$К_{СТ}$ =0.8:

$$Т_2 = 27 \cdot 0.9 \cdot 0.8 = 19.44 \text{ людино-днів.}$$

Для третього завдання (використовується алгоритм другої групи складності, ступінь новизни Г з використанням перемінної інформації):

$$Тр = 12 \text{ людино-днів;}$$

$$Кп = 0.72; \quad Кст = 0.8;$$

$$То = 12 \cdot 0.72 \cdot 0,8 = 6.91.$$

Для четвертого завдання (використовується алгоритм третьої групи складності, ступінь новизни Г):

$$Тр = 8 \text{ людино-днів;}$$

$$Кп = 0.6; \quad Кст = 1;$$

$$То = 8 \cdot 0.6 \cdot 1 = 4.8.$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$Т_I = (122.4 + 19.44 + 6.91) \cdot 8 = 1190 \text{ людино-годин;}$$

T_{II} = (122.4 + 19.44 + 4.8) · 8 = 1173.12 людино-годин;

Найбільш високу трудомісткість має варіант I.

В розробці беруть участь два програмісти з окладом 6000 грн., один фінансовий аналітик з окладом 9000 грн. Визначимо зарплату за годину за формулою:

$$С_ч = \frac{М}{T_m \cdot t} грн., \quad (4.12)$$

де М – місячний оклад працівників; T_m – кількість робочих днів тиждень; t – кількість робочих годин в день.

$$С_ч = \frac{6000 + 6000 + 9000}{3 \cdot 21 \cdot 8} = 41.67 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за формулою

$$С_{ЗП} = С_ч \cdot T_i \cdot К_Д, \quad (4.13)$$

де $С_ч$– величина погодинної оплати праці програміста;T_i – трудомісткість відповідного завдання; $К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП}$ = 41.67· 1190 · 1.2 = 59504.76 грн.

II. $С_{ЗП}$ = 41.67· 1173.12 · 1.2 = 58660.69 грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22$ = 59504.76· 0.22 = 13091.05 грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22$ = 58660.69 · 0.22 = 12905.35 грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 6000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 6000 \cdot 0.2 = 14400 \text{ грн.} \quad (4.14)$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 14400 \cdot (1 + 0.2) = 17280 \text{ грн.} \quad (4.15)$$

Відрахування на єдиний соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 17280 \cdot 0.22 = 3801.6 \text{ грн.} \quad (4.16)$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 8000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 8000 = 2300 \text{ грн.}, \quad (4.17)$$

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача; $К_А$– річна норма амортизації; $Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.15 \cdot 8000 \cdot 0.05 = 460 \text{ грн.}, \quad (4.18)$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 8 – 16) \cdot 8 \cdot 0.9 = 1706.4 \text{годин}, \quad (4.19)$$

де $Д_К$ – календарна кількість днів у році; $Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів; $Д_Р$ – кількість днів планових ремонтів устаткування; t – кількість робочих годин в день; $К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1706.4 \cdot 0.156 \cdot 0.2 \cdot 2.0218 = 107.64 \text{ грн.}, \quad (4.20)$$

де $N_С$ – середньо-споживча потужність приладу; $К_З$– коефіцієнтом зайнятості приладу; $Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0.67 = 8000 \cdot 0.67 = 5360 \text{ грн.} \quad (4.21)$$

Тоді, річні експлуатаційні витрати будуть:

$$C_{ЕКС} = C_{ЗП} + C_{ВІД} + C_{А} + C_{Р} + C_{ЕЛ} + C_{Н} \quad (4.22)$$

$$C_{ЕКС} = 17280 + 3801.6 + 2300 + 460 + 107.64 + 5360 = 29309.24 \text{ грн.} \quad (4.23)$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{М\text{-}Г} = C_{ЕКС} / T_{ЕФ} = 29309.24 / 1706.4 = 18.1 \text{ грн/час.} \quad (4.24)$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$C_{М} = C_{М\text{-}Г} \cdot T \quad (4.25)$$

I. $C_{М} = 18.1 \cdot 1190 = 21539$ грн.;

II. $C_{М} = 18.1 \cdot 1173.12 = 21233.472$ грн.;

Накладні витрати складають 67% від заробітної плати:

$$C_{Н} = C_{ЗП} \cdot 0.67 \quad (4.26)$$

I. $C_{Н} = 59504.76 \cdot 0.67 = 39868.19$ грн.;

II. $C_{Н} = 58660.69 \cdot 0.67 = 39302.66$ грн.;

Отже, вартість розробки ПП за варіантами становить:

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_{М} + C_{Н} \quad (4.27)$$

I. $C_{ПП} = 59504.76 + 13091.05 + 21539 + 39868.19 = 134002$ грн.;

II. $C_{ПП} = 58660.69 + 12905.35 + 21233.47 + 39302.66 = 132102.17$ грн.;

4.2 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕРj} = К_{Кj} / С_{Фj}, \quad (4.28)$$

$К_{ТЕР1} = 2.879 / 134002 = 0.21 \cdot 10^{-4}$;

$К_{ТЕР2} = 3.803 / 132102.17 = 0.29 \cdot 10^{-4}$;

Як бачимо, найбільш ефективним є другий варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 0.29 \cdot 10^{-4}$.

4.5 Висновки

В даному розділі проведено повний функціонально-вартісний аналіз ПП, який було розроблено в рамках дипломного проекту. Процес аналізу можна умовно розділити на дві частини.

В першій з них проведено дослідження ПП з технічної точки зору: було визначено основні функції ПП та сформовано множину варіантів їх реалізації; на основі обчислених значень параметрів, а також експертних оцінок їх важливості було обчислено коефіцієнт технічного рівня, який і дав змогу визначити оптимальну з технічної точки зору альтернативу реалізації функцій ПП.

Другу частину ФВА присвячено вибору із альтернативних варіантів реалізації найбільш економічно обґрунтованого. Порівняння запропонованих варіантів реалізації в рамках даної частини виконувалось за коефіцієнтом ефективності, для обчислення якого були обчислені такі допоміжні параметри, як трудомісткість, витрати на заробітну плату, накладні витрати.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є другий варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості $К_{ТЕР} = 0.29 \cdot 10^{-4}$.

Цей варіант реалізації програмного продукту має такі параметри:

– мова програмування – Java;

– використання готових бібліотек;

– інтерфейс користувача, створений за технологією Java Swing;

ВИСНОВКИ

Для детального ознайомлення з предметною областю дослідження було проведено огляд ключових ідеї побудови Семантичної павутини, семантичних відношень, особливостей визначення та структури онтологій у контексті комп'ютерних наук, їх класифікацій.

Було проаналізовано мову OWL для опису онтологій семантичної павутини, охарактеризовано компоненти, з яких зазвичай складаються онтології, описані за допомогою OWL, їхні особливості та суть застосування.

Розглянуто основні різновиди мови OWL(OWL Lite , OWL DL, OWL Full), проаналізовано можливості кодного з них. Також продемонстровано особливості синтаксисів, які підтримує мова (OWL2 Functional, OWL2 XML, Манчестерський, RDF/XML, RDF/Turtle).

Було проаналізовано поняття семантичного різонера, основну концепцію його побудови(Open World Reasoning - категорію, що позначає відкритість баз знань, яка заснована на припущенні про відкритість світу (open world assumption), що принципово відрізняє їх від баз даних, заснованих на припущення про замкнутість світу (closed world assumption)).Також було виділено головні задачі різонерів.

Було проаналізовано принципи роботи таких алгоритмів: прямого виведення, зворотного виведення, побудови семантичних таблиць, побудови гіпертаблиць.

Було здійснено огляд існуючих різонерів, особливостей їх реалізації.

В якості практичної частини дипломної роботи було розроблено кросплатформний додаток на мові Java, який дає змогу перевірити швидкодію та якість виконання основних функцій різонерів(Structural reasoner, Pellet, Hermit) для вказаної онтології, описаної на мові OWL. Додаток є відкритим для удосконалення, його архітектура дозволяє додавати для аналізу будь-які різонери, що реалізують

інтерфейс OWLReasoner фреймворку OWL API. Завантаження необхідних бібліотек не є проблемою, оскільки програма реалізована у вигляді Java Maven.

Було проаналізовано архітектуру та функціонал розробленого додатку, наведено UML-діаграму класів, спроектованих для написання програмного коду, оглянуто основні технології та фреймворки, які було використано у ході розробки програми(OWL API, Java Maven, Java Swing).

Було вибрано шість онтологій для тестування додатку, зроблено їх огляд та аналіз результатів програми для кожної з них. Для розглянутих онтологій було визначено, що оптимальним варіантом для виведення логічних фактів є Pellet – різонер, що реалізовує метод семантичних таблиць. Але перевірку на консистентність системи варто проводити, використовуючи Hermit – різонер, що реалізовує алгоритм гіпертаблиць. Structural reasoner виявився найменш ефективним у виведенні зв’язків «індивідуал – клас», «клас - підклас», «клас – непересічний клас», адже у багатьох випадках кількість таких зв’язків була суттєво меншою за кількість зв’язків, яку вивели Pellet та Hermit.

Розроблений додаток дає змогу користувачеві вибрати оптимальний різонер для виконання кожної з функцій над конкретною онтологією в залежності від її цілей, архітектури, кількості явних і неявних фактів т.д. Це забезпечує кращу швидкодію роботи з базою знань.

ПЕРЕЛІК ПОСИЛАНЬ

1. Константинова Н.С. Онтологии как системы хранения знаний / Константинова Н.С. , Митрофанова О.А. - Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению "Информационно-телекоммуникационные системы", 2008. - 54 с.
2. Gardiner Tom. Automated Benchmarking of Description Logic Reasoners / Gardiner Tom, Ian Horrocks, Dmitry Tsarkov. - Description Logics Workshop 2006. – 8 с.
3. Guarino N. Understanding, Building, and Using Ontologies / Guarino N. - Режим доступу: http://ksi.cpsc.ucalgary.ca/KAW/KAW96/guarino/guarino.html. - Дата доступу: 18.05.2016
4. Hovy E. Презентации Ontologies: lecture 1 , lecture 2 Issues of Content, lecture 3 Methods for Automated Ontology Building / Hovy E. - Information Sciences Institute University of Southern California, 2005. – 65 с.
5. Hovy E. A Standard for Large Ontologies / Hovy E. - Режим доступу: http://www.isi.edu/nsf/papers/hovy2.htm. – Дата доступу: 20.05.2016
6. Коваль С.А. Безэкземплярные и экземплярные онтологии / Коваль С.А. - Материалы XXXVI Междунар. филологической конференции в Санкт-Петербургском университете, 2007. - Режим доступу: http://skowal.narod.ru/research/ontology2007. - Дата доступу: 20.05.2016
7. Roussopoulos N.D. A semantic network model of data bases / Roussopoulos N.D. — TR No 104, Department of Computer Science, University of Toronto, 1976. – 126 с.
8. Hayes, P. OWL Web Ontology Language Semantics and Abstract Syntax / Hayes, P., Horrocks, I., Patel-Schneider, P.F. - W3C Recommendation, 10 February 2004. – 136 с.
9. Cornet R. Non-standard reasoning services for the debugging of description logic terminologies. / Cornet R., Schlobach S. - Gottlob, G., Walsh, T., eds.: IJCAI, Morgan Kaufmann , 2003. - 529 с.

10. Bock J. Benchmarking OWL Reasoners / Bock J., Haase P., Ji Q., Volz R. - ARea2008 - Workshop on Advancing Reasoning on the Web: Scalability and Commonsense, 2008. – 104 с.
11. Schalkoff R. Intelligent Systems: Principles, Paradigms and Pragmatics / Schalkoff R. - Jones & Bartlett Learning, 2009. – 762 с.
12. Hayes-Roth F. - Building Expert Systems / Hayes-Roth F., Waterman D., Lenat D. - Addison-Wesley, 1983. – 254 с.
13. Kaczor K. Overview of Expert System Shells / Kaczor K., Szymon B., Grzegorz J. - Krakow, Poland: Institute of Automatics: AGH University of Science and Technology, Poland, 5 December 2010. – 334 с.
14. Бет Э. Математическая теория логического вывода / Бет Э. – М.: Наука, 1967. – 523 с.
15. Эдельман С.Л. Математическая логика / Эдельман С.Л. – М.: Высшая школа, 1975. – 423 с.
16. Драгалин А.Г. Введение в математическую логику / Драгалин А.Г., Колмогоров А.Н. – М.: МГУ, 1982. - 120 с.
17. Hähnle, R. Tableaux and Related Methods. Handbook of Automated Reasoning / Hähnle, R. – Volume I. Elsevier science, 2001. – 277 с.
18. Baumgartner P. Hyper Tableaux / Baumgartner P., Furbach U. - Niemela Universitat Koblenz Institut für Informatik Rheinau 1, 56075 Koblenz, Germany, 2013. – 18 с.
19. Офіційна документація фреймворку OWL API - Режим доступу : http://owlapi.sourceforge.net/documentation.html. – Дата доступу: 20.05.2016
20. Marbaise B. Maven – Available Plugins / Marbaise B. - Margulies Karl-Heinz, 2014. – 351 с.
21. Wooldridge M. Introduction to MultiAgent Systems / Wooldridge M. John Wiley and Sons, 2002. – 214 с.

ДОДАТОК А

ВИХІДНИЙ КОД РОЗРОБЛЕНОЇ ПРОГРАМИ

1. Клас ReasonerWork

```
package com.mycompany.mavenproject1;

import org.semanticweb.owlapi.model.OWLClass;
import org.semanticweb.owlapi.model.OWLIndividual;
import org.semanticweb.owlapi.model.OWLNamedIndividual;
import org.semanticweb.owlapi.model.OWLOntologyCreationException;
import org.semanticweb.owlapi.reasoner.InferenceType;
import org.semanticweb.owlapi.reasoner.Node;
import org.semanticweb.owlapi.reasoner.NodeSet;
import org.semanticweb.owlapi.reasoner.OWLReasoner;
import uk.ac.manchester.cs.owl.owlapi.OWLDataFactoryImpl;

public class ReasonerWork {
   OWLReasoner reasoner;

    public long getConsistencyCheckingTime() {
      long s = System.nanoTime();
            boolean consistent = reasoner.isConsistent();
            long e = System.nanoTime();
      long timeUsed = e - s;
      return timeUsed;
   }

   public long getInferencesPrecomputingTime(InferenceType inferenceType){
       long s = System.nanoTime();
       reasoner.precomputeInferences();
       long e = System.nanoTime();
       long timeUsed = e - s;
       return timeUsed;
    }

     public long getSubclassesGettingTime(){
       long s = System.nanoTime();
       for(OWLClass currentClass: reasoner.getRootOntology().getClassesInSignature()){
          NodeSet<OWLClass> subClasses = reasoner.getSubClasses(currentClass, false);
       }
```

```
  long e = System.nanoTime();
  long timeUsed = e - s;
  return timeUsed;
}

 public int  getSubclassesCount(){
  int subClassesCount = 0;
  for(OWLClass currentClass: reasoner.getRootOntology().getClassesInSignature()){
    NodeSet<OWLClass> subClasses = reasoner.getSubClasses(currentClass, false);
    for (OWLClass subClass : subClasses.getFlattened()) {
      subClassesCount++;
    }
  }
  return subClassesCount;
 }

 public int  getClassesCount(){
  int classesCount = 0;
  for(OWLClass currentClass: reasoner.getRootOntology().getClassesInSignature()){
     classesCount++;
  }
  return classesCount;
 }
  public int  getIndividualsCount(){
  int individualsCount = 0;
  for(OWLClass currentClass: reasoner.getRootOntology().getClassesInSignature()){
     NodeSet<OWLNamedIndividual> individuals = reasoner.getInstances(currentClass, false);
    for (OWLNamedIndividual ind : individuals.getFlattened()) {
      individualsCount++;
    }
  }
  return individualsCount;
 }

 public long  getIndividualsGettingTime(){
  long s = System.nanoTime();
  for(OWLClass currentClass: reasoner.getRootOntology().getClassesInSignature()){
     NodeSet<OWLNamedIndividual> individuals = reasoner.getInstances(currentClass, false);
  }
  long e = System.nanoTime();
  long timeUsed = e - s;
  return timeUsed;
 }
```

```
    public long getUnsatisfiableClassesGettingTime() {
     long s = System.nanoTime();
        Node<OWLClass> unsatisfiable = reasoner.getUnsatisfiableClasses();
           long e = System.nanoTime();
     long timeUsed = e - s;
     return timeUsed;
   }

    public long getDisjointClassesTime(){
      long s = System.nanoTime();
      for(OWLClass currentClass: reasoner.getRootOntology().getClassesInSignature()){
        NodeSet<OWLClass> disjointClasses = reasoner.getDisjointClasses(currentClass);
      }
      long e = System.nanoTime();
      long timeUsed = e - s;
      return timeUsed;
    }

    public int getDisjointClassesCount(){
      int disjointClassesCount = 0;
      for(OWLClass currentClass: reasoner.getRootOntology().getClassesInSignature()){
        NodeSet<OWLClass> disjointClasses = reasoner.getDisjointClasses(currentClass);
        for (OWLClass disJ : disjointClasses.getFlattened()) {
          disjointClassesCount++;
        }
      }
      return disjointClassesCount;
    }
}
```

2. Клас StructuralReasonerWork

```
package com.mycompany.mavenproject1;

import org.mindswap.pellet.utils.progress.ConsoleProgressMonitor;
import org.semanticweb.owlapi.apibinding.OWLManager;
import org.semanticweb.owlapi.model.IRI;
import org.semanticweb.owlapi.model.OWLClass;
import org.semanticweb.owlapi.model.OWLOntology;
import org.semanticweb.owlapi.model.OWLOntologyCreationException;
import org.semanticweb.owlapi.model.OWLOntologyManager;
import org.semanticweb.owlapi.reasoner.InferenceType;
import org.semanticweb.owlapi.reasoner.NodeSet;
```

```
import org.semanticweb.owlapi.reasoner.OWLReasoner;
import org.semanticweb.owlapi.reasoner.OWLReasonerConfiguration;
import org.semanticweb.owlapi.reasoner.OWLReasonerFactory;
import org.semanticweb.owlapi.reasoner.ReasonerProgressMonitor;
import org.semanticweb.owlapi.reasoner.SimpleConfiguration;
import org.semanticweb.owlapi.reasoner.structural.StructuralReasonerFactory;

public class StructuralReasonerWork extends ReasonerWork{
   OWLReasonerFactory reasonerFactory;

   StructuralReasonerWork(OWLOntology ontology) throws OWLOntologyCreationException{
      reasonerFactory = new StructuralReasonerFactory();
      reasoner = reasonerFactory.createReasoner(ontology);
      System.out.println("struct norm");
   }

   @Override
   public String toString(){
      return "Structural reasoner ";
   }
}
```

3. Клас PelletReasonerWork

```
package com.mycompany.mavenproject1;

import com.clarkparsia.pellet.owlapiv3.PelletReasoner;
import com.clarkparsia.pellet.owlapiv3.PelletReasonerFactory;
import org.semanticweb.owlapi.apibinding.OWLManager;
import org.semanticweb.owlapi.model.IRI;
import org.semanticweb.owlapi.model.OWLClass;
import org.semanticweb.owlapi.model.OWLOntology;
import org.semanticweb.owlapi.model.OWLOntologyCreationException;
import org.semanticweb.owlapi.model.OWLOntologyManager;
import org.semanticweb.owlapi.reasoner.InferenceType;
import org.semanticweb.owlapi.reasoner.NodeSet;
import org.semanticweb.owlapi.reasoner.OWLReasoner;


public class PelletReasonerWork extends ReasonerWork{

   PelletReasonerWork(OWLOntology ontology) throws OWLOntologyCreationException{
      reasoner = PelletReasonerFactory.getInstance().createReasoner(ontology );
      System.out.println("pellet norm");
```

```
    }

    @Override
    public String toString(){
        return "Pellet reasoner ";
    }
}
```

4. Клас HermitReasonerWork

```
package com.mycompany.mavenproject1;

import java.util.Collections;
import java.util.Set;
import org.semanticweb.HermiT.Reasoner;
import org.semanticweb.owlapi.apibinding.OWLManager;
import org.semanticweb.owlapi.model.IRI;
import org.semanticweb.owlapi.model.OWLClass;
import org.semanticweb.owlapi.model.OWLClassExpression;
import org.semanticweb.owlapi.model.OWLOntology;
import org.semanticweb.owlapi.model.OWLOntologyCreationException;
import org.semanticweb.owlapi.model.OWLOntologyManager;
import org.coode.owlapi.manchesterowlsyntax.ManchesterOWLSyntaxEditorParser;
import org.semanticweb.owlapi.reasoner.InferenceType;
import org.semanticweb.owlapi.reasoner.NodeSet;
import org.semanticweb.owlapi.reasoner.OWLReasoner;
import org.semanticweb.owlapi.util.ShortFormProvider;
import org.semanticweb.owlapi.util.SimpleShortFormProvider;

public class HermitReasonerWork extends ReasonerWork{

    HermitReasonerWork(OWLOntology ontology) throws OWLOntologyCreationException{
        reasoner=new Reasoner(ontology);
        System.out.println("hermit norm");
    }

    @Override
    public String toString(){
        return "Hermit reasoner ";
    }
}
```

5. Клас ReasonersCollectionToAnalyse

```
package com.mycompany.mavenproject1;
```

```
import java.util.logging.Level;
import java.util.logging.Logger;
import org.semanticweb.owlapi.apibinding.OWLManager;
import org.semanticweb.owlapi.model.IRI;
import org.semanticweb.owlapi.model.OWLOntology;
import org.semanticweb.owlapi.model.OWLOntologyCreationException;
import org.semanticweb.owlapi.model.OWLOntologyManager;

public class ReasonersCollectionToAnalyse {
  OWLOntologyManager manager;
  OWLOntology  ontology;
  ReasonerWork structuralReasoner;
  ReasonerWork hermitReasoner ;
  ReasonerWork pelletReasoner ;

  public ReasonersCollectionToAnalyse() {
     manager = OWLManager.createOWLOntologyManager();
  }
  public void initReasonersCollectionToAnalyse(String ontologyIRI){
       try {
          ontology = manager.loadOntologyFromOntologyDocument(IRI.create(ontologyIRI));
          structuralReasoner = new StructuralReasonerWork(ontology);
          hermitReasoner = new HermitReasonerWork(ontology);
          pelletReasoner = new PelletReasonerWork(ontology);
     } catch (OWLOntologyCreationException ex) {
         Logger.getLogger(ReasonersCollectionToAnalyse.class.getName()).log(Level.SEVERE, null, ex);
     }
  }

}
```

6. Клас View

```
package com.mycompany.mavenproject1;
import org.semanticweb.owlapi.reasoner.InferenceType;

public class View extends javax.swing.JFrame {
   ReasonersCollectionToAnalyse reasoners = new ReasonersCollectionToAnalyse();
   public View() {
      initComponents();
   }

   /**
```

```
 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {

    ontologyPathField = new javax.swing.JTextField();
    jLabel1 = new javax.swing.JLabel();
    jButton1 = new javax.swing.JButton();
    jScrollPane1 = new javax.swing.JScrollPane();
    jTable1 = new javax.swing.JTable();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
    setTitle("Reasoners analyser");
    setBackground(new java.awt.Color(255, 204, 0));

    ontologyPathField.setText("http://protege.cim3.net/file/pub/ontologies/travel/travel.owl");

    jLabel1.setText("Ontology IRI");

    jButton1.setText("Analyse");
    jButton1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jButton1ActionPerformed(evt);
        }
    });

    jTable1.setFont(jTable1.getFont().deriveFont(jTable1.getFont().getSize()+1f));
    jTable1.setModel(new javax.swing.table.DefaultTableModel(
        new Object [][] {
            {"Inferences precomputing time (Class hierarchy)", null, null, null},
            {"Inferences precomputing time (Data property hierarchy)", null, null, null},
            {"Inferences precomputing time (Object property hierarchy)", null, null, null},
            {"Classes count", null, null, null},
            {"Subclasses getting time", null, null, null},
            {"Subclasses count", null, null, null},
            {"Disjoint classes getting time", null, null, null},
            {"Disjoint classes count", null, null, null},
            {"Individuals getting time", null, null, null},
            {"Individuals count", null, null, null},
            {"Consistency checking time", null, null, null}
```

```
            },
            new String [] {
                "Reasoner name", "Structural", "Pellet", "Hermit"
            }
        ));
        jScrollPane1.setViewportView(jTable1);

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addGap(24, 24, 24)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addComponent(jScrollPane1, javax.swing.GroupLayout.Alignment.TRAILING,
javax.swing.GroupLayout.DEFAULT_SIZE, 664, Short.MAX_VALUE)
                    .addGroup(layout.createSequentialGroup()
                        .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                            .addComponent(jLabel1, javax.swing.GroupLayout.PREFERRED_SIZE, 92,
javax.swing.GroupLayout.PREFERRED_SIZE)
                            .addComponent(jButton1, javax.swing.GroupLayout.PREFERRED_SIZE, 92,
javax.swing.GroupLayout.PREFERRED_SIZE))
                        .addGap(18, 18, 18)
                        .addComponent(ontologyPathField)))
                .addContainerGap())
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addGap(27, 27, 27)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                    .addComponent(jLabel1, javax.swing.GroupLayout.PREFERRED_SIZE, 32,
javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addComponent(ontologyPathField))
                .addGap(18, 18, 18)
                .addComponent(jButton1)
                .addGap(34, 34, 34)
                .addComponent(jScrollPane1, javax.swing.GroupLayout.PREFERRED_SIZE, 208,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addContainerGap())
        );

        pack();
```

```
    }// </editor-fold>

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        reasoners.initReasonersCollectionToAnalyse(ontologyPathField.getText());

jTable1.setValueAt(reasoners.structuralReasoner.getInferencesPrecomputingTime(InferenceType.CLASS_HIERARCHY)+ "
ns", 0, 1);
        jTable1.setValueAt(reasoners.pelletReasoner.getInferencesPrecomputingTime(InferenceType.CLASS_HIERARCHY)+ "
ns", 0, 2);
        jTable1.setValueAt(reasoners.hermitReasoner.getInferencesPrecomputingTime(InferenceType.CLASS_HIERARCHY)+ "
ns", 0, 3);


jTable1.setValueAt(reasoners.structuralReasoner.getInferencesPrecomputingTime(InferenceType.DATA_PROPERTY_HIERA
RCHY)+ " ns", 1, 1);

jTable1.setValueAt(reasoners.pelletReasoner.getInferencesPrecomputingTime(InferenceType.DATA_PROPERTY_HIERARC
HY)+ " ns", 1, 2);

jTable1.setValueAt(reasoners.hermitReasoner.getInferencesPrecomputingTime(InferenceType.DATA_PROPERTY_HIERAR
CHY)+ " ns", 1, 3);


jTable1.setValueAt(reasoners.structuralReasoner.getInferencesPrecomputingTime(InferenceType.OBJECT_PROPERTY_HIE
RARCHY)+ " ns", 2, 1);

jTable1.setValueAt(reasoners.pelletReasoner.getInferencesPrecomputingTime(InferenceType.OBJECT_PROPERTY_HIERAR
CHY)+ " ns", 2, 2);

jTable1.setValueAt(reasoners.hermitReasoner.getInferencesPrecomputingTime(InferenceType.OBJECT_PROPERTY_HIERA
RCHY)+ " ns", 2, 3);

        jTable1.setValueAt(reasoners.structuralReasoner.getClassesCount(), 3, 1);
        jTable1.setValueAt(reasoners.pelletReasoner.getClassesCount(), 3, 2);
        jTable1.setValueAt(reasoners.hermitReasoner.getClassesCount(), 3, 3);

        jTable1.setValueAt(reasoners.structuralReasoner.getSubclassesGettingTime() + " ns", 4, 1);
        jTable1.setValueAt(reasoners.pelletReasoner.getSubclassesGettingTime()+ " ns", 4, 2);
        jTable1.setValueAt(reasoners.hermitReasoner.getSubclassesGettingTime()+ " ns", 4, 3);

        jTable1.setValueAt(reasoners.structuralReasoner.getSubclassesCount(), 5, 1);
        jTable1.setValueAt(reasoners.pelletReasoner.getSubclassesCount(), 5, 2);
        jTable1.setValueAt(reasoners.hermitReasoner.getSubclassesCount(), 5, 3);
```

```
    jTable1.setValueAt(reasoners.structuralReasoner.getDisjointClassesTime()+ " ns", 6, 1);
    jTable1.setValueAt(reasoners.pelletReasoner.getDisjointClassesTime()+ " ns", 6, 2);
    jTable1.setValueAt(reasoners.hermitReasoner.getDisjointClassesTime()+ " ns", 6, 3);

    jTable1.setValueAt(reasoners.structuralReasoner.getDisjointClassesCount(), 7, 1);
    jTable1.setValueAt(reasoners.pelletReasoner.getDisjointClassesCount(), 7, 2);
    jTable1.setValueAt(reasoners.hermitReasoner.getDisjointClassesCount(), 7, 3);

    jTable1.setValueAt(reasoners.structuralReasoner.getIndividualsGettingTime()+ " ns", 8, 1);
    jTable1.setValueAt(reasoners.pelletReasoner.getIndividualsGettingTime()+ " ns", 8, 2);
    jTable1.setValueAt(reasoners.hermitReasoner.getIndividualsGettingTime()+ " ns", 8, 3);

    jTable1.setValueAt(reasoners.structuralReasoner.getIndividualsCount() , 9, 1);
    jTable1.setValueAt(reasoners.pelletReasoner.getIndividualsCount(), 9, 2);
    jTable1.setValueAt(reasoners.hermitReasoner.getIndividualsCount(), 9, 3);

    jTable1.setValueAt(reasoners.structuralReasoner.getConsistencyCheckingTime()+ " ns", 10, 1);
    jTable1.setValueAt(reasoners.pelletReasoner.getConsistencyCheckingTime()+ " ns", 10, 2);
    jTable1.setValueAt(reasoners.hermitReasoner.getConsistencyCheckingTime()+ " ns", 10, 3);
    System.out.println("loaded");
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    /* Set the Nimbus look and feel */
    //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
    /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
     * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
     */
    try {
        for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.UIManager.getInstalledLookAndFeels()) {
            if ("Nimbus".equals(info.getName())) {
                javax.swing.UIManager.setLookAndFeel(info.getClassName());
                break;
            }
        }
    } catch (ClassNotFoundException ex) {
        java.util.logging.Logger.getLogger(View.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    } catch (InstantiationException ex) {
        java.util.logging.Logger.getLogger(View.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
```

```
        } catch (IllegalAccessException ex) {
            java.util.logging.Logger.getLogger(View.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
        } catch (javax.swing.UnsupportedLookAndFeelException ex) {
            java.util.logging.Logger.getLogger(View.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
        }
        //</editor-fold>

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new View().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JButton jButtonAnalyse;
    private javax.swing.JLabel jLabelOntologyIRI;
    private javax.swing.JScrollPane jScrollPaneTable;
    private javax.swing.JTable jTableKyiгдei;
    private javax.swing.JTextFieldOntologyIRI;
    // End of variables declaration
}
```

7. Файл pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>mavenproject1</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <repositories>
        <repository>
            <id>unknown-jars-temp-repo</id>
            <name>A temporary repository created by NetBeans for libraries and jars it could not identify. Please replace the
dependencies in this repository with correct ones and delete this repository.</name>
            <url>file:${project.basedir}/lib</url>
        </repository>
    </repositories>
    <dependencies>
        <dependency>
```

```
    <groupId>net.sourceforge.owlapi</groupId>
    <artifactId>owlapi-distribution</artifactId>
    <version>3.4.5</version>
</dependency>

<dependency>
    <groupId>com.github.ansell.pellet</groupId>
    <artifactId>pellet-jena</artifactId>
    <version>2.3.3</version>
</dependency>
<dependency>
    <groupId>com.hermit-reasoner</groupId>
    <artifactId>org.semanticweb.hermit</artifactId>
    <version>1.3.8.4</version>
</dependency>
<dependency>
    <groupId>com.hp.hpl.jena</groupId>
    <artifactId>arq</artifactId>
    <version>2.8.3</version>
</dependency>
<dependency>
    <groupId>com.github.ansell.pellet</groupId>
    <artifactId>pellet-common</artifactId>
    <version>2.3.3</version>
</dependency>

<dependency>
    <groupId>org.aksw.jena-sparql-api</groupId>
    <artifactId>jena-sparql-api-core</artifactId>
    <version>2.13.0-5</version>
</dependency>

<dependency>
    <groupId>org.aksw.jena-sparql-api</groupId>
    <artifactId>jena-sparql-api-server</artifactId>
    <version>2.13.0-5</version>
</dependency>
  <dependency>
      <groupId>com.github.ansell.pellet</groupId>
      <artifactId>pellet-query</artifactId>
      <version>2.3.6-ansell</version>
      <type>jar</type>
  </dependency>
```

```
        <dependency>
            <groupId>com.github.ansell.pellet</groupId>
            <artifactId>pellet-owlapiv3</artifactId>
            <version>2.3.2</version>
        </dependency>
        <dependency>
            <groupId>net.sourceforge.owlapi</groupId>
            <artifactId>owlapi-parsers</artifactId>
            <version>3.4</version>
        </dependency>
      <dependency>
        <groupId>unknown.binary</groupId>
        <artifactId>AbsoluteLayout</artifactId>
        <version>SNAPSHOT</version>
      </dependency>
    </dependencies>
    <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>1.7</maven.compiler.source>
      <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
</project>
```