Polsko-Japońska Wyższa Szkoła Technik Komputerowych

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Marcin Stępień
Nr albumu: s3185

Filtr RDF dla systemu Odra

Praca magisterska
Napisana pod kierunkiem
prof. Kazimierza Subiety

Warszawa, Lipiec 2009

Streszczenie pracy

Praca *Filtr RDF dla systemu Odra* dotyczy implementacji importera danych RDF do składu systemu Odra. Filtr pozwala programiście języka SBQL na przetwarzanie danych RDF, w szczególności wykorzystanie mechanizmów charakterystycznych dla podejścia SBA w przetwarzaniu danych o strukturze grafu skierowanego jakim jest graf RDF. Autor opisuje charakterystykę filozofii Semantic Web skupiając się na jej podstawowym językiem opisu zasobów - Resource Description Framework (RDF).

Zaproponowana koncepcja importu przechowywania danych RDF przewiduje wykorzystanie predefiniowanej biblioteki języka SBQL rezygnując tym samym z próby reprezentacji danych RDF bezpośrednio jako obiekty systemu Odra w sposób zbliżony do filtru XML. Decyzje projektową autor tłumaczy charakterystyką formatu RDF oraz samą koncepcją Semantic Web zakładającą otwarty, dynamiczny charakter metadanych zawierających informacje o semantyce rozpatrywanych zasobów.

Autor opisuje w pracy główne powody przyjętego rozwiązania, identyfikuje słabe strony oraz przytacza zauważone błędy w implementacji samego systemu Odra mające wpływ na przyszłą rozbudowę i formułowanie zapytań do grafu RDF.

Praca zawiera propozycje rozszerzenia biblioteki RDF.sbql o kolejne warstwy aplikacyjne ukrywające struktury charakterystyczne dla samej gramatyki RDF oraz dostarczające narzędzi dla efektywnego przetwarzania danych semantycznych.

Słowa kluczowe: Odra, SBQL, Integracja, Semantic Web, RDF, Software Development, IDE, Jena

Spis Treści

1 Kontekst i cel pracy

Niniejsza praca jest próbą przygotowania środowiska programistycznego dla przetwarzania danych semantycznych w systemie Odra. Idea *Semantic Web*, choć powstała niedługo po opracowaniu samego WWW, wciąż znajduje się na etapie akademickich badań i pierwszych testów prototypów. O ile opis danych metadaymi semantycznymi znajduje już zastosowanie w projektach badawczych (np. biologii) to w dziedzinach informatyki bliskiej zwykłym użytkownikom wciąż jest nieobecna dla zwykłego użytkownika. Dzieje się tak między innymi z powodu braku efektywnych narzędzi developerskich jak również wciąż niezrealizowanego tzw. *killer app* uwidaczniającego zalety przetwarzania danych semantycznych.

Praca ma na celu rozszerzenie funkcjonalności systemu Odra o filtr danych RDF. Możliwość importowania danych zapisanych w formacie RDF pozwala na przetwarzanie i odpytywanie metadanych zasobów internetowych oraz dokumentów repozytorium (np. projektów takich jak *e-Gov Bus*). W pracy została wykorzystana stale rozwijana przez pracowników HP Labs biblioteka open source: Jena.

Rezultatem pracy jest rozszerzenie systemu Odra o filtr RDF pozwalający na import danych RDF. Filtr wraz z dedykowanymi klasami SBQL dla systemu Odra otwiera drogę badań nad możliwością zastosowań SBQL w przetwarzaniu danych RDF; szczególnie interesującym kierunkiem jest porównanie SBQL z językiem zapytań do RDF tj. SPARQL oraz z implementacjami RDF w systemach SQL. Filtr RDF rozszerza zakres zastosowań systemu Odra o zagadnienia związane z Semantic Web stwarzając obiecujące perspektywy badawcze oraz praktyczne użycie na polu integracji danych.

2 Stan sztuki - Semantic Web i obiektowe bazy danych

2.1 Semantic Web

2.1.1 Próba przetwarzania semantyki danych przez automaty

Twórcy pojęcia Sieć Semantyczna (ang. *Semantic Web*) za cel postawili zdefiniowanie standardów pozwalających na przetwarzanie zbiorów danych zarządzanych przez dowolne systemy, również wzajemnie heterogeniczne *[14]* oraz przez automaty z uwzględnieniem ich semantycznej treści.

Dzisiejszy stan zarządzania treścią, zbiorami danych pochodzących z różnych źródeł, w szczególności sieci Internet, cały czas pozostawia zadania kojarzenia i końcowej fazy wyszukiwania danych w gestii człowieka. Technologie, standardy i metodologie Semantic Web skupiają się na umożliwieniu przetwarzania semantycznego danych dostępnych w rozległych systemach informacyjnych pracujących w środowisku heterogenicznym, a w szczególności na World Wide Web opisując dane przy użyciu zestandaryzowanych ontologii. Scalając strony www za pomocą opisu treści (metadanych) możliwym staje się wnioskowanie o tej treści. W szerszym zastosowaniu można spodziewać się wysokiej przydatności tej technologii jako wspólnego mianownika w komunikacji danych pomiędzy systemami/urządzeniami komponując się w trend rosnącej liczby agentów i urządzeń przenośnych przypadających na użytkownika *[16]*.

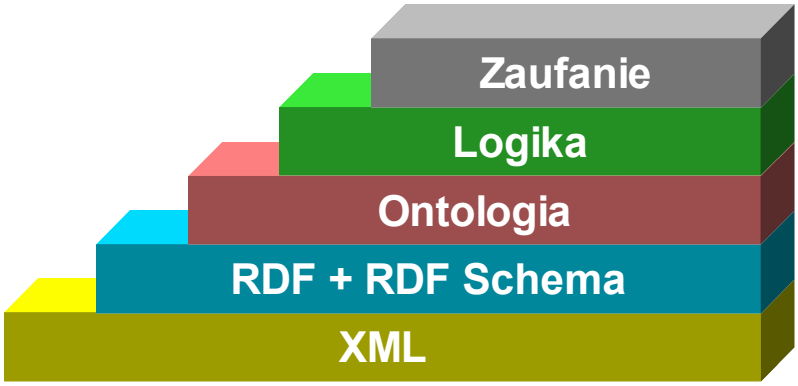
Technologie Semantic Web mogą zostać użyte na wiele sposobów przekształcając funkcjonalność sieci web poprzez dodanie metadanych do treści rozszerzając możliwości wyszukiwania i zarządzania danymi. Wyszukiwanie zasobów może być przeprowadzane przez automaty które mogą opierać kryteria wyszukujące np. na podstawie kontekstu procesu w jakim działa rozpatrywany agent.

Sieci Semantyczne to również wsparcie dla rozwiązań integracji danych. Przykładem potencjalnego zastosowania jest nadanie opisu spełniającego przyjętą ontologię dla Web Service w celu rozszerzenia możliwości wyszukania i złożenia usługi. Definiowanie wspólnych ontologi to także szansa dla budowy zestandaryzowany osłon (*wrappers*) dla systemów informacyjnych w celu uproszczenia ich integracji. Wreszcie nadanie wspólnych środków opisu semantyki to *lingua franca* w komunikacji pomiędzy agentami działającymi na oprogramowaniu jak również na poziomie połączeń pomiędzy heterogenicznymi systemami operującymi na oddzielnych urządzeniach sprzętowych.

2.1.2 Warstwy architektury Semantic Web

Semantic Web jako rozszerzenie WWW jest infrastrukturą służącą do opisywania danych metadanymi pozwalającymi na przetwarzanie w kontekście semantyki. Głównym zadaniem jest możliwość opisu i odczytania *znaczenia* zasobów internetowych. Architekturę Semantic Web realizującą te cele można

podzielić na 5 warstw *[6]*:



- **warstwa XML** - reprezentuje syntaktyczny zapis danych. Format XML został wybrany jako standard serializacji danych RDF, jest ugruntowaną technologią hierarchicznego zapisu danych obecną w WWW
- **warstwa RDF** - reprezentuje *znaczenie* danych, ich semantyczną reprezentacje
- **warstwa Ontologi** - reprezentuje formalny, powszechnie uzgodniony słownik wyrażania znaczenia danych
- **warstwa Logiki** - realizuje zadania wnioskowania na danych semantycznych, danych posiadających *znaczenie;* może obejmować elementu logiki predykatów jak również arbitralnie przyjęte reguły wnioskowania
- **warstwa Zaufania** - mechanizmy pozwalające na zestandaryzowanie procesu autoryzacji użytkowników, identyfikacji ich zasobów oraz określenia praw udostępniania zasobów

2.1.3 Podstawowe elementy infrastruktury Semantic Web

- Niepowtarzalne identyfikatory zasobów – URI. Zasoby identyfikowane są za pomocą URI *[10]*. Takie rozwiązanie składa odpowiedzialność za nadawanie identyfikatorów dla zasobów na barki:
  - ➔ Jednostek administracyjnych nadających nazwy domen ogólnoświatowych i krajowych
  - ➔ Lokalnych systemów zarządzających treścią
  - ➔ Przyjętej systematyki nadawaniu identyfikacji (np. zgodnej z lokalnym prawem - Pesel, NIP, ISBN, itd.)
- Standardowa składnia dla opisu metadanych - RDF
- Standardowe środki opisu właściwości metadanych - RDFS
- Standardowe środki opisu relacji pomiędzy elementami metadanych (ontologie zdefiniowane za pomocą OWL Web Ontology Language)

- Powszechnie przyjęte gramatyki (ontologie)

2.1.4 Semantic Web jako przestrzeń integracji

2.1.4.1 Istniejące rozwiązania

Od momentu powstania standardu RDF przez kolejne 10 lat swojego istnienia środowiska związane z koncepcją Semantic Web nie doczekały się tzw. *killer application* mogącej uwypuklić możliwości i zalety technologii RDF w praktyce. Standard RDF obecny jest jednak szeroko rozpowszechniony w gramatykach znanych jako: RSS oraz vCard.

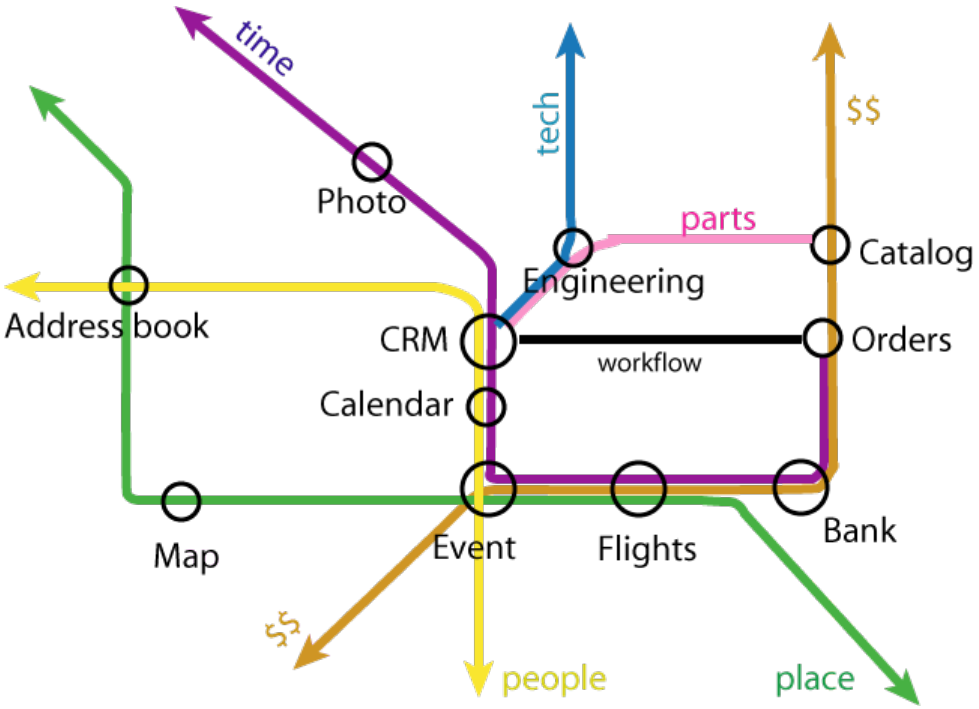
RSS jako standard przesyłania nagłówków wiadomości został stworzony jako gramatyka RDF/XML*[12]* przez firmę Netscape. Przez pewien czas rozwoju gramtyki RSS porzucono wykorzystanie technologii RDF, lecz powrócono do niej z powodu korzyści jakie niesie jej zastosowanie m in.: wykorzystanie dużego stopnia elastyczności i możliwości rozszerzania gramatyk budowanych w technologii RDF włącznie z wykorzystaniem przestrzeni nazw dzięki którym uniknięto kolizji nazw. Twórcy standardu RSS odnaleźli w technologii RDF odpowiedź na problem rozszerzania RSS bez potrzeby cyklicznego wydawania nowych wersji specyfikacji, dodatkowo zastosowanie RDF otwiera możliwość łączenia danych RSS z innymi gramatykami RDF. Gramatyka stała się rzeczywistym standardem i ostatecznie w wersji RSS 1.0 bazuje na formacie RDF/XML.

vCard to kolejny przykład zastosowania uzgodnionej gramatyki przynoszącej dla użytkowników korzyść wymiany wizytówek kontaktów w dowolnej aplikacji obsługującej tę gramatykę. Jakkolwiek oryginalnym formatem zapisu vCard jest XML, W3C włączyło vCard do podstawowaego zestawu gramatyk RDF.

2.1.4.2 Możliwe zastosowania w koncepcji EAI oraz Application Service Bus

Tim Berners-Lee, autor i główny architekt koncepcji oparcia Semnatic Web na technologii RDF wskazuje na specyfikę rozwiązań bazujących na połączeniu dotychczasowego podejścia do rozwiązania problemu integracji heterogenicznych systemów informatycznych, traktowanych zarówno jako usługi w koncepcji Architektury Zorientowanej na Usługi (SOA) jak również *hub* integracyjny w klasycznym podejściu do integracji systemów (EAI) z koncepcją Semantic Web. Na corocznych meetingach American Society for Information Science and Technology Tim Berners-Lee w seminarium *Managing and Enhancing Information: Cultures and Conflicts* zaprezentował podsumowanie wizji rozwoju technologii RDF.

Jego częścią jest zobrazowanie pomysłu integracji aplikacji na podstawie dziedziny pojęciowej (*Applications connected by concepts*). Otwarty charakter danych zapisanych w grafach RDF pozwala łączyć i wnioskować o powiązaniach pomiędzy zbiorami danych pochodzących z różnych klas pojęciowych i aplikacji wykorzystując mechanizmy mapowania danych o zasobach na podstawie danych o ontologii reguł RDFS oraz bezpośredniej identyfikacji tożsamych zasobów na podstawie URI zasobów RDF.

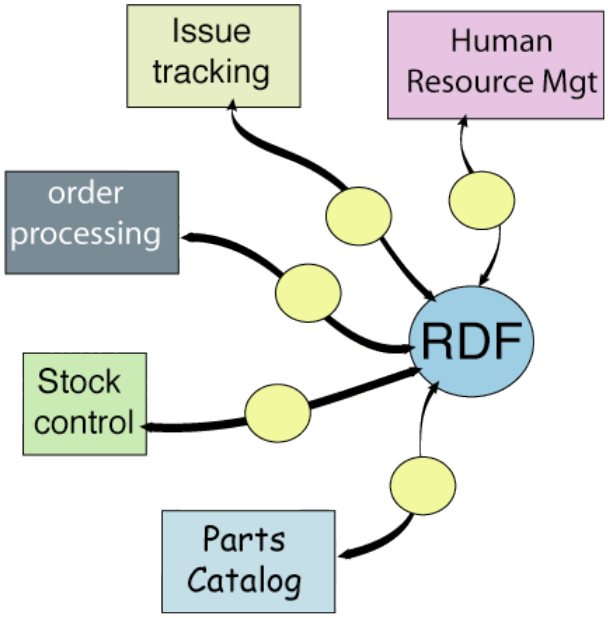


Szkic punktów integracji na podstawie dziedziny pojęciowej [17]

Za prosty przykład obiektu pojęciowego, mogącego brać udział w integracji aplikacji, obejmujący jedynie zakres integracji na poziomie danych, można wziąć dane zapisane w formacie vCard. Standardowa, uzgodniona gramatyka, dzięki której format danych wizytówkowych czytelny jest dla wielu aplikacji można uznać za „punkt styku" aplikacji. Obiekty vCard mogą zostać wykorzystane jako pośrednik dla systemów przetwarzających dane adresowe. Idąc dalej, zasoby RDF zawarte w obiekcie vCard mogą podlegać dalszemu przetworzeniu w kontekście integracji z grafami RDF reprezentującymi inne obiekty pojęciowe związane z procesem lub aplikacją.*[2]*

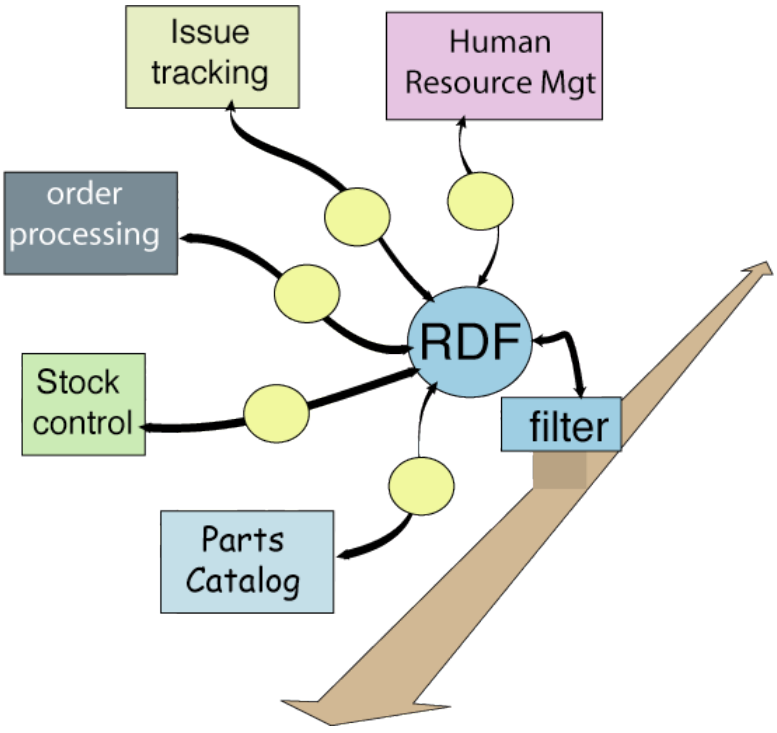
Integracja systemów bazująca na mapowaniu semantycznym rodzi nowe wyzwania i możliwości: stanowi most pomiędzy dziedziną pojęciową aplikacji definiowaną przez twórców oprogramowania a narzędziami mapowania schematu danych integrowanych aplikacji. Wprowadzenie wspólnej gramatyki dla opisu danych wraz z mechanizmami integracji gramatyk daje możliwość wykonania generycznych operacji integrujących dane biznesowe, jak również pozwala opisane semantycznie obiekty biznesowe używać w procesie biznesowym (lub aplikacji), bazując na kontekście procesu i właściwościach obiektu w oderwaniu od samej aplikacji.

Kolejnym przykładem zastosowania formatu RDF jest jego użycie jako zestandaryzowanej gramatyki zapisu danych dla architektury *Enterprise Application Integration.* Technologia RDF może zostać użyta jako standardowy format zapisu danych w scentralizowanym miejscu styku systemów.



RDF jako HUB integracyjny [17]

Rozszerzeniem wizji użycia RDF w architekturze Enterprise Application Integration jest koncepcja nazwana przez Tim Bernsers-Lee *Global Integration Bus* mająca związek z koncepcją *Enterprise Service Bus* i *Architekturą Zorientowaną na Usługi*. Wprowadzenie adaptera *RDF - szyna aplikacyjna* łączy cechy koncepcji RDF z koncepcją SOA, np. otwiera drogę do korzystania z dobrodziejstw globalnego zakresu nazewnictwa i identyfikacji obiektów na podstawie URI.



RDF w roli adaptera Global Integration Bus [17]

2.2 SBA

Podejście stosowe, Stack-Based Architecture lub Stack-Based Approach - SBA, to teoria obiektowych języków zapytań w myśl której języki zapytań są językami programowania. Takie podejście implikuje stosowanie koncepcji charakterystycznych dla języków programowania w językach zapytań.

„Podejście stosowe umożliwiło stworzenie spójnej teorii niezależnej od specyficznego modelu danych. Można ją stosować dla relacyjnych, hierarchicznych, obiektowych i obiektowo-relacyjnych baz danych oraz dla repozytoriów XML. Dla dowolnego z tych modeli podejście stosowe umożliwia stworzenie i szybkie zaimplementowanie bardzo mocnego języka zapytań dla praktycznie dowolnie wybranego celu. Dzięki temu podejście to ucina dość jałowe dyskusje odnośnie do tego, który model danych sprzyja lub nie sprzyja realizacji języka zapytań i który ma „zdrowe podstawy matematyczne", a który ich „nie ma". Podejście stosowe do pewnego stopnia unifikuje i klasyfikuje modele danych, redukując w ten sposób niesławny syndrom „jeszcze jednego modelu obiektowego". Syndrom ten polega na uporczywym proponowaniu kolejnych obiektowych modeli danych, których celem jest zredukowanie drobnych wad i braków poprzednich modeli. W istocie ten ciąg poprawek i mutacji nie ma końca...

...Podejście stosowe jest konkurencją dla teorii znanych z modelu relacyjnego, takich jak algebra relacyjna, rachunek relacyjny i logika matematyczna. Popularne podejścia do obiektowych języków zapytań są oparte na rozszerzeniach i modyfikacjach tych teorii w postaci tzw. „algebr obiektowych" oraz specjalnych logik i rachunków. Podejście stosowe eksponuje braki i wady tych teorii, pokazując ich ograniczenia i nieadekwatność do problemu."[1]

Dla twórców oprogramowania narzędzia bazujące na podejściu stosowym przynoszą nowe rozwiązania na polu języków programowania łączących możliwości języka zapytań o mocy algorytmicznej przewyższającej możliwości standardu SQL oraz proponują skuteczne narzędzie w budowie rozwiązań integracyjnych – aktualizowalne perspektywy.

2.3 SBQL

2.3.1 Charakterystyka

SBQL (Stack Based Query Language) to obiektowy język programowania i zapytań do baz danych oparty na podejściu stosowym (SBA - Stack Based Approach). Autorem tej koncepcji jest prof. Kazimierz Subieta. SBQL charakteryzuje się wysoką mocą algorytmiczną zarówno dla zapytań jak i w przetwarzaniu struktur danych. Język SBQL łączy w sobie cechy języka programowania oraz języka zapytań. Pozwala programiście na korzystanie z dobrodziejstw makroskopowych zapytań i efektywną obsługę kolekcji w blokach kodu zawierających konstrukcje imperatywne charakterystyczne dla języków programowania rozwiązując tym samym problem tzw. *niezgodności impedancji* tj. zanurzenia języka wyższego poziomy (zapytań) w języku

niższego poziomu (programowania) powodującą utratę mocy algorytmicznej, ograniczenie środków programowania, zmniejszoną wydajność, trudniejsze utrzymanie kodu itd.

2.3.2 Wybrane cechy języka SBQL

2.3.2.1 Operator tranzytywnego domknięcia

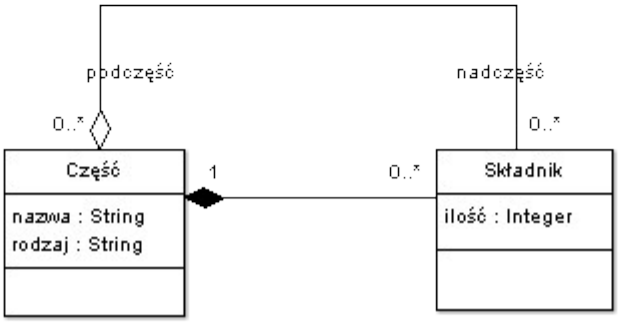
Jednym z bardziej interesujących mechanizmów charakterystycznych dla języka SBQL, a nieobecnych w innych językach zapytań, jest operator tranzytywnego domknięcia ***close by***.

Tranzytywne domknięcie w SBQL jest operatorem niealgebraicznym, który został stworzony dla aplikacji przetwarzających struktury danych o rekurencyjnej strukturze danych albo strukturze grafu dla których typowe operatory języka zapytań takie jak selekcja, projekcja, kwantyfikatory, złączenia są niewystarczające. Operator zwraca wszystkie wszystkie węzły struktury danych do których może nawigować według podanej relacji (np. asocjacji) oraz pozwala wykonywać operacje obliczeniowe w grafie.

Warto zwrócić uwagę na ten operator w kontekście przetwarzania grafów RDF. Autor pracy dostrzega przydatność operatora w przeszukiwaniu danych RDF ze względu na strukturę grafu skierowanego oraz zwraca uwagę na możliwość promocji koncepcji SBA na forum aplikacji przetwarzającej dane RDF dzięki sile algorytmicznej operatora tranzytywnego domknięcia, którą można zestawić z rozwiązaniami systemów relacyjnych baz danych.

Podstawowym przykładem zapytania, używanym w publikacjach prof. Kazimierza Subiety, ilustrującym wykorzystanie tranzytywnego domknięcia jest zapytanie do struktury danych typu *Bill-of-Material* (BOM).

BOM jest opisem hierarchii części danego wyrobu, obrazuje relacje część-podczęść.



Uproszczony schemat BOM

Przykładowe zapytanie: podaj wszystkie podczęści składające się na część o nazwie „silnik”

```
distinct((Część where nazwa = ”silnik”)
  close by (składnik.nadczęść.Część))
```

Podzapytanie *(Część where nazwa = ”silnik”)* zwraca referencję do obiektu *Część* o nazwie „silnik”.

Operator *close by* przeszukuje atrybuty zwróconego obiektu, a następnie rekurencyjnie te atrybuty które reprezentują asocjacje *nadczęść* aż do momentu gdy wszystkie składniki części „silnik” które pośrednio lub bezpośrednio wskazują na niego asocjacją *nadczęść* zostaną zwrócone jako wynik.

3 Narzędzia i metodologie

3.1 RDF

3.1.1 Definicja i charakterystyka

RDF (ang. Resource Description Framework) - jest językiem opisu zasobów kótry dostarcza środków opisu metadanych dokumentów w celu nadania im semantyki. Wg. nazwy jest to zręb programistyczny, w szczególności język zdefiniowany przez jego zastosowanie tj. reprezentacja informacji o zasobach (resource) początkowo w sieci internetowej WWW, a następnie w szerszym zakresie. Informacje w dokumentach RDF są reprezentowane w formie *zdań RDF* przeznaczonych do przetwarzania przez wyszukiwarki, agenty, brokery informacyjne, przeglądarki internetowe oraz ludzi. Pierwsza oficjalna rekomendacja specyfikacji RDF została opublikowana przez World Wide Consorcjum w 1999 roku, w szczególności przez Ora Lassila'a i Ralph Swick'a, na podstawie wcześniejszych prac i analiz prowadzonych przez Tim Berneers Lee.

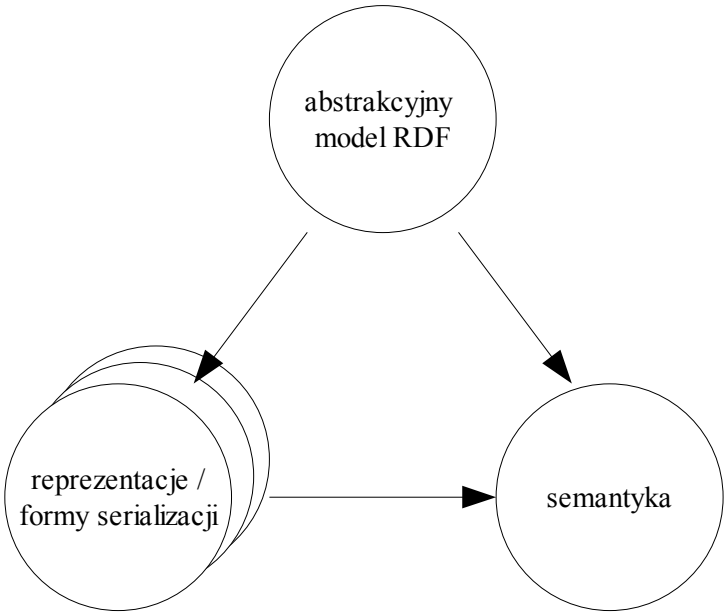
RDF charakteryzuje skoncentrowany na *właściwościach [7][10]*, elastyczny model danych. *Zasoby* mogą przyjąć właściwości oraz typy w dowolnym momencie, bez względu na dotychczasowe właściwości zasobu. Takie podejście pozwala opisywać już istniejące, zmienne zbiory danych metadanymi o nieustalonym schemacie lecz związanymi z ustalonymi słownikami opisującymi semantykę metadanych. Zasób, właściwość i wartość właściwości tworzą *trójkę RDF* (inaczej *zasób*, *predykat*, *obiekt*). W odróżnieniu od podejścia obiektowego, w którym stosuje się hierarchie typów, definiując jakie atrybuty może posiadać obiekt, RDF dopuszcza asocjacje zasobów z dowolnymi właściwościami; *zdanie RDF* może zostać wprowadzone do dokumentu RDF w dowolnym czasie łączą *zasób* z *wartością właściwości*.

Jednym z wyzwań stojącym przed implementacją technologii RDF w infrastrukturę Semantic Web jest rozwiązanie problemu persystencji danych RDF. Rekomendowaną formą serializacji danych RDF jest format RDF/XML który przechowuje dane RDF w dokumentach XML. Serializacja do XML, jakkolwiek pozwalająca korzystać z istniejących technologii przetwarzania samych dokumentów XML, wiąże się z ograniczeniami przetwarzania. Ograniczenia te dotyczą przetwarzania dużych dokumentów XML, ich scalania, odpytywania podzbioru danych RDF, jak również z możliwością dowolnej reprezentacji tych samych danych RDF w dokumentach XML. Implementacje systemów RDF często jako skład danych wykorzystują relacyjne jak i obiektowe bazy danych. W takich systemach występują ograniczenia związane z mapowaniem danych semantycznych charakteryzujących się otwartością oraz swobodą modyfikacji struktury danych. Bazy SQL wymagają stałego zdefiniowanego typu danych kolumn w tabeli zaś bazy obiektowe ograniczają zmiany struktur poprzez typowanie.

3.1.2 RDF jako abstrakcyjny model danych

Jednym z głównych aspektów specyfikacji RDF jest opis formatu jako abstrakcyjny model danych opisujący daną semantykę niezależnie od notacji i metod serializacji. Specyfikacja sugeruje by zdania RDF opisywały *fakty*. Nie jest to jednak formalny składnik specyfikacji.

Reprezentacja abstrakcyjnego modelu RDF może przyjmować różne formy serializacji. Aktualnie najpopularniejsza (posiadającą najwięcej implementacji) jest realizowana za pomocą formy XML tzw. RDFXML



Wielość reprezentacji abstrakcyjnego modelu RDF opisu semantyki

Dokumenty RDF należy rozpatrywać na trzech poziomach abstrakcji:

- **poziom składniowy** - dokument RDF zapisane w danej formie serializacji - np dokument XML
- **poziom strukturalny** - dotyczy danych RDF przedstawionych jako graf skierowany, zbiór trójek RDF
- **poziom semantyczny** - obejmuje semantykę opartą na uzgodnionych wartościach predykatów

RDF identyfikuje dane (**Zasoby**, **Właściwości**) za pomocą **URI**. Opis zasobów realizowany jest za pomocą prostych Właściwości i ich wartości. Identyfikowane zasoby oraz wartości ich właściwości pozwalają tworzyć *zdania* reprezentowane jako trójki RDF (**temat** [zasób], **predykat** [właściwość], **obiekt**).

3.1.3 Poziom składniowy

3.1.3.1 Turtle

Notacja Turtle (Terse RDF Triple Language) nie posiada statusu oficjalnego standardu formy serializacji RDF, lecz zyskała akceptację programistów Semantic Web dzięki czytelnej formie zapisu i jest obsługiwana przez narzędzia i biblioteki przetwarzające RDF.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://example.org/stuff/1.0/> .

<http://www.w3.org/TR/rdf-syntax-grammar>
  dc:title "RDF/XML Syntax Specification (Revised)" ;
  ex:editor [
    ex:fullname "Dave Beckett";
    ex:homePage <http://purl.org/net/dajobe/>
  ] .
```

Notacja Turtle jest używana w języku zapytań RDF SPARQL (SPARQL Protocol And RDF Query Language) *[13]* po klauzuli CONSTRUCT pozwalając na formułowanie *zapytania przez przykład* (*queries by example*).

3.1.3.2 N-Triples

N-Triples jest uproszczoną formą serializacji będącą podzbiorem formy Trutle, przeznaczona do definiowania zbiorów testowych danych RDF, zaproponowaną przez David'a Beckett'a. Charakteryzuje się liniowym zapisem zdań RDF oraz stosunkowo niewielką liczbą możliwości zapisu tego samego grafu RDF (porównując do innych notacji).

Dla odróżnienia od plików Notation 3, dane N-Triples umieszcza się w plikach o rozszerzeniu .nt

```
<http://w3.org/> <http://purl.org/dc/creator> "Dave Beckett" .
<http://w3.org/> <http://purl.org/dc/creator> "Art Barstow" .
<http://w3.org/> <http://purl.org/dc/publisher> <http://www.w3.org/> .
```

Każda linia w notacji N3 reprezentuje pojedyncze zdanie RDF lub komentarz, linie kończy znak ".". Każdy element zdania RDF (temat, predykat, obiekt) oddzielany jest znakiem spacji.

3.1.3.3 Notation 3 (N3)

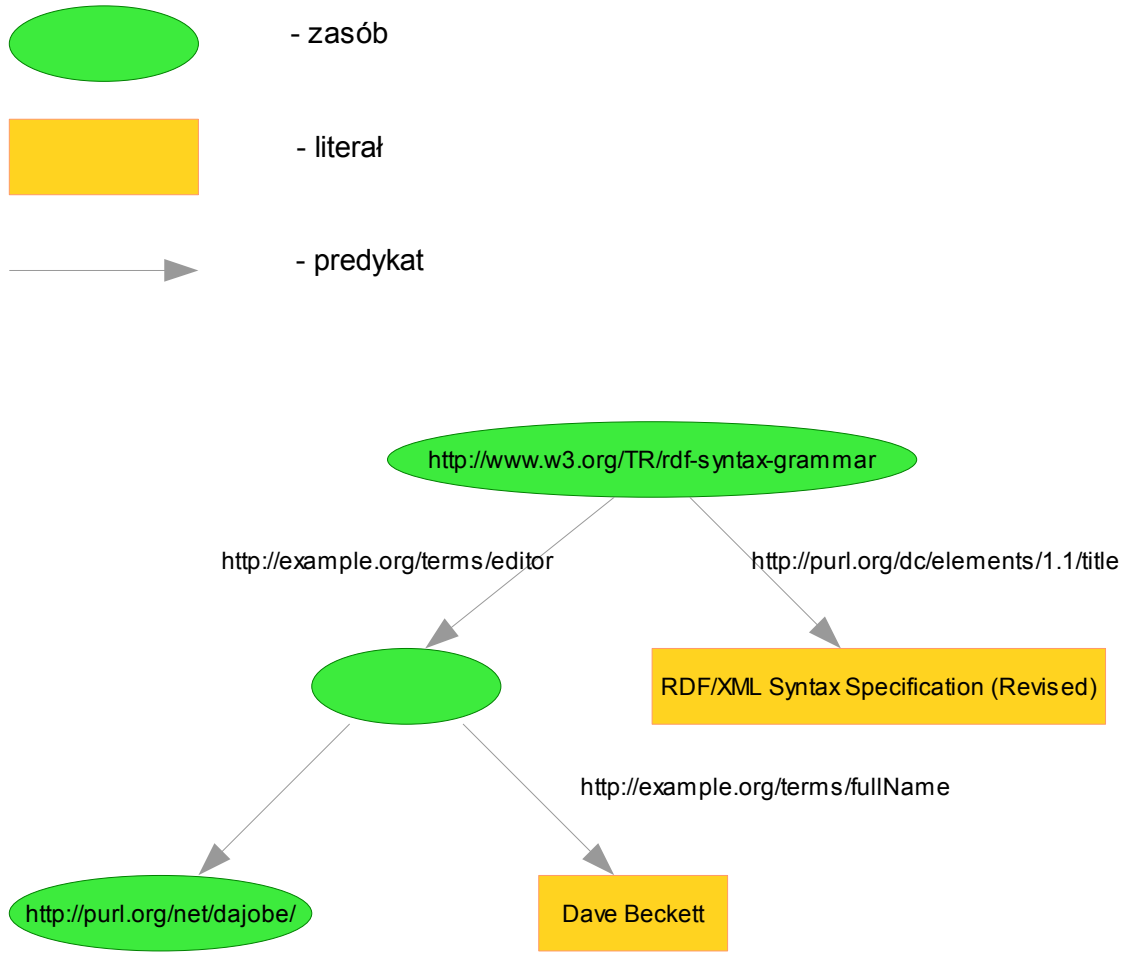
```
@prefix dc: <http://purl.org/dc/elements/1.1/>.

<http://pl.pjwstk.edu/publications/filtr_rdf>
  dc:title "Filtr RDF";
  dc:publisher "Polsko-Japońska Wyższa Szkoła Technik Komputerowych".
```

Notation3, w skrócie nazywana N3 jest formą serializacji zaprojektowaną w celu czytelnego dla człowieka zapisu danych RDF. Autorem tej formy jest Tim Berners-Lee oraz zespół Semantic Web community.

3.1.3.4 RDF/XML

Forma RDF/XML jest główną, przyjętą jako standard, metodą serializacji danych RDF. *Zasoby* oraz *predykaty* reprezentowane są przez URI *[12]*. URI mogą występować w skróconej formie tzw. *Qnames* wg. definicji dostępnej w *przestrzeni nazw* (*Namespaces*) składając się z *przedrostka* oraz *nazwy lokalnej*. *Literał* RDF reprezentowany jest przez tekstowy element XML *[8]* .

```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
  <ex:editor>
    <rdf:Description>
      <ex:homePage>
        <rdf:Description rdf:about="http://purl.org/net/dajobe/">
        </rdf:Description>
      </ex:homePage>
    </rdf:Description>
  </ex:editor>
</rdf:Description>

<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
  <ex:editor>
    <rdf:Description>
      <ex:fullName>Dave Beckett</ex:fullName>
    </rdf:Description>
  </ex:editor>
</rdf:Description>

<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
  <dc:title>RDF/XML Syntax Specification (Revised)</dc:title>
</rdf:Description>
```



Powyższy przykład pochodzący ze specyfikacji składni RDF/XML http://www.w3.org/TR/rdf-syntax-grammar/#section-Syntax-node-property-elements opisuje następujące elementu RDF:

1. Zasób RDF identyfikowany przez URI o wartości *http://www.w3.org/TR/rdf-syntax-grammar*
2. Predykat RDF oznaczony przez URI o wartości *http://example.org/terms/editor*
3. Zasób anonimowy (nie posiadający identyfikatora URI)
4. Predykat RDF oznaczony przez URI o wartości *http://example.org/terms/homePage*
5. Zasób RDF identyfikowany przez URI o wartości *http://purl.org/net/dajobe/*

3.1.4 Poziom strukturalny

3.1.4.1 Model RDF jako graf

Na poziomie strukturalnym dane RDF możemy rozpatrywać jako zbiór **Trójek RDF** tworzących skierowany graf RDF.

Trójka RDF inaczej nazywana **zdaniem RDF** (ang. *RDF Statement*) odpowiada koncepcji wyrażania faktów za pomocą prostych zdań w których wyróżnia się część nazywaną *tematem* odpowiadająca podmiotowi o którym mowa, *predykat* odpowiadający właściwości, przymiotowi oraz *obiekt* określający wartość przymiotu *[10]*.

Trójka RDF składa się z 3 elementów:

1. **temat** (ang. *subject*)
2. **predykat** (określany również jako *właściwość,* ang. *predicate, property*)
3. **obiekt** (ang. *object*)

Temat w zdaniu RDF to *zasób RDF* lub *zasób anonimowy RDF*.

Obiekt w zdaniu RDF to *zasób RDF*, *zasób anonimowy RDF* lub *literał.*

Graf RDF to zbiór zdań RDF - węzły grafu reprezentują temat lub obiekt, krawędź reprezentuje predykat. Graf utworzony ze zdań RDF reprezentuje koniunkcje tych zdań.

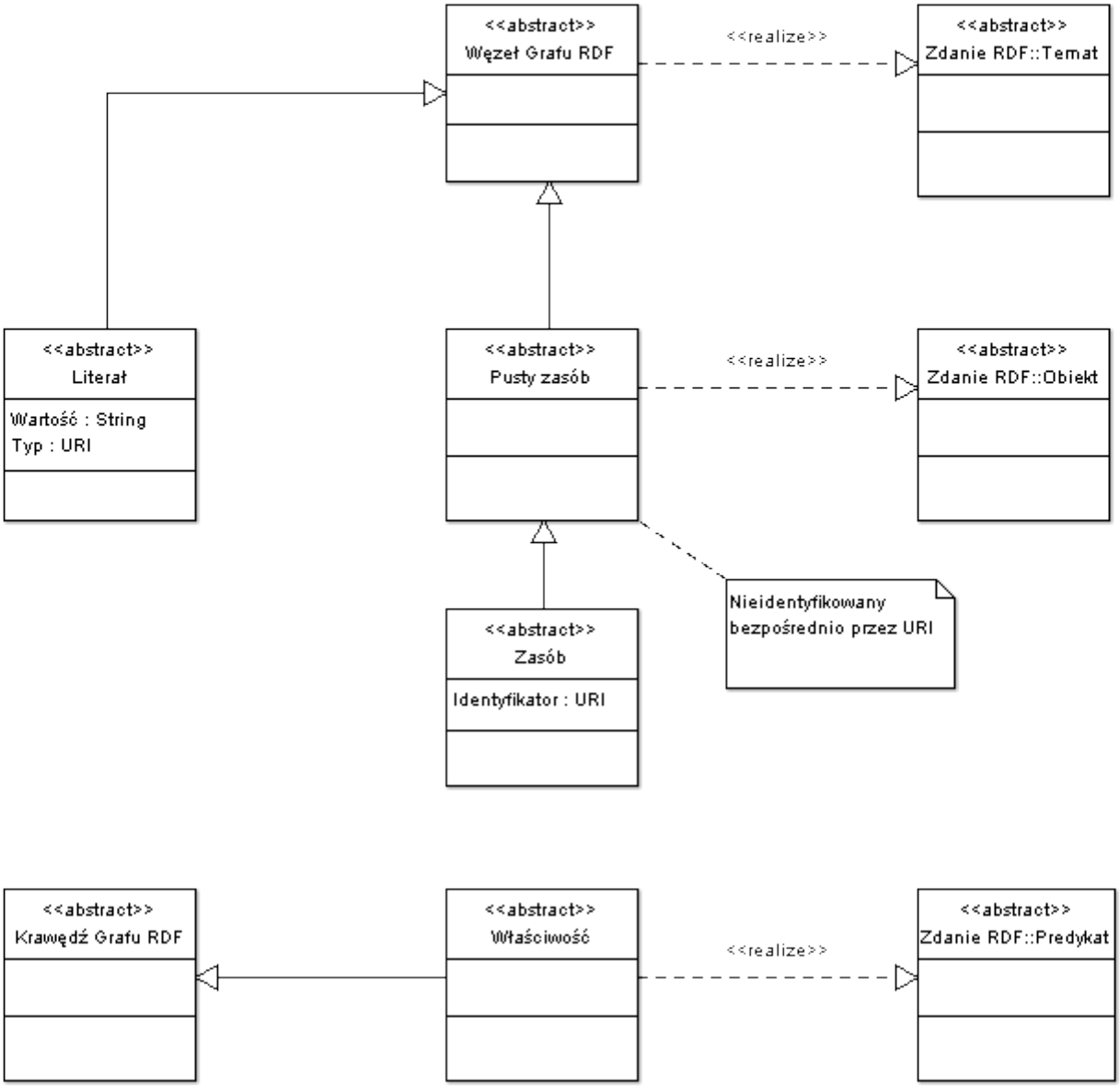


Diagram elementów grafu RDF

3.1.4.2 Struktura RDF

- **Predefiniowane predykaty RDF**

  RDF definiuje zbiór predefiniowanych, słownikowych wartości predykatów (URI). Predykaty te stosowane są do identyfikacji i opisu zasobów tworzących struktury języka RDF takie jak kolekcje, data typy literałów itp.

  Predykaty pre definiowane RDF opisywane prefiksememem "*rdf:*" odpowiadającym przestrzeni nazw URI: http://www.w3.org/1999/02/22-rdf-syntax-ns

  Niektóre węzły grafu RDF tworzące struktury charakterystyczne dla RDF identyfikowane są

przez przedefiniowany predykat: "*rdf:type*" (np. kolekcje), inne wyróżniane są już na poziomie składniowym dokumentu RDF (np. *Literał* za pomocą atrybutu XML: *rdf:parseType="Literal"*)

- **Węzły grafu RDF**
  - **Literał -** Literały identyfikują wartości takie jak liczby, daty, czy ciągi znaków za pomocą reprezentacji leksykalnej. Wartości reprezentowane przez literały można zapisać również jako zasób RDF identyfikowany przez URI odpowiadającym literałowi.

    Literał w *zdaniu RDF* może też występować jedynie jako *obiekt*.

    Specyfikacja RDF wyróżnia dwa rodzaje literałów:

    - ➔ **literał prosty** - rekomendowana forma zawierająca łańcuch znaków odpowiadający wartości literału oraz tag informujący o użytym języku (zgodny z [RFC-3066])
    - ➔ **literał typowany** - łańcuch znaków wraz z predefiniowanym predykatem "*rdf:datatype*" wskazujący *datatyp* literału. RDF pre definiujee jedynie jeden rodzaj *datatypu* tj. "*rdf:XMLLiteral"* przeznaczony do oznaczania literałów zawierających dane XML wewnątrz dokumentów RDF.
  - **Zasób -** Zasób RDF w grafie RDF reprezentuje najmniejszą identyfikowalną porcję danych. Podstawową koncepcją i nośnikiem danych dla zasobu RDF jest **URI**. Dzięki użyciu URI zasób RDF może reprezentować osoby, pojęcia, typy rozpatrywanych pojęć oraz wartości rozpatrywanych pojęć. Zasób RDF w zdaniu RDF może występować zarówno jako *temat* jak i *obiekt*.
  - **Zasób anonimowy -** Zasób anonimowy w odróżnieniu od *zasobu RDF* nie jest identyfikowany przez URI. Zazwyczaj w grafie RDF występuje jako węzeł pomocniczy tworzący kolekcje lub złożone struktury (np. strukturyzowane właściwości zasobu tj. adres pocztowy) za pomocą kombinacji predykatu "*rdf:type"* i predyktatu "*rdf:value"*. Zasób anonimowy pozwala reprezentować relacje n-arne pomiędzy zasobami w grafie RDF, często występuje niejawnie w dokumentach RDF, pozwala uniknąć tworzenia pośredniczących zasobów RDF ze sztucznie wygenerowanym URI nie mającym autonomicznego znaczenia. Zasób anonimowy w zdaniu RDF może występować zarówno jako *temat* jak i *obiekt*. Specyfikacja RDF **nie obejmuje sposobu numeracji** zasobów anonimowych, w praktyce zależy ona od użytej notacji dokumentów RDF i przyjętych konwencji. Specyfikacja nie przewiduje możliwości rozróżnienia czy dane dwa zasoby anonimowe są tożsame czy też różne w grafie RDF. Przyjętą konwencją jest numeracja zasobów anonimowych za pomocą wewnętrznego identyfikatora "*rdf:nodeID*" **unikalnego jedynie w obrębie danego dokumentu RDF**.
- **Reprezentacja kolekcji**
  - **Kontenery –** Kontenery są to klasy RDF, niosące informacje o elementach w nich zawartych jako jego członkach, nie informująca jednak czy lista członków kontenera wymieniona w danym

dokumencie RDF jest pełna.

W3C specyfikuje trzy rodzaje kontenerów RDF:

1. **Bag** - identyfikowana przez wartość predykatu *rdf:type* równym *rdf:Bag*, jest to nieuporządkowana kolekcja z powtórzeniami
2. **Seq** - identyfikowana przez wartość predykatu *rdf:type* równym *rdf:Seq* - odpowiada uporządkowanej kolekcji z powtórzeniami
3. **Alt** - identyfikowana przez wartość predykatu *rdf:type* równym *rdf:Alt* - kolekcja wartości alternatywnych

Kontenery mogą zawierać zasoby RDF, zasoby anonimowe lub literały.

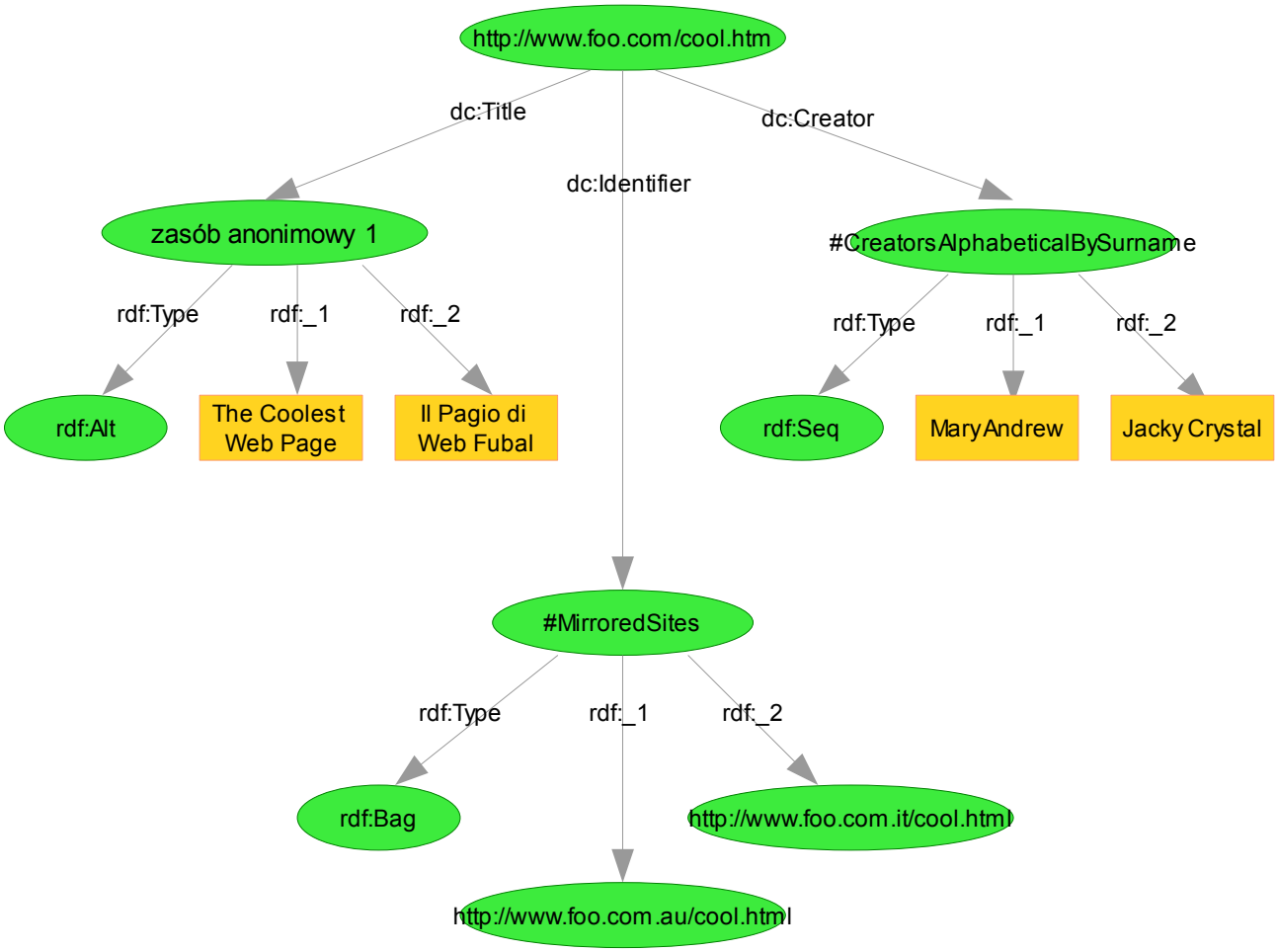
Elementy kontenera RDF identyfikowane są przez predykaty o wartościach "rdf:_*n*" gdzie n jest kolejną wartością liczby naturalnej nadającą kolejność elementom w kontenerze lub, na poziomie składniowym, w dokumencie XML, przez znacznik <*rdf:li*> - w takim przypadku o porządku w kontenerze decyduje kolejność elementu w dokumencie XML.

Zasób reprezentujący kontener może być zarówno zasobem identyfikowanym przez URI jak i anonimowym.

W praktyce kontener RDF reprezentowany jest przez zasób anonimowy i nie występuje bezpośrednio w dokumencie RDF np:

```
<rdf:RDF>

  <rdf:Description about="http://www.foo.com/cool.html">

    <dc:Creator>
      <rdf:Seq ID="CreatorsAlphabeticalBySurname">
        <rdf:li>Mary Andrew</rdf:li>
        <rdf:li>Jacky Crystal</rdf:li>
      </rdf:Seq>
    </dc:Creator>

    <dc:Identifier>
      <rdf:Bag ID="MirroredSites">
        <rdf:li rdf:resource="http://www.foo.com.au/cool.html"/>
        <rdf:li rdf:resource="http://www.foo.com.it/cool.html"/>
      </rdf:Bag>
    </dc:Identifier>

    <dc:Title>
      <rdf:Alt>
        <rdf:li xml:lang="en">The Coolest Web Page</rdf:li>
        <rdf:li xml:lang="it">Il Pagio di Web Fuba</rdf:li>
      </rdf:Alt>
    </dc:Title>

  </rdf:Description>

</rdf:RDF>
```

źródło: http://www.w3.org/2000/10/rdf-tests/RDF-Model-Syntax_1.0/ms_7.2_1.rdf



- **Kolekcje** - Kolekcja RDF jest zasobem identyfikowanym przez pre definiowany predykat *rdf:List* zawierającym zasoby, lub literały w strukturze listy. Elementy listy wskazują predykaty *rdf:first* oraz *rdf:rest*. Lista może zawierać pre definiowany zasób RDF *rdf:nil*. Kolekcje RDF, w odróżnieniu od kontenerów RDF, pozwalają określić czy dana kolekcja jest *zamknięta* tzn. czy istnieją elementy (zasięg asercji jest globalny) tej kolekcji nie wymienione w danym dokumencie.

3.1.5 Poziom semantyczny

3.1.5.1 Reprezentacja faktów w RDF

RDF na poziomie semantycznym można rozpatrywać jako ustandaryzowany nośnik metadanych opartych na wcześniej uzgodnionych słownikach.

Trójka RDF w tym kontekście, pozwala wyrażać **proste fakty**, związek pomiędzy tematem a obiektem, reprezentującymi dwa byty, określa predykat i jednocześnie nazywa i jednoznacznie identyfikuje tę relacje

za pomocą URI.

W odróżnieniu od rachunku predykatów pierwszego rzędu, gdzie predykaty mogą przyjmować zdefiniowaną liczbę argumentów oraz w odróżnieniu od systemów relacyjnych baz danych, gdzie tabele mogą zawierać jedynie zdefiniowaną liczbę kolumn *trójka RDF* może przyjąć jedynie *jeden* temat oraz *jeden* obiekt połączone *jednym* predykatem. Ograniczenie dla trójki RDF wynika z spełnienia **założenia otwartości** struktury danych przeznaczonych do opisu ontologii (tzn. RDF). Aby przedstawić złożone zdanie o faktach w RDF należy je **zdekomponować** do zbioru trójek RDF. Zbiór trójek RDF tworzący graf RDF przeznaczony jest do dalszego rozszerzenia poprzez dodanie nowej definicji (zdań RDF) w opisywanej ontologii lub próbie połączenia ontologii.

3.1.5.2 Postulat otwartości

Założenie otwartości RDF w kontekście użycia jako narzędzia opisu ontologii:

- każdy może stawiać *zdania* o wszelkich *zasobach*
- brak zagwarantowania *kompletności* języka
- brak zagwarantowania *spójności* danych opisywanych przez RDF

3.1.5.3 Sylogizm zdań RDF (entailment)

Otwarta specyfika RDF nie daje gwarancji spójności danych lecz oparcie struktury na trójkach RDF dostarcza podstawowych mechanizmów definiowania formalnych semantyk.

Zarówno specyfikacja RDFS jak i OWL rozszerza RDF, bazując na strukturze trójek, dodając predefiniowane zestawy predykatów i zasobów *[11]*. Rozszerzenia mają na celu dostarczyć zestandaryzowanych środków opisu ontologii. Oparcie na strukturze grafu utworzonego z trójek RDF pozwala automatom przetwarzającym dane RDF na przeprowadzanie sądów logicznych na ontologiach.

Podstawą sądu jest jednak zawsze gramatyka uzgodniona przez grupy interesu, twórców oprogramowania lub, w eksperymentalnych projektach, samych użytkownikach systemu (np. portalu społecznościowego), tworzących gramatykę wokół treści współdzielonej z innymi użytkownikami.

3.1.6 RDF a XML

Podstawową różnicą pomiędzy strukturą danych w dokumentach XML a RDF jest hierarchiczna struktura XML podczas gdy RDF posiada bardziej spłaszczoną strukturę grafu opartą na trójkach RDF. Różnica te pośrednio powoduje stosunkowo nieczytelny dla człowieka charakter zapisu RDF/XML. Hierarchiczna natura dokumentów XML wymusza na aplikacjach je przetwarzających odczytanie dokumentu XML od tagu otwierającego, aż do tagu zamykającego rozpatrywany element XML wraz z wszystkimi pod elementami. W rezultacie takiego podejścia dokumenty XML są w całości przetrzymywane w pamięci, co może negatywnie wpływać na zakres danych możliwych do przetworzenia w szczególności dla dużych dokumentów XML lub

przetwarzaniu wielu dokumentów XML. Przetwarzanie dokumentów RDF, również tych seriazlizowanych metodą RDF/XML, pozwala na odczytywanie poszczególnych elementów RDF, bez potrzeby odczytywania elementów będących w relacji z rozpatrywanym. Dane w dokumentach RDF, w odróżnieniu od XML, mogą zostać zapisane w oderwaniu od fizycznej lokacji, kolejności względem innych elementów w dokumencie.

3.1.7 Funkcje integracji

Integracja danych opartych o RDF można podzielić na trzy poziomy *[6]*:

Agregacja – dotyczy integracji dwóch lub więcej grafów RDF (źródeł danych) o różnym schemacie. Połączenia pomiędzy zasobami w takim przypadku tworzone są *explicite*.

Integracja – ma miejsce w przypadku zastosowania dwóch lub więcej źródeł danych współdzielących ten sam schemat.
Proces integracji może zawierać semantycznie odwzorowanie zasobów, identyfikacje zasobów identycznych i ich scalanie oraz rozwiązywanie nazw URI.

Wnioskowanie – jeden lub więcej grafów RDF współdzielących ten sam schemat poddanych procesowi dodawania nowych zasobów i predykatów, zdań RDF, na podstawie istniejących zdań RDF i reguł RDFS, OWL.

3.1.8 Dojrzałość standardu

Większość dostępnych narzędzi przetwarzających RDF nie w pełni implementuje specyfikacje samego RDF

Specyfikacje związane z RDF i budowane na jego podstawie takie jak RDFS czy OWL w większości zaimplementowane są jedynie w ograniczonym zakresie, często są to narzędzia mające status badawczy.

3.1.9 Gramatyki/słowniki RDF i technologie budowane na czubku RDF

3.1.9.1 RDFS - RDF Schema

RDFS (RDF Schema) jest językiem opisu zasobów, taksonomii rozszerzającym RDF. RDFS wprowadza pojęcia *klas zasobów*, *domeny* (*domain*) oraz *zasięgu* (*range*) predykatów, pozwala opisać relacje pomiędzy klasami zasobów.

RDFS pozwala wnioskować o właściwościach predykatów w grafie RDF - np o tranzytywności relacji pomiędzy danymi zasobami, pozwala wnioskować o przynależności zasobu do danej klasy na podstawie *domeny* predykatu jaki posiada dany zasób (definiując taką regułę za pomocą *zasięgu* predykatu) jak również na podstawie przynależności do innej klasy zasobów (i w tym sensie pozwala na wnioskowaniu o istnieniu predykatów, nie wymienionych *explicite*, związanych z relacjami zachodzących pomiędzy klasami na

podstawie zdefiniowanych predykatów zasobów).

Predykaty w RDFS mają swoją reprezentacje jako zasób RDF należący do pre definiowanej klasy RDFS odpowiadającej predykatom (na poziomie RDF opis zdań RDF za pomocą zdań RDF nazywany jest *reifikacją* **)**, RDFS rozszerzający RDF wprowadza pre definiowane wartości predykatów i klas dla takiego auto opisu jak i dla opisu relacji związanymi z wprowadzeniem pojęcia klasy zasobów).

RDFS rozszerza proces scalania danych RDF o możliwość identyfikacji semantycznej tożsamości predykatów i zasobów na podstawie ich *klas*, *domeny* i *zasięgu*.

RDFS wprowadza mechanizmy pozwalające na podstawowy zapis ontologii. Dodatkowo, dla RDFS oraz RDF jako jego podstawy, został wyspecyfikowany przez W3C zbiór reguł sądu logicznego (*entailment rules*) *[11]* określający zasady uzupełnienia danego grafu RDFS o zasoby i predykaty, które wynikają z danego grafu, a nie są *explicite* w nim zawarte (np. predykat określający typ danego zasobu można obliczyć na podstawie typów obiektów należących do tej samej klasy) (dokument W3C: *RDF Semantics*)

RDFS jest oficjalną rekomendacją W3C.

3.1.9.2 OWL - Web Ontology Language

OWL (Web Ontoloty Language) jest oficjalną specyfikacją W3C języka opisu ontologii. Rozszerza RFDS o dodatkowe słowniki predykatów, określające relacje pomiędzy klasami, liczebność, właściwości samych predykatów oraz bogatsze typowanie.

OWL pozwala opisać *explicite* semantyczną tożsamość (lub jej brak) wybranych predykatów, klas lub instancji klas, kategoryzuje predykaty ze względu na właściwości relacji (funkcja, funkcja odwrotna, symetryczność, tranzytywność), wprowadza słownikowy zasób reprezentujący instancje klasy, słownikowe predykaty określające rozłączność i równoważność klas, wprowadza jawną obsługę wielu URI identyfikujących jeden zasób itp.

O ile RDFS może być używany jako podstawowy język opisu ontologii, OWL dostarcza szereg środków semantycznego opisu wiedzy przeznaczonego do automatycznego przetwarzania. Jako konstrukcja o wyższym poziomie abstrakcji posiada stosunkowo nieliczne implementacje przetwarzające ten język w porównaniu do RDF i RDFS. Jednak na poziomie syntaktycznym dokumenty OWL mogą być traktowane jako zapis RDF i powinny przechodzić pomyślnie parsowanie RDF.

OWL posiada trzy podjęzyki *[9]*: w kolejności malejącej ekspresywności: OWL Full, OWL DL i OWL Lite:

- **OWL Full** obejmuje pełny zakres specyfikacji języka Web Ontology Language. Pozwala arbitralnie łączyć słownikowe prymitywy OWL wraz z RDF i RDFS. Jest w pełni kompatybilny z RDF zarówno semantycznie jak i syntaktycznie: poprawny dokument RDF jest również poprawnym dokumentem OWL. Pełna swoboda łączenia słownikowych predykatów OWL oraz ekspresywność OWL Full implikuje najbardziej złożony charakter, z trzech pod języków OWL, procesu

przetwarzania, nie dając gwarancji kompletności obliczeniowej.

- **OWL DL** jest podzbiorem OWL Full starającym się wypełnić kompromis pomiędzy ekspresywnością OWL a kompletnością obliczeniową. OWL DL realizuje swoje założenia poprzez ograniczenia pewnych konstrukcji OWL Full, takie jak *type separation* (np. zasób reprezentujący klasę, nie może być jednocześnie właściwością lub instancją klasy, a właściwość, nie może być klasą ani instancją)
- **OWL Lite** jest najprostszym w implementacji i najczytelniejszym z trzech pod języków OWL z ograniczoną ekspresywnością. Wspiera głównie hierarchie klasyfikacji oraz uproszczone asocjacje (np ogranicza wartości liczebności asocjacji tylko do 0 lub 1).

3.1.9.3 Dublin Core

Dublin Core jest to zbiór elementów opisu zasobów z dziedziny materiałów wideo, nagrań dźwiękowych, tekstu , obrazów oraz materiałów je łączących takich jak strony www.

Standard ten został opracowany i jest utrzymywany przez specjalistyczne grupy zainteresowania związane z naukami informatycznymi, bibliotekoznawstwem, muzeologią i naukami związanymi, zarówno na płaszczyźnie teoretycznej, jak i praktycznej. Proces uzgadniania ontologii Dublin Core odbywa się za pośrednictwem forum organizacji Dublin Core Metadata Initiative.

W podstawowej wersji 1.1 standard składa się z piętnastu elementów:

1. Title
2. Creator
3. Subject
4. Description
5. Publisher
6. Contributor
7. Date
8. Type
9. Format
10. Identifier
11. Source
12. Language
13. Relation
14. Coverage
15. Rights

Każdy z elementów opisujący dany zasób może być wielokrotnie powtarzany i jest opcjonalny. RDF jest

używany jako domyślna implementacja, a elementy DC reprezentowane są jako predykaty RDF.

3.2 Implementacja SBA - system ODRA

System Odra jest implementacją podejścia stosowego (SBA) do budowy języków zapytań. Bazuje na koncepcji ewaluacji zapytań na stosie środowiskowym (environmental stack, ENVS) składającym się z dynamicznego zbioru sekcji przechowujących nazwane referencje do używanych w zapytaniach obiektów oraz koncepcji przechowywania wyników zapytań na stosie wyników (result stack, QRES).

System jest kolejną implementacją podejścia SBA powstałą w wyniku prac grupy badawczej skupionej wokół Prof. Dr. hab. Kaziemierza Subiety zrzeszającej naukowców oraz studentów wymieniających wnioski z implementacji na forum http://odraforum.pjwstk.edu.pl.

4 Filtr RDF dla systemu ODRA

4.1 Przetwarzanie danych RDF w systemie ODRA - założenia

Dane RDF zostały odwzorowane w systemie ODRA jako obiekty klas SBQL. Rozwiązanie to pozwala programiście języka SBQL na dalsze przetwarzanie i odpytywanie danych RDF.

Mimo podobieństw topologicznych pomiędzy organizacją danych w systemie SBA i języku RDF autor pracy podjął decyzje projektową przeniesienia struktury RDF na poziom obiektów SBQL podyktowaną potrzebą zapewnienia struktury danych RDF otwartej na dalsze przetwarzanie i ewentualne nadbudowanie konstrukcji o wyższym poziomie abstrakcji. Dodatkowo mapowanie struktury RDF do SBA napotkało na ograniczenia związane z implementacją ODRA (więcej na temat ograniczeń w rozdziale: 6.2 Napotkane błędy i ograniczenia)

Podczas importu danych system korzysta z przygotowanego kodu bibliotecznego SBQL.

Klasy SBQL dostarczone z importerem mogą zostać rozszerzone oraz mogą stanowić podstawę dla aplikacji przetwarzającej dane RDF. Takie podejście dostarcza programiście jednolitej struktury danych i pozwala na ich efektywne przetwarzanie oraz odpytywanie jedynie na podstawie rozpatrywanej gramatyki.

4.2 Reprezentacja RDF jako obiekty SBQL

4.2.1 Reprezentacja danych RDF w systemie ODRA

Głównym założeniem projektowym dla filtru RDF dla systemu ODRA jest reprezentacja danych RDF za pomocą dedykowanych klas SBQL.

Podejście polegające na budowie struktury przy użyciu wcześniej przygotowanych klas SBQL zamiast próby bezpośredniego tworzenia obiektów w składzie ODRA reprezentującymi *temat RDF,* zostało podyktowane potrzebą elastycznego rozszerzania modułu przetwarzającego dane RDF oraz potrzebą zapewnienia rozszerzalnej struktury grafu RDF.

Graf RDF jest odwzorowany bezpośrednio w systemie Odra jako kolekcje obiektów klasy *RDFPropertyClass* reprezentujące krawędzie grafu oraz *RDFNodeClass* i klas z niej dziedziczących jako wierzchołki grafu.

Właściwości zasobów RDF charakterystyczne jedynie dla języka RDF związane z określaniem typu zasobu (*rdf:type*), liczności i innych są również reprezentowane jako elementu grafu RDF.

Autor pracy podjął decyzje odwzorowaniu kolekcji RDF jako asocjacje obiektów reprezentujący *zasób*

anonimowy RDF przy pomocy predykatów określających typ i wystąpienia w kolekcji RDF.

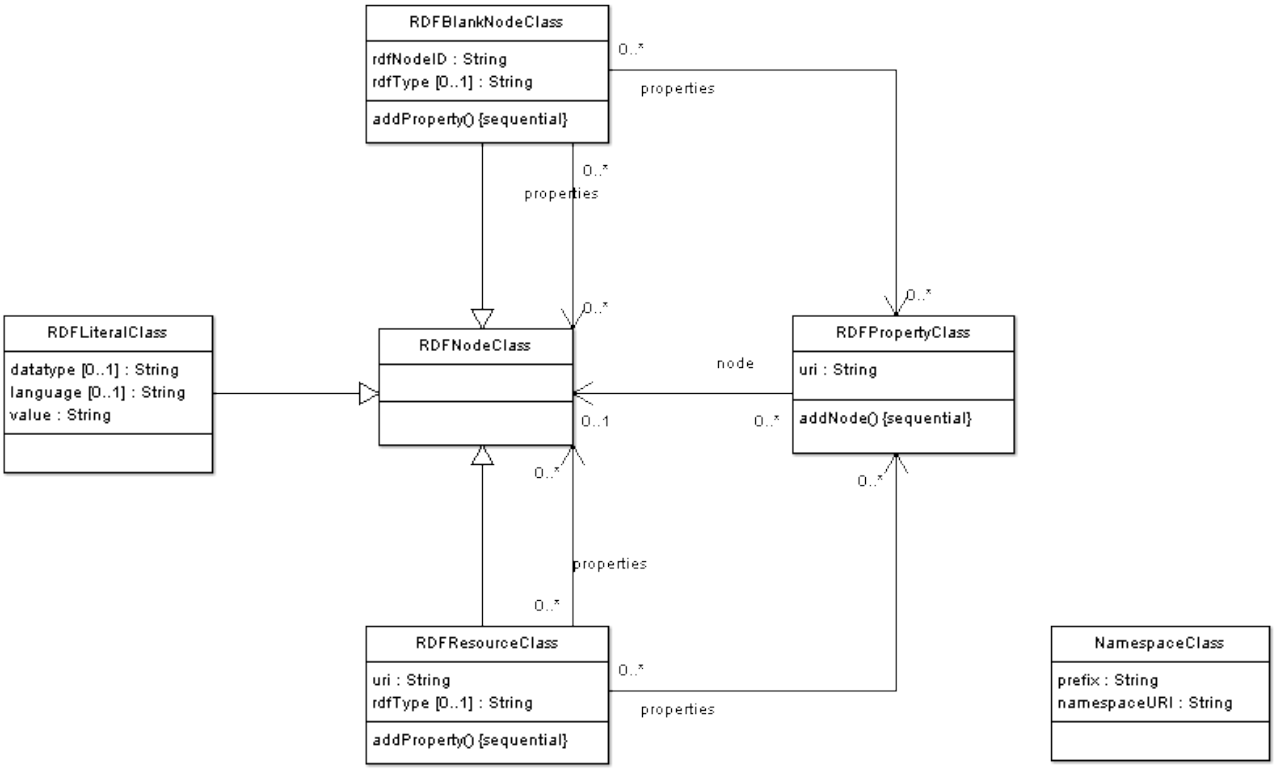
Założenie zostało podyktowane następującymi czynnikami:

1. jest to bezpośrednie odwzorowanie grafu RDF do składu Odra
2. zapewnia kompatybilną strukturę danych dla przyszłych wersji importera RDF dla systemu Odra
3. deleguje odpowiedzialność identyfikacji zasobów anonimowych na dodatkowe mechanizmy obecne w przyszłych warstwach aplikacyjnych zrębu programistycznego RDF-SBQL

4.2.2 Klasy SBQL

Elementy grafu RDF są reprezentowane przez klasy SBQL modułu bibliotecznego RDF.sbql

- *RDFNodeClass*
- *RDFBlankNodeClass*
- *RDFResourceClass*
- *RDFLiteralClass*
- *RDFPropertyClass*



klasy modułu bibliotecznego RDF.sbql

4.2.3 Obsługa przestrzeni nazw RDF

Powszechną metodą zapisywania URI identyfikującej zasoby oraz właściwości dla każdej z metod serializacji jest skrócona forma, dla notacji XML/RDF tzw. *QName*.

Konstrukcje *Qname* specyfikuje dokument grupy W3C: http://www.w3.org/TR/REC-xml-names/ w szczególności we fragmencie: http://www.w3.org/TR/REC-xml-names/#NT-QName.

Rekomendacja składni RDF w notacji XML/RDF specyfikuje konstruowanie QName jako konkatenacji łańcuchów znaków reprezentujących przedrostek (*prefix*) i części lokalną URI (*localPart*) połączoną znakiem ":".

Podczas importu danych do modułu RDF, filtr tworzy obiekty klasy NamespaceClass dla każdego wcześniej nie napotkanego przedrostka *QName* posiadającego definicje w sekcji przestrzeni dokumentu RDF.

Klasa *sbql* reprezentująca przestrzenie nazw w module RDF:

```
class NamespaceClass {
  instance Namespace : {
    prefix : string;
    namespaceURI : string;
  }
}
```

Domyślnie, podczas importu URI zasobów oraz predykatów zapisywane są w skróconej formie *QName,* a ich rozwinięcie dla programisty systemu Odra dostępne jest w ekstensji klasy *NamespaceClass*.

Importer pozwala na zapis URI w pełnej formie. Aby przeprowadzić import danych RDF identyfikowanych przez URI zapisanej w pełnej formie należy użyć parametru "*FullURI"* dla komendy importu, np.:

```
load "C:\resources\rdf\RDF-Model-Syntax_1.0\ms_7.4_2.rdf" using
RDFImporter("FullURI")
```

Import danych RDF do składu Odra identyfikowanych przez pełną formę URI nie wpływa na wypełnianie mapowania URI ekstensjiii klasy *NamespaceClass* - filtr również tworzy obiekty klasy *NamespaceClass* dla każdego wcześniej nie napotkanego przedrostka *QName* .

4.2.4 Moduł biblioteczny RDF.sbql

Kod sbql modułu bibliotecznego RDF.sbql został umieszczony w lokalizacji

/res/standard_library/RDF.sbql

(system plików po stronie serwera), względem pliku bazy ODRA.

Obecność modułu w tej lokalizacji jest wymagana dla poprawnego działania importera - importer buduje pod moduł RDF w module w którym został uruchomiony import danych RDF, a następnie wywołuje procedury sbql w procesie budowania struktury odpowiadającej danym RDF.

Moduł RDF został podzielony na **sekcje**:

1. **struktura danych** - zawiera ekstensje klas reprezentujących elementy grafu RDF oraz definicje tych klas
2. **interfejs importera** - zawiera procedury wywoływane przez importer. Trzonem sekcji jest sześć procedur odpowiadającym *zdaniu RDF*, tworzące krawędź łączącą dwa wierzchołki grafu RDF, uwzględniające typy obiektów sbql tj.
   - *makeStatementResourceToResource* - wywołuje procedurę tworzącą obiekt klasy *RDFPropertyClass* reprezentujacą krawędź grafu RDF oraz obiekty typu *RDFResourceClass* reprezentujące wierzchołki grafu RDF połączone krawędzią
   - *makeStatementResourceToLiteral* - wywołuje procedurę tworzącą obiekt klasy *RDFPropertyClass* reprezentujacą krawędź grafu RDF oraz obiekty typu *RDFResourceClass* i *RDFLiteralClass* reprezentujące wierzchołki grafu RDF połączone krawędzią
   - *makeStatementResourceToAnon* - wywołuje procedurę tworzącą obiekt klasy *RDFPropertyClass* reprezentujacą krawędź grafu RDF oraz obiekty typu *RDFResourceClass* i *RDFBlankNodeClass* reprezentujące wierzchołki grafu RDF połączone krawędzią
   - *makeStatementAnonToResource* - wywołuje procedurę tworzącą obiekt klasy *RDFPropertyClass* reprezentujacą krawędź grafu RDF oraz obiekty typu *RDFBlankNodeClass* i *RDFResourceClass* i reprezentujące wierzchołki grafu RDF połączone krawędzią
   - *makeStatementAnonToAnon* - wywołuje procedurę tworzącą obiekt klasy *RDFPropertyClass* reprezentujacą krawędź grafu RDF oraz obiekty typu *RDFBlankNodeClass* reprezentujące wierzchołki grafu RDF połączone krawędzią
   - *makeStatementAnonToLiteral* - wywołuje procedurę tworzącą obiekt klasy *RDFPropertyClass* reprezentujacą krawędź grafu RDF oraz obiekty typu *RDFBlankNodeClass* reprezentujące wierzchołki grafu RDF połączone krawędzią

Procedury SBQL wywoływane przez importer przyjmują argumenty jedynie typu *String*.

```
makeStatementResourceToLiteral(res_uri : string; res_type : string; prop_uri :
string; lit_value : string; lit_datatype : string; lit_language : string) {
  res : ref RDFResourceClass;
  res := ref (createResource(res_uri;res_type));
  prop : ref RDFPropertyClass;
  prop := ref (createProperty(prop_uri));

  res.addProperty(prop);
  prop.addNode(ref(create permanent RDFLiteral (lit_value as value, lit_datatype
as datatype, lit_language as language)));
}
```

Przykład procedury SBQL tworzenia trójki zasób-predykat-literał

3. **procedury tworzenia obiektów odpowiadającym elementom grafu RDF** - procedury tworzą lub zwracają poprzednio utworzone obiekty identyfikowane przez URI.

```
createResource(res_uri : string; rdfType : string) : RDFResource {
  if (exists (RDFResource where uri = res_uri)){ return (RDFResource where uri =
res_uri);}
  return (create permanent RDFResource (res_uri as uri, rdfType as rdfType));
}
```

4. **procedury operujące na grafie RDF** - sekcja przeznaczona na procedury przetwarzające graf RDF w momencie następującym po imporcie.

4.3 Proces importu danych RDF do systemu ODRA

1. Wywołanie przez użytkownika komendy importu danych RDF w środowisku ODRA CLI do danego modułu. Wywołanie przy użyciu składni:

```
load "<zasób>" using <nazwa plugin'u>
```

lub:

```
load "<zasób>" using <nazwa plugin'u>("parametry")
```

gdzie:

zasób – ścieżka do pliku rdf

nazwa plugin'u – nazwa pluginu dla komendy *load.* Dla filtru RDF aktualnie zaimplementowany jest jeden plugin o nazwie *RDFImporter.*

parametry – lista parametrów rozpoznawanych przez importer rozdzielonych znakami:

[spacja] „,”, „;”, \n, \t, \r, \f

Aktualny stan implementacji przewiduje parametry formy URI z jakimi zostaną importowane zasoby i predkaty rdf do składu systemu Odra:

FullURI – URI identyfikujące zasoby oraz predykaty RDF zapisywane będą w pełniej formie, pominięcie tego parametru w wywołaniu procedury importu spowoduje import danych RDF z polami URI w skróconej formie tzw. *Qname*

Importer dostarcza również możliwość eksperymentalnego importu danych rdf w formie serializacji innej niż RDF/XML:

N3 – wywołanie importu przy użyciu parsera rdf formy serializacji N3

N-TRIPLE - wywołanie importu przy użyciu parsera rdf formy serializacji N-Triple

przykład:

```
load "C:\rdf\ms_7.4_2.rdf" using RDFImporter
```

```
load "C:\rdf\ms_7.4_2.nt" using RDFImporter("FullURI, N-TRIPLE")
```

2. W danym module zostaje skompilowany plik biblioteczny RDF.sbql
   Importer kompiluje plik biblioteczny RDF.sbql i buduje pod moduł o nazwie "RDF" (np. dla importu do modułu o nazwie "modul_importu" zostanie zbudowany moduł o nazwie "modul_importu.RDF").
   W pod module zostają umieszczone ekstensje klas tworzących elementy grafu RDF w składzie ODRA:

```
RDFNode : RDFNodeClass[0..*];
RDFLiteral : RDFLiteralClass[0..*];
RDFBlankNode : RDFBlankNodeClass[0..*];
RDFResource : RDFResourceClass[0..*];
RDFProperty : RDFPropertyClass[0..*];
Namespace : NamespaceClass[0..*];
```

3. Importer wywołuje metody pod modułu RDF, modułu wybranego jako cel importu, do budowania

grafu RDF, przechodząc wszystkie krawędzie, reprezentujące predykaty, grafu RDF.

Dla każdego predykatu wywoływana jest procedura odpowiadająca *zdaniu RDF* budująca strukturę grafu RDF w składzie Odra "zdanie po zdaniu":

- *makeStatementResourceToResource* - wywoływane jest dla krawędzi łączące zasoby
- *makeStatementResourceToLiteral* - wywoływane jest dla krawędzi łączące zasób z literałem
- *makeStatementResourceToAnon* - wywoływane jest dla krawędzi łączącej zasób z zasobem anonimowym
- *makeStatementAnonToResource* - wywoływane jest dla krawędzi łączącej zasób anonimowy z zasobem
- *makeStatementAnonToAnon* - wywoływane jest dla krawędzi łączący zasoby anonimowe
- *makeStatementAnonToLiteral* - wywoływane jest dla krawędzi łączącej zasób anonimowy z literałem

Każda z procedur SBQL tworzy obiekt klasy RDFProperty reprezentujący predykat RDF oraz tworzy lub wywołuje ze składu już utworzony obiekt odpowiedniej klasy reprezentujący węzły połączone predykatem.

Dla każdego wcześniej nie napotkanego URI identyfikującego zasób lub predykat tworzony jest obiekt klasy *NamespaceClass* reprezentujący parę: *prefiks* oraz odpowiadającą mu przestrzeń nazw.

4.4 Przykładowe zapytania SBQL do danych RDF

4.4.1 Hello World

Najprostszym zapytaniem do modułu SBQL przechowującego dane RDF będzie wypisanie uri identyfikujące zasoby:

```
RDFResource.uri
```

lub URI predykatów RDF:

```
RDFProperty.uri
```

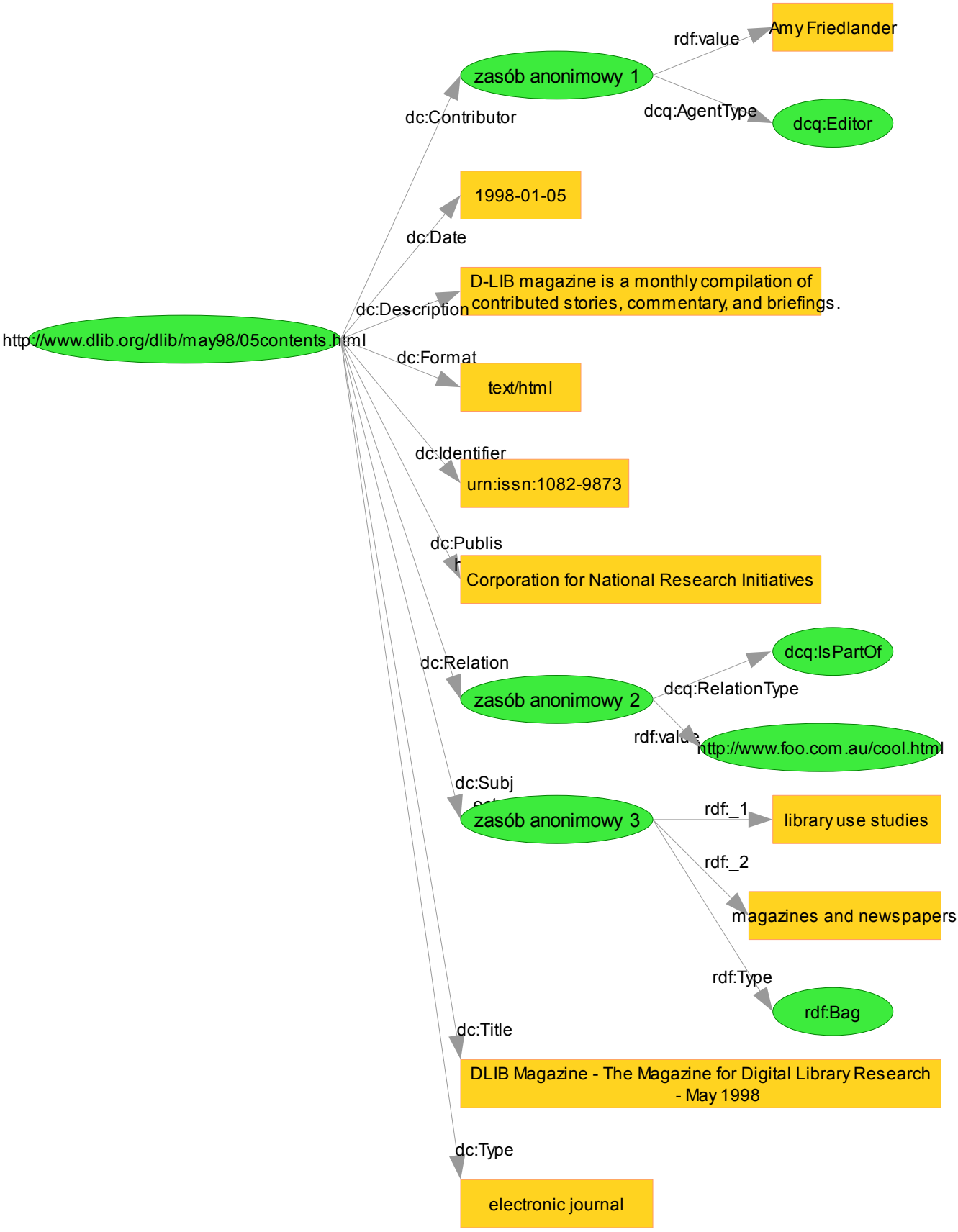
4.4.2 Zapytania do zasobów ontologii Dublin Core

Bardziej zaawansowanym przykładem są zapytania do zbioru zasobów spełniającego gramatykę Dublin Core:

```
<?xml version="1.0"?>

<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/metadata/dublin_core#"
    xmlns:dcq="http://purl.org/metadata/dublin_core_qualifiers#">

    <rdf:Description about="http://www.dlib.org/dlib/may98/05contents.html">
      <dc:Title>DLIB Magazine - The Magazine for Digital Library Research
        - May 1998</dc:Title>
      <dc:Description>D-LIB magazine is a monthly compilation of
       contributed stories, commentary, and briefings.</dc:Description>
      <dc:Contributor rdf:parseType="Resource">
        <dcq:AgentType
          rdf:resource="http://purl.org/metadata/dublin_core_qualifiers#Editor
"/>
        <rdf:value>Amy Friedlander</rdf:value>
      </dc:Contributor>
      <dc:Publisher>Corporation for National Research
Initiatives</dc:Publisher>
      <dc:Date>1998-01-05</dc:Date>
      <dc:Type>electronic journal</dc:Type>
      <dc:Subject>
        <rdf:Bag>
          <rdf:li>library use studies</rdf:li>
          <rdf:li>magazines and newspapers</rdf:li>
        </rdf:Bag>
      </dc:Subject>
      <dc:Format>text/html</dc:Format>
      <dc:Identifier>urn:issn:1082-9873</dc:Identifier>
      <dc:Relation rdf:parseType="Resource">
        <dcq:RelationType
          rdf:resource="http://purl.org/metadata/dublin_core_qualifiers#IsPart
Of"/>
        <rdf:value resource="http://www.dlib.org"/>
      </dc:Relation>
    </rdf:Description>
</rdf:RDF>
```

źródło: http://www.w3.org/2000/10/rdf-tests/RDF-Model-Syntax_1.0/ms_7.4_2.rdf

Przytoczony przykład oprócz podstawowej gramatyki rdf korzysta z przestrzeni nazw gramatyk dublin core - „dc” oraz ”dcq”.


http://www.dlib.org/dlib/may98/05contents.html
dc:Contributor
zasób anonimowy 1
rdf:value
Amy Friedlander
dcq:AgentType
dcq:Editor
dc:Date
1998-01-05
dc:Description
D-LIB magazine is a monthly compilation of contributed stories, commentary, and briefings.
dc:Format
text/html
dc:Identifier
urn:issn:1082-9873
dc:Publis
Corporation for National Research Initiatives
dc:Relation
zasób anonimowy 2
dcq:IsPartOf
dcq:RelationType
http://www.foo.com.au/cool.html
dc:Subj
zasób anonimowy 3
rdf:_1
library use studies
rdf:_2
magazines and newspapers
rdf:Type
rdf:Bag
dc:Title
DLIB Magazine - The Magazine for Digital Library Research - May 1998
dc:Type
electronic journal

Poniżej znajduje się zapytanie do przykładowego zbioru RDF zwracające imię i nazwisko osoby współtworzącej treść zasobu „*http://www.dlib.org/dlib/may98/05contents.html*” tj. współautora strony internetowej *Amy Friedlander*.

```
(RDFLiteralClass)(
  ((RDFPropertyClass)
    (((RDFBlankNodeClass)
      ((RDFProperty where uri = "dc:Contributor").node)
    ).properties where uri=”rdf:value”)
  ).node
).value
```

Zapytanie w pierwszej kolejności przeszukuje ekstensje predykatów w poszukiwaniu obiektu klasy *RDFPropertyClass* o wartości identyfikatora uri równej *"dc:Contributor"*, następnie węzeł związany ze zwróconym predykatem rzutuje na obiekt klasy *RDFBlankNodeClass* by przeszukać predykaty z niego wychodzące i zwrócić predykat o wartości URI równym *”rdf:value”*. Ostatecznie zapytanie rzutuje węzeł związany z otrzymanym predykatem na obiekt klasy *RDFLiteralClass* by zwrócić wartość pola *value* tego obiektu.

Przedstawiona konstrukcja zapytania niesie ze sobą **problem potrzeby rzutowania typów** węzłów biorących udział w przeszukiwaniu grafu RDF. Programista sbql budujący zapytania jest zmuszony **przewidywać typ** węzłów biorących udział w zapytaniu.

W praktycznym użyciu struktury RDF w module SBQL warto rozwiązać ten problem poprzez implementacje metod przeszukujących węzły grafu na które wskazuje rozpatrywany węzeł biorąc za argument przeszukiwania podaną wartość URI predykatu. Implementacja metod dla węzłów powinna zawierać użycie operatora ***instanceof***. Więcej informacji o proponowanych rozszerzeniach znaleźć można w rozdziale: 6.3 Możliwości rozbudowy.

Uwaga:

Autor niniejszej pracy podczas implementacji i testów filtra RDF napotkał na błędy systemu Odra związane z działaniem operatora ***instanceof*** a także w mechanizmie rzutowania typów. Aby przystąpić do implementacji i testowania proponowanych rozwiązań należy usunąć błędy wspomnianych mechanizmów języka SBQL w implementacji Odra. Więcej informacji o błędach implementacji systemu Odra można znaleźć w rozdziale: 6.2 Napotkane błędy i ograniczenia.

4.4.3 Zapytania wykorzystujące tranzytywne domknięcie

Struktura grafu skierowanego jaki tworzą zdania RDF sprawia, że przed programistą języka SBQL pojawia się możliwość wykorzystania mocy operatora tranzytywnego domknięcie tj. *close by* oraz *leaves by*.

Operator tranzytywnego domknięcia może znaleźć szerokie zastosowanie w implementacji metod przeszukujących węzły grafu z aktualnie rozpatrywanego zasobu o których mowa w punkcie 6.3 Możliwości rozbudowy.

Prostym przykładem zapytania wykorzystującego operator tranzytywnego domknięcie może być zapytanie:

podaj wszystkie wynalazki których warunkiem koniecznym powstania był wynalazek i praktyczne zastosowanie prądu elektrycznego

tj. podaj wszystkie zasoby na które wskazuje predykat *„pozwoliło opracować”* zaczynając od zasobu *„prąd elektryczny”*:

```
distinct(
  (RDFResource where uri=”prąd elektryczny”)
  close by
  ((RDFResourceClass)
    (property where uri=”pozwoliło opracować”).node
  )
)
```

Powyższe zapytanie zwraca kolekcję obiektów klasy *RDFResourceClass* bez powtórzeń które są związane z obiektem klasy *RDFResourceClass* o wartości pola *uri* równej *„prąd elektryczny”* za pomocą asocjacji realizowanej przez obiekt klasy *RDFPropertyClass* o wartości pola *uri* równej *„pozwoliło opracować”*.

Tranzytywne domknięcie może zostać użyte jako środek służący przeglądaniu grafu RDF w celu odnalezienia zasobów *składających się* na dany zasób. *Składanie się* w tym wypadku oznacza relacje reprezentowaną przez predykat który jest tranzytywny dla grafu. Ciekawym efektem może być rozszerzanie definicji relacji po klauzurze *close by* tak by relacja była była złożeniem dwóch lub więcej predykatów dających sensowny z punktu widzenia zastosowania zbiór zasobów RDF do których można *dotrzeć* dzięki tranzytywności tak zdefiniowanej relacji.

5 Implementacja

5.1 Biblioteka Jena 2

W niniejszej pracy została wykorzystana wiodącą biblioteka przetwarzania danych RDF - Jena 2.

Jena 2 jest projektem open-source, zaimplementowanym w języku Java, ogólnodostępnym na portalu SourceForge rozwijanej przez Brian'a McBride'a przy wsparciu firmy Hewlett-Packard.

Główną wartością Jena 2 jest API przeznaczone do przetwarzania grafu RDF, posiada implementację szeregu mechanizmów pozwalających na manipulację danymi RDF takimi jak:

- parser RDF/XML
- język zapytań RDQL
- modułu odczytu/zapisu dla notacji RDF/XML, N3, N-Triple *[15]*

Dane RDF przetwarzane przez API biblioteki mogą być przechowywane w pamięci operacyjnej jak również w wybranych relacyjnych systemach bazodanowych jako tabele trójek RDF.

Jena posiada API języka Java pozwalające na manipulację i tworzenie grafów RDF za pomocą zestawu klas Java reprezentujących grafy RDF, zasoby, predykaty, literały RDF itd. Graf RDF jest reprezentowany przez klasę *com.hp.hpl.jena.rdf.model.Model*, zdanie RDF przez klasę *com.hp.hpl.jena.rdf.model.Statement [3]*

Poniższy przykład obrazuje odczytanie danych RDF, serializowanych 3 metodami (*RDF/XML, N-TRIPLE, N3*) do obiektu klasy *com.hp.hpl.jena.rdf.model.Model* biblioteki Jena:

```
 // utworzenie pustego modelu
 Model model = ModelFactory.createDefaultModel();

 // użycie klasy FileManager w celu odczytania pliku
 InputStream in = FileManager.get().open( inputFileName );
if (in == null) {
    throw new IllegalArgumentException(
                                  "File: " + inputFileName + " not found");
}

// odczyt pliku rdf serializowanego metodą RDF/XML
model.read(in, null);

// odczyt pliku rdf serializowanego metodą  N-TRIPLE
model.read(in, "", "N-TRIPLE");

// odczyt pliku rdf serializowanego metodą  N3
model.read(in, "", "N3");
```

Wypisanie zdań RDF grafu można realizować za pomocą metody *listStatements()* obiektu klasy *Model.* Poniższy przykład wypisuje na konsole zdania RDF z podziałem na ich temat, predykat i obiekt.

```
// lista zdań Rdf w modelu
StmtIterator iter = model.listStatements();

// wypisanie predykatu, tematu i obiektu każdego zdania RDF
while (iter.hasNext()) {
    Statement stmt      = iter.nextStatement();
    Resource  subject   = stmt.getSubject();     // temat
    Property  predicate = stmt.getPredicate();   // predykat
    RDFNode   object    = stmt.getObject();      // obiekt

    System.out.print(subject.toString());
    System.out.print(" " + predicate.toString() + " ");
    if (object instanceof Resource) {
       System.out.print(object.toString());
    } else {
        // obiekt jest literałem
        System.out.print(" \"" + object.toString() + "\"");
    }

    System.out.println(" .");
}
```

Dodatkowo API biblioteki Jena oferuje szereg metod pozwalających na odpytanie modelu na podstawie

podanych tematów, predykatów i obiektów RDF.

Generyczną metodą odpytywania modelu jest metoda *ListStatements(Resource r, Property p, RDFNode o)* obiektu klasy *Model*. Metoda pozwala wyszukać zdania RDF zawierajace podany temat, predykat i obiekt zdania RDF, przy czym każdy z parametrów może przyjąć wartość *null* oznaczające dowolne dopasowanie parametru w porównaniu zapytania.

```
//Utwórz zasób „Stanisław Lem”
Resource lem = model.createResource(„writers.com/Stanislaw_Lem”);
//Utwórz zasób „Wizja Lokalna”
Resource wizja_lokalna = model.createResource(„books.com/Wizja lokalna”);
//Utwórz predykat oznaczający autora
Property author = model.createProperty(„dc:Author”);

// Zwróć zdanie mówiące, że Stanisław Lem jest autorem książki „Wizja Lokalna”
model.listStatements(lem,author,wizja_lokalna);

// Zwróć zdania mówiące cokolwiek o Stanisławie Lemie jako temacie i książce
„Wizja lokalna” jako obiekcie zdania
model.listStatements(lem,null,wizja_lokalna);

// Zwróć wszystkie zdania mówiące o Staniesławie Lemie jako temacie
model.listStatements(lem,null,null);

// Zwróć wszystkie zdania w których „autor” jest predykatem zdania
model.listStatements(null,author,null);
```

5.2 Rozwiązanie budowy i reprezentacji grafu RDF w składzie Odra

5.2.1 Reprezentacja grafu RDF w składzie systemu Odra

Jedno z początkowo rozpatrywanych założeń budowania struktury grafu RDF w składzie systemu Odra wskazywało na budowę obiektów SBQL, specyficznej dla węzła klasy, reprezentujących węzły grafu RDF wraz z asocjacjami pomiędzy obiektami reprezentującymi predykat RDF.

Podejście to jest bliskie rozwiązaniu przyjętemu w istniejącym filtrze XML dla systemu Odra, wykazuje jednak szereg wad użyteczności w przetwarzaniu danych RDF przez aplikacje napisane w języku sbql:

1. Ze względu na specyfikę RDF i jego *postulat otwartości*, należy przyjąć, że dany temat RDF, węzeł grafu RDF, może w dowolnym momencie zmienić zestaw predykatów, co wymagałoby zmiany typu obiektu SBQL.
2. Dla danych RDF wymagany byłby dodatkowy zbiór RDFS definiujący typy obiektów RDF, wypełniający metadane Odra. W praktyce należy się spodziewać, że produkcyjne dane RDF mogą nie posiadać zdefiniowanej dziedziny w postaci odpowiadającego grafu na poziomie RDFS.
3. Specyfikacja RDF/RDFS dopuszcza aby zasób RDF posiadał zestaw wielu typów, zmiennych w

czasie.

4. Dla skutecznego przetwarzania i odpytywania danych RDF identyfikowanych przez URI programista sbql powinien otrzymać możliwość przetwarzania łańcucha znaków składającego się na URI oraz ewentualną ich modyfikację (np. użycie skróconej wartości URI za pomocą wcześniej zdefiniowanych przestrzeni nazw)
5. Asocjacja obiektów sbql reprezentujących węzły grafu RDF powinna być możliwa do identyfikacji za pomocą URI wraz ze wszystkimi dozwolonymi dla URI, a zabronionymi jako nazwa pola klasy języka SBQL znakami.

Na podstawie wyżej wymienionych przesłanek autor pracy zdecydował, w odróżnieniu od metody użytej w filtrze XML, zastosować predefiniowaną strukturę danych dla budowy grafu RDF.

5.2.2 Proces budowy elementów grafu RDF

Budowa grafu RDF w składzie Odra opiera się na wywoływaniu przygotowanych procedur modułu bibliotecznego RDF.sbql.

Wywołanie importu dla danego modułu powoduje skonstruowanie pod modułu RDF oraz skompilowanie kodu RDF.sbql. Procedury zawarte w RDF.sbql są następnie wywoływane przez kod importera za pomocą klasy odra.*filters.RDF.common.SBQLHelper* realizującą:

- wywołania kodu sbql, w tym budowa drzewa AST, kontrola typów oraz optymalizacja zapytań
- linkowanie i kompilacje modułu dla którego został wywołany kod SBQL (jeżeli potrzebne)

Etap wywołania kodu RDF.sbql poprzedza budowa obiektu klasy *com.hp.hpl.jena.rdf.model.Model* zawartej w bibliotece *Jena*, na podstawie kodu RDF wczytanego według notacji podanej jako parametr importu. Domyślna notacja importu danych RDF to RDF/XML.

Importer przechodzi wszystkie predykaty zawarte w modelu RDF i w zależności od typu węzłów RDF połączonych przez aktualnie przetwarzany predykat wywołuje metodę odpowiadającą zdaniu RDF tworzącą trójkę RDF w składzie Odra.

Dodatkowo, dla każdego przedrostka w przestrzeni nazw modelu RDF, wypełnia przestrzenie nazw pod modułu RDF za pomocą procedur SBQL.

5.3 Obsługa identyfikacji zasobów anonimowych

Jednym z wyzwań stojącym przed programistą aplikacji przetwarzającej RDF jest scalanie grafów RDF.

Obok procesu scalania danych za pomocą przeszukiwania grafu pod kątem identyczności URI dodatkowo należy zarządzać identyfikacją zasobów anonimowych.

Specyfikacja RDF nie definiuje sposobu numeracji zasobów anonimowych. Przyjętą konwencją jest

użycie *identyfikatora zasobu anonimowego* - lokalnego identyfikatora współistniejącego obok URI. Podczas scalania grafów RDF identyfikatory zasobów anonimowych powinny zostać przetworzone pod kątem zapewnienia jednoznacznej identyfikacji.

Aby obsłużyć taki przypadek programista SBQL powinien przeprowadzić import drugiego grafu RDF do oddzielnego modułu a następnie scalić grafy RDF używając warstwy aplikacyjnej SBQL.

W module bibliotecznym RDF.sbql identyfikator zasobu anonimowego zapisywany jest pod polem *rdfNodeID* typu string klasy *RDFBlankNodeClass*. Wartości identyfikatorów przepisywane są bezpośrednio z rozpatrywanego modelu RDF i zależą od użytej notacji. Zazwyczaj w notacji RDF/XML identyfikatory przyjmują wartości kolejnych liczb naturalnych zaczynając od 1.

Uwaga: istnieje możliwość złamania niepowtarzalności numeracji zasobów anonimowych w przypadku importu do modułu sbql więcej niż jednego grafu RDF.

W przypadku, gdy drugi graf RDF zostanie zaimportowany do modułu już wypełnionymi danymi pierwszego grafu RDF i importer napotka zasób anonimowy o identyfikatorze równym zasobowi już istniejącemu, to potraktuje nowy zasób anonimowy jako poprzedni, już istniejący, i połączy krawędziami grafu RDF istniejący zasób anonimowy do wierzchołków grafu, do których powinien zostać połączony nowy zasób anonimowy.

5.4 Obsługa formy serializacji RDF

Importer obsługuje dane RDF zapisane za pomocą form: RDF/XML, dodatkowo N-TRIPLE, N3. Domyślną formą serializacji jest RDF/XML. Użytkownik może skorzystać z formy innej niż domyślna podając odpowiedni parametr importu.

5.5 Środowisko implementacyjne

Implementacja filtru RDF została wykonana w środowisku programistycznym Eclipse 3.3.0.

Ostatnia, testowana wersja importera została zintegrowana z kodem źródłowym Odra pobranej z repozytorium svn:

```
https://dev.kis.p.lodz.pl:8443/svn/egovbus/ODRA2/trunk
```

o numerze rewizji: 2659 (ostatnia zmiana kodu repozytorium: 11.12.2008)

Użyta wersja Javy to 1.6.0_06.

Podczas testów i budowy modułu bibliotecznego RDF.sbql zostało użyte środowisko IDE systemu Odra oparte o edytor jEdit - ODRA-IDE w wersji 2007.09.05-01. Efektywne wykorzystanie ODRA-IDE polegało

na współpracy z bazą Odra odpaloną w trybie debug'u w systemie Eclipse - ODRA-IDE pomija część komunikatów systemu Odra które można prześledzić jedynie na konsoli Eclipse oraz w samym debuggerze.

W pracy została wykorzystana biblioteka Jena w wersji 2.5.4.

6 Dalsza rozbudowa i zastosowanie

6.1 Ocena przyjętego zastosowania

Zaletą przyjętego rozwiązania jest jejgo otwarty charakter pozwalający programiście sbql na budowę warstw aplikacyjnych odpowiadających warstwom charakterystycznym dla Semantic Web. Autor pracy przyjął założenie budowy dodatkowych warstw aplikacyjnych pliku bibliotecznego rdf sbql oraz rozszerzenia zaproponowanej struktury o metody przeszukujące podgraf z uwzględnieniem zwracanego typu wierzchołka oraz typów wierzchołków biorących udział w zapytaniu. Część implementacji po stronie sbql możliwa będzie do zrealizowania jedynie po usunięciu błędów i niedostatków implementacyjnych systemu Odra.

Za wadę można uznać konieczność utrzymania pliku bibliotecznego RDF.sbql w strukturze plików. Rozwiązaniem mogącym dostarczać praktycznych problemów użycia zastosowanego rozwiązania jest konieczność przewidywania typów węzłów w zapytaniach do zbiorów RDF. Problem ten jednak należy rozwiązać rozszerzając strukturę RDF o dodatkowe metody pomocnicze w odpytywaniu danych RDF jak również kolejne warstwy aplikacyjne SBQL.

Kod filtru RDF warto poddać modyfikacjom mającym na celu integracje z istniejącymi standardami kodowania systemu Odra, np. warto rozważyć zmianę implementacji importera - zamiast wywoływać kod sbql tworzący graf RDF *zdanie po zdaniu* w klasie *SBQLHelper* można wywoływać procedury na poziomie konstruowania drzewa AST.

6.2 Napotkane błędy i ograniczenia

6.2.1 System Odra

6.2.1.1 Bag jako jedyna dostępna kolekcja

Ograniczeniem dla przyszłego rozszerzenia struktury danych RDF w systemie Odra jest aktualnie jedynie dostępna kolekcja *Bag*.

Ograniczenie było również jednym z czynników zaniechania prób implementacji bezpośredniego mapowania kolekcji RDF na kolekcje systemu Odra.

6.2.1.2 Błąd powtórnej kompilacji modułu

System Odra w implementacji aktualnej dla niniejszej pracy obarczony jest błędem nie pozwalającym programiście aplikacji SBQL na ponowną kompilację danego modułu.

Podczas próby ponownej kompilajci system zgłąsza wyjątek Java:

```
odra.sbql.builder.OrganizerException: Module field (procedure) '<nazwa
procedury>' is not compatible with an older version at
odra.sbql.builder.ModuleOrganizer.createProcedure(ModuleOrganizer.java:157)
```

Występowanie błędu wymusza na programiście SBQL ponowne budowanie instancji serwera bazy Odra dla każdej próby kompilacji. Dodatkowo filtr danych RDF uruchomiony ponownie dla danego modułu podejmuje próbę kompilacji plików źródłowych biblioteki RDF.sbql co może powodować komunikaty błędów kompilacji.

6.2.1.3 Brak mechanizmu rzutowanie typów w zapytaniach

Jednym z głównych ograniczeń implementacji systemu Odra nie pozwalającemu w pełni wykorzystać teoretyczne założeniom języka SBQL jest brak mechanizmu rzutowania w zapytaniach sbql.

Zapytania takie są typowe w przetwarzaniu zbiorów RDF w zaproponowanej strukturze pozbawionej implementacji przeszukiwania węzłów związanych z danym węzłem.

Przykładowe zapytanie SBQL do zbioru RDF przeszukujące graf w poszukiwaniu uri zasobu na podstawie URI predykatu wskazującego na szukany zasób:

```
((RDFResourceClass)((RDFProperty where uri = "dc:Format").node)).uri
```

Dla w.w. zapytania system zgłasza wyjątek o wartości *„Procedure reference expected”*.

6.2.1.4 Błędne działanie operatora instanceof

Podczas implementacji i testów filtru RDF autor pracy odkrył błąd w działaniu operatora *instanceof*. System Odra zwraca prawidłowy wynik TRUE gdy obie strony porównania posiadają faktycznie ten sam typ. W sytuacji gdy obie strony porównania operatora *instanceof* posiadają typ różny, system Odra nie zwraca wartości FALSE lecz zgłasza wyjątek Java związany z implementacją Odra.

6.2.2 Błędy RDF

Podczas testów filtru RDF na danych pochodzących z oficjalnych publikacji traktujących o technologi RDF, autor pracy napotkał na błędy syntaktyczne danych testowych. Błędy wynikały ze zmian specyfikacji syntaktyki RDF w przeciągu ostatnich dziesięciu lat od jej ogłoszenia.

6.3 Możliwości rozbudowy

6.3.1 Kod Odra

Jednym z funkcjonalnych rozszerzeń filtru RDF dla Odry jest możliwość mapowania kolekcji RDF na kolekcje Odra. Jakkolwiek w czasie implementacji filtru RDF system Odra obsługiwał jedynie kolekcje typu Bag, autor proponuje implementacje rozszerzenia jako opcję i pozostawienie odwzorowania kolekcji RDF jako zdania RDF w systemie Odra jako podstawowy tryby importu danych.

Autor dostrzega możliwość zmiany implementacji wywoływania procedur tworzenia grafu RDF za pomocą budowy drzewa wywołań AST mającą jedynie wywołać odpowiednie metody SBQL tworzące graf RDF.

6.3.2 Aplikacje SBQL

6.3.2.1 Warstwa obsługi danych RDF

Zaproponowany przez autora filtr RDF został zaprojektowany z myślą o zbudowaniu zrębu programistycznego dla programisty SBQL przetwarzającego dane RDF. Naturalnym rozszerzeniem pliku bibliotecznego RDF.sbql są moduły SBQL implementujące kolejne warstwy abstrakcji zrębu programistycznego.

Autor dostrzega następujące aspekty warte realizacji w kolejnych warstwach biblioteki:

- **Obsługa nadawania identyfikatorów dla zasobów anonimowych po stronie modułu bibliotecznego SBQL –** w obecnym kształcie biblioteki nadawanie identyfikatorów podczas tworzenia zasobów anonimowych jest niemożliwe (procedura SBQL: *createBlankNode*), warto jednak rozważyć metodę nadającą identyfikatory dla zasobów anonimowych uruchamianą w momencie następującym po imporcie całego grafu; taka procedura mogłaby brać udział również w procesie nadawania zasobów anonimowych podczas integracji grafów RDF
- **klasa reprezentująca URI**
- **klasa reprezentująca zdanie RDF**
- **rozszerzenie klasy reprezentującej przestrzeń nazw** o metody związane z przetwarzaniem łańcuchów znaków w celu porównania
- **zasłonięcie gramatyki RDF przez obiekty SBQL –** opcjonalne rozszerzenie klas biblioteki RDF.sbql i dodanie nowych klas w celu wyeliminowania predykatów gramatyki podstawowej RDF, tj.:
  - *rdf:XMLLiteral*

- *rdf:Property*
- *rdf:type*
- *rdf:value*
- *rdf:Statement (reifikacja)*
- *rdf:subject (reifikacja)*
- *rdf:predicate (reifikacja)*
- *rdf:object (reifikacja)*
- *rdf:Bag (kolekcje)*
- *rdf:Seq (kolekcje)*
- *rdf:Alt (kolekcje)*
- *rdf:List (kolekcje)*
- *rdf:first (kolekcje)*
- *rdf:rest (kolekcje)*
- *rdf:nil (kolekcje)*

Podstawową gramatykę RDF wraz z RDFS specyfikuje dokument: *http://www.w3.org/TR/rdf-schema/*

podzbiorem takiej implementacji jest:

- **zasłonięcie kolekcji RDF przez kolekcje SBQL** – gdzie zasoby identyfikowane jako kolekcje zostałyby reprezentowane w składzie Odra jako kolekcje języka SBQL, dotyczy: *Bag*, *Seq*, kolekcja *Alt* nie mająca swojego odwzorowania w języku SBQL

W kolejnych warstwach aplikacyjnych można rozważyć:

- **zasłonięcie gramatyki RDFS przez obiekty SBQL** wraz z implementacją reguł RDFS (*Etailment Rules*) tj. predykatów:
  - *rdfs:Resource*
  - *rdfs:Class*
  - *rdfs:Literal*
  - *rdfs:Datatype*
  - *rdfs:range*
  - *rdfs:domain*
  - *rdfs:subClassOf*
  - *rdfs:subPropertyOf*
  - *rdfs:label*
  - *rdfs:comment*
  - *rdfs:ContainerMembershipProperty*

- *rdfs:member*
- *rdfs:seeAlso*
- *rdfs:isDefinedBy*

- **zasłonięcie gramatyki OWL przez obiekty SBQL**

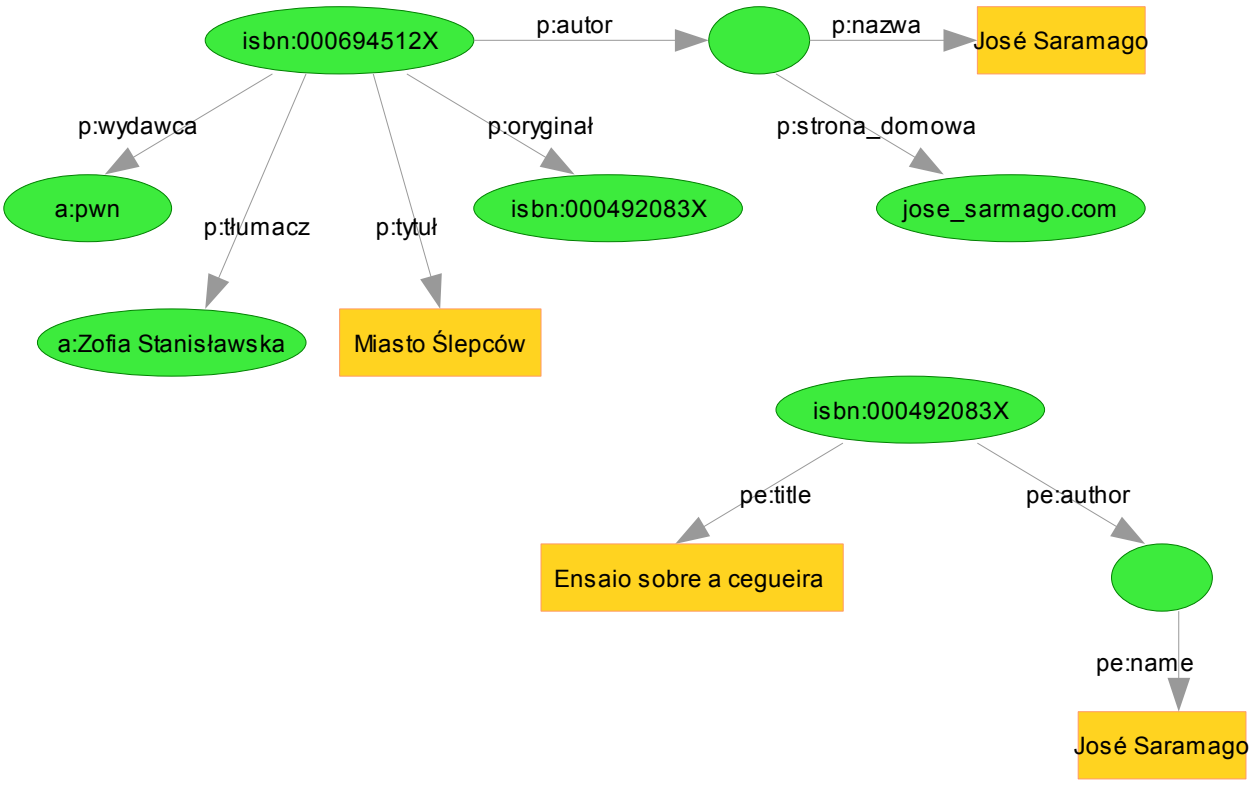
6.3.2.2 Warstwa integracji grafów RDF

Warstwy aplikacji SBQL powinny realizować założenia twórców specyfikacji RDFS ora OWL. W tym celu warto rozważyć implementacje mechanizmów integracji grafów RDF realizując zadania:

1. **Identyfikacja identycznych zasobów wg URI**

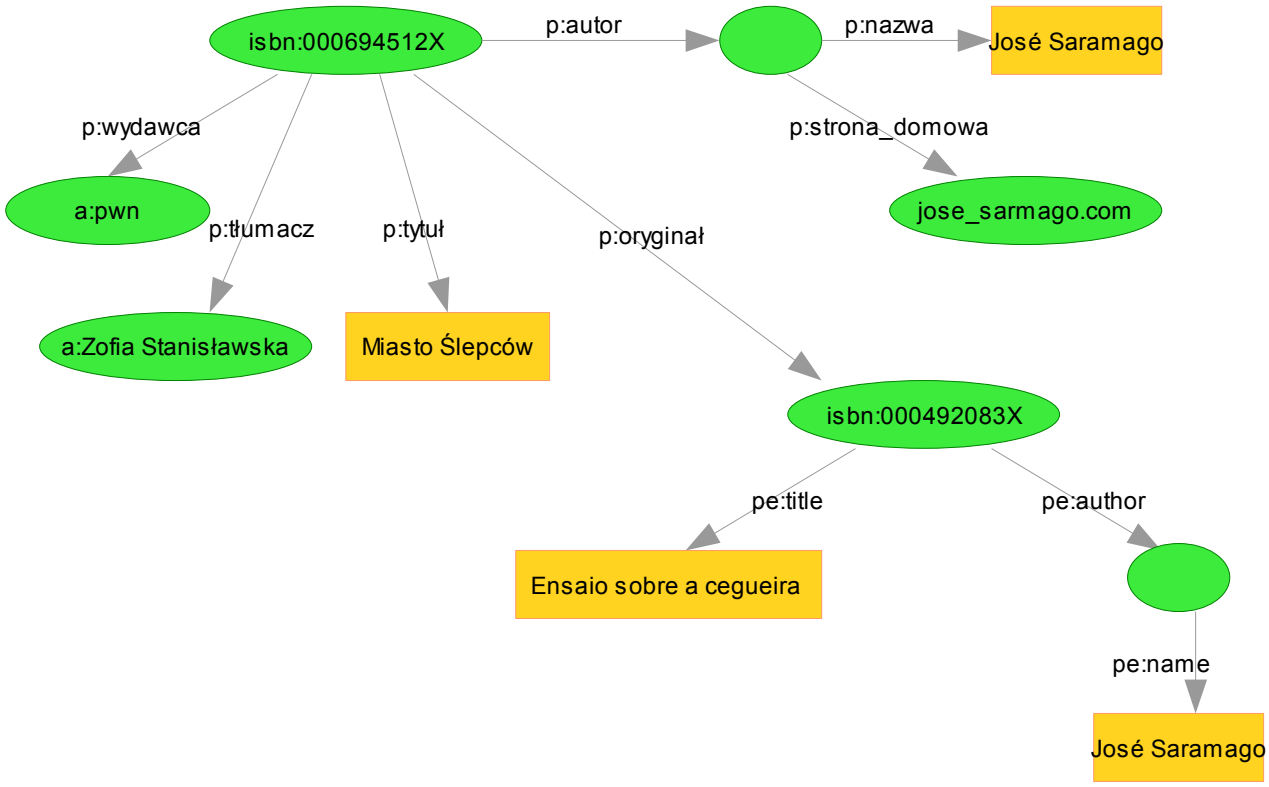
   Integracja grafów RDF w podstawowej formie bazuje na identyfikacji (na podstawie URI) zasobów występujących jednocześnie w grafach integrowanych.

   Poniższy przykład obrazuje 2 grafy RDF: pierwszy opisuje książkę o numerze isbn:000694512X, drugi pozycje isbn:000492083X:



   Obie pozycje pozycje opisują książkę tego samego autora – *Jose Saramago*, przy czym pierwszy (czytając od lewej) graf mówi o wydaniu polskim o tytule *Miasto Ślepców*, drugi zaś o oryginalnym wydaniu portugalskim.

Opis polskiego wydania posiada zdanie RDF mówiące o numerze isbn oryginału książki. Ponieważ URI reprezentujące oryginał książki odpowiada URI zasobu wskazującego oryginalne źródło polskiego tłumaczenia na numer ISBN aplikacja przetwarzająca graf RDF powinna implementować możliwość połączenia grafów za pomocą predykatu wskazującego na tożsame zasoby:



2. **Identyfikacja semantycznych relacji między zasobami i predykatami**
   Kolejnym stopniem integracji danych RDF jest łączenie grafów na podstawie metadanych opisujących schemat RDFS oraz metadanych semantycznych opisanych w grafach OWL związanych z wiedzą zapisaną w rozpatrywanych do integracji grafach RDF.

   Aplikacja przetwarzająca przykładowe grafy RDF powinna posiadać mechanizmy modyfikacji grafów integrowanych bazujące na regułach RDFS oraz OWL zapisanych jako zestaw słownikowych predykatów RDFS i OWL.

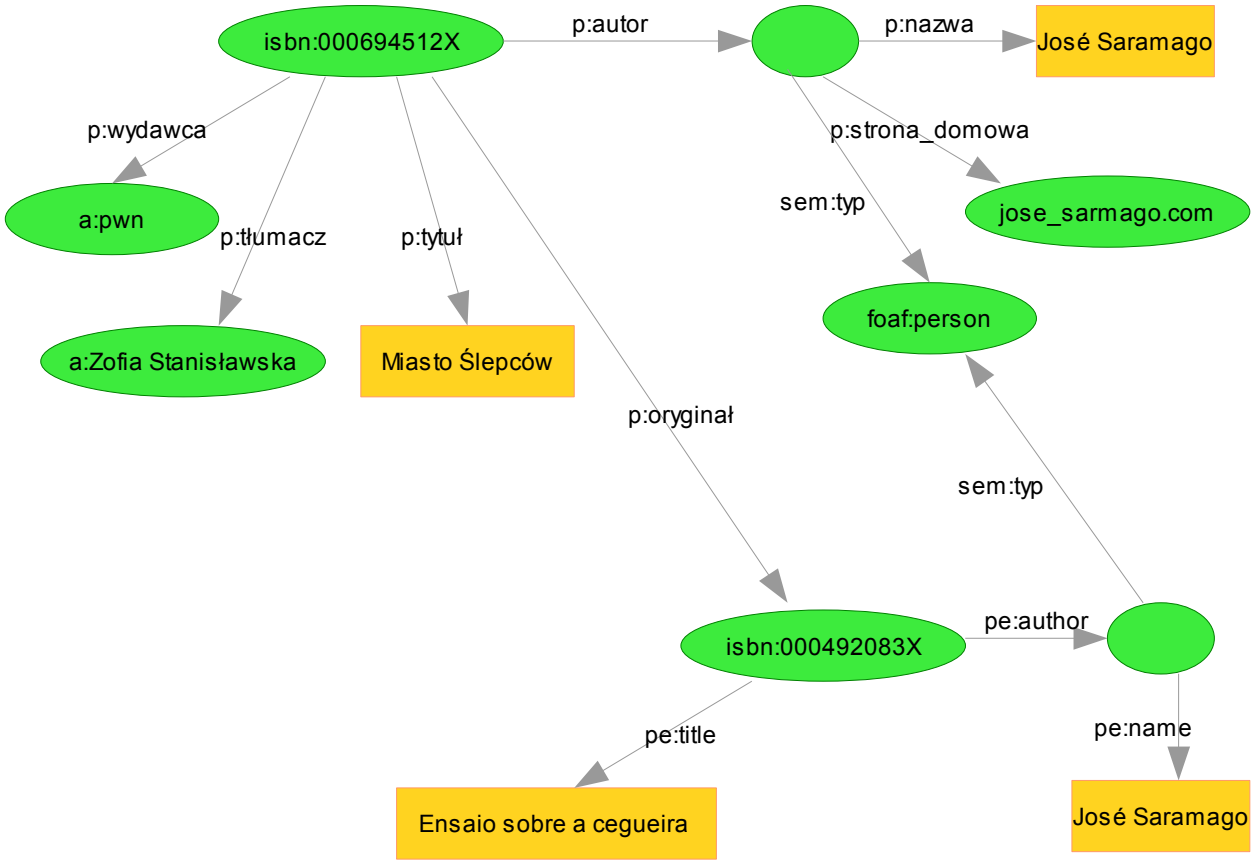
   Poniższy przykład zdania RDF grafu OWL mówi, że predykat *pe:author* jest ekwiwalentem

predykatu *p:autor*.

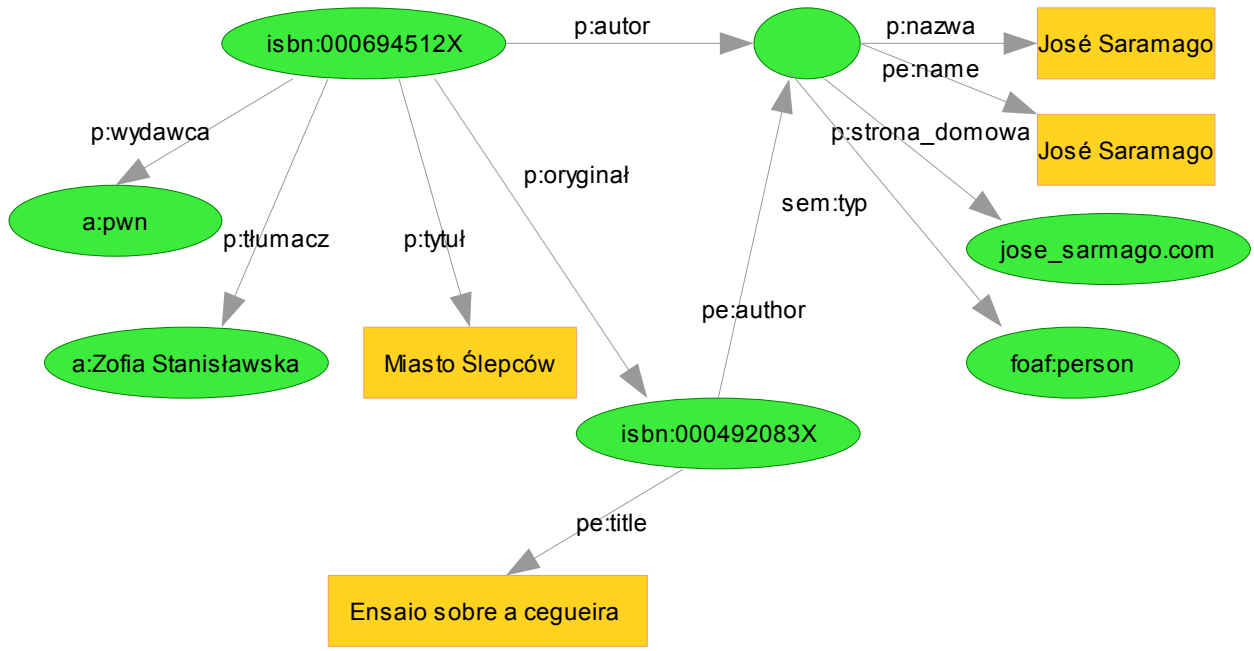
W konsekwencji obsługi słownikowego predykatu *owl:equivalentProperty* aplikacja przetwarzająca graf RDF powinna przetworzyć grafy zgodnie z semantycznym znaczeniem predykatu.
W tym rozpatrywanym przykładzie wiedza o relacji bycia ekwiwalentnym między predykatem *pe:author* a *p:autor* wprowadza możliwość identyfikacji zasobów na które wskazują *pe:author* i *p:autor* jako tożsamy w sensie semantycznym (np. mowa o tym samym autorze *Ensaio sobre a cegueira* tej samej książki nie zwracając uwagę na fakt wydania w innym języku)
Fakt tożsamości tematu może zostać obliczony na podstawie rachunku predykatów OWL.

3. Wnioskowanie o zasobach i predykatach

Kolejnym mechanizmem związanym z integracją danych RDF jest wnioskowanie o występowaniu dodatkowych zasobów nie wymienionych explicite w grafach źródłowych. Wnioskowanie można oprzeć, podobnie jak w pkt 2., o metadane semantyczne opisujące podzbiór rozpatrywanych zasobów.
W przytoczonym przykładzie związanie predykatu *p:autor* i *pe:author* z semantycznym pojęciem *autor* oraz dodatkowe zdanie RDF (np. pochodzące z ustalonej gramatyki) mówiące o typie zasobów identyfikowanych jako autor, pozwoliło dodać zdanie RDF o zasobach na które wskazują mówiące, że jest to typ *foaf:person:*



Kolejnym *wnioskiem* bazującym na zdaniu RDF *pe:author owl:equivalentProperty p:autor* mówiącego o semantycznej równoważności predykatów *pe:author* i *p:autor* z pkt. 2. jest integracja zasobów anonimowych na które wskazywały predykaty oznaczające *autora*:



Realizacja wymienionych zadań powinna być poprzedzona rozwiązaniem problemu **identyfikacji zasobów anonimowych.**

6.3.2.3 Składowane metody i procedury odpytujące graf RDF

Tworząc środowisko programistyczne w systemie Odra dla programisty przetwarzającego dane RDF warto przygotować zestaw standardowych procedur odpytujących dane RDF.

Formy zapytań powinny bazować na

- zdaniach RDF – generyczna metoda, przy użyciu klasy SBQL reprezentującej zdanie RDF, podając przykładowe zdanie zostanie zwrócony zbiór zdań odpowiadających.
- predykatach – predykat może zostać użyty jako argument w zapytaniu o zasoby lub literały na które wskazuje lub zasoby które są dla niego tematem zdania RDF
- zasobach

Dodatkowo, pożądaną cechą zapytań do grafów RDF jest przeszukiwanie podgrafu (np. na który wskazuje predykat związany z danym zasobem). Implementacja takiego zapytania dla programisty SBQL powinna przynieść ułatwienie w przypadku zapytania na podstawie danego predykatu o zasoby co do których nie ma pewności, czy jest on bezpośrednim następnikiem predykatu, na bazie którego jest formułowane zapytanie czy też występuje pośredni zasób, najczęściej zasób anonimowy. Opcjonalnie można poddać implementacji zapytania do podgrafu o zasoby związane z danym zasobem w *n-tym* kroku (tzn. następującym po kolejnej warstwie predykatów).
Szczególnym przypadkiem, wartym obsłużenia przez biblioteki RDF w systemie Odra, jest zasób anonimowy, używany najczęściej jako agregator predykatów prowadzący do literałów i zasobów, którego fakt użycia zamiast zasobu identyfikowane URI zależy od rozpatrywanej gramatyki RDF.

Kolejnym aspektem predefiniowanych zapytań jest analiza leksykalna URI zasobów. Implementacja zapytań w tym przypadku powinna pozwolić programiście języka SBQL na użycie przygotowanych zapytań o zasoby identyfikowane np. z daną domeną lub poddomeną URI (np. *podaj zasoby z domeny odra.pjwstk.edu.pl*).

Warto rozważyć zapytania składowane do grafów RDF operujące na warstwie obsługi danych RDF dla której kolekcje RDF są *przezroczyste* tj. niejawnie zamieniane na kolekcje SBQL.

6.3.2.4 Klasy-wrappery gramatyk

Dla standardowych gramatyk RDF warto opracować statyczne klasy RDF reprezentujące te gramatyki.

Predefiniowany zestaw predykatów danej gramatyki reprezentowany byłby jako statyczne pola używane przez programistę SBQL jak również przez inne moduły biblioteczne związane z modułem RDF.

Szczególnym przypadkiem takich gramatyk bibliotecznych są gramatyki OWL oraz RDFS które mogą być współdzielone przez mechanizmy integracji i wnioskowania o integracji zasobów opisanych w punkcie 6.3.2.2 Warstwa integracji grafów RDF

6.3.2.5 Rozwiązanie problemu rzutowania typów w zapytaniach do grafu RDF

Użycie predefiniowanej struktury dla grafu RDF, gdzie węzły grafu reprezentowane są przez obiekty wyspecyfikowanych klas, niesie ze sobą konieczność używania podczas formułowania zapytań rzutowania typów. Ma to miejsce szczególnie podczas obsługi w podzapytaniu obiektu zdania RDF czyli węzła grafu RDF na który wskazuje predykat.

Węzeł RDF związany z predykatem może być obiektem klas:

- *RDFLiteralClass*
- *RDFBlankNodeClass*
- *RDFResourceClass*

Klasy te dziedziczą po klasie *RDFNodeClass*.

W podzapytaniu obiekt zdania RDF zwracany jest jako obiekt klasy SBQL *RDFNodeClass.*

Każde zapytanie o obiekt zdania RDF musi być rzutowane na typ w przypadku dalszego przetwarzania, np.:

```
(RDFResourceClass)
   (property where uri="<wartość uri>").node
```

Tworząc zapytanie należy założyć, że programista SBQL nie jest w stanie przewidzieć typ SBQL obiektu RDF na który wskazuje predykat mimo że w niektórych przypadkach może wnioskować o typie obiektu zdań RDF danego predykatu na podstawie definicji gramatyki RDF.

Sytuacja ta jest spowodowana założeniem o odwzorowaniu typu węzła RDF w typach języka SBQL. Co więcej klasy dziedziczące z klasy *RDFNodeClass* posiadają różne atrybuty, np. *RDFBlankNodeClass* nie posiada w swojej strukturze atrybutu *uri* a obiekty tej klasy identyfikowane są za pomocą atrybutu o nazwie *rdfNodeID.*

Autor pracy proponuje następujące grupy rozwiązań:

1. Spłaszczenie struktury SBQL dla danych RDF
   - Zdefiniowanie w strukturze klas SBQL takich samych atrybutów dla każdej klasy
   - Zdefiniowanie jednej klasy dla wszelkich elementów grafu RDF
2. Opracowanie metod zwracających obiekt predykatu ze względu na typ, np.:

```
(property where uri="<wartość uri>").getResource
```

W takim przypadku obsługa typu obiektu zostałaby nadal w gestii programisty języka SBQL, zmniejszeniu uległby jedynie stopień komplikacji zapytań.

3. Korzystanie z zestawu zapytań składowanych dla grafów RDF (6.3.2.3 Składowane metody i procedury odpytujące graf RDF)

6.3.3 Rozszerzenie biblioteki Jena o persystencję RDF w bazie SBQL

Autor pracy dostrzega szansę dla technologi SBQL we włączeniu silnika obiektowej bazy danych Odra jako **warstwę persystencji** oraz warstwę **odpytywania danych** *[5]*.

Rozwiązanie rozszerzałoby formułowania zapytań do danych RDF o zapytania w języku SBQL obok już istniejącego języka zapytań SPARQL.

Zaletą takiego podejścia jest wykorzystanie rozwijanej biblioteki Jena i jej funkcjonalności w aplikacjach opartych o technologie Java przy jednoczesnym wykorzystaniu siły algorytmicznej zapytań SBQL w odpytywaniu danych opartych o strukturę grafu jakimi są dane RDF.

Dla samego środowiska Stack Based Approach włączenie systemu Odra do biblioteki Jena stwarza okazję do ujawnienia szerszym społecznościom Semantic Web oraz programistom języka Java możliwości technologi języka SBQL.

7 Podsumowanie

Zaproponowany filtr wraz z podstawowym plikiem bibliotecznym RDF.sbql jest zaczątkiem środowiska programistycznego przeznaczonego do przetwarzania danych RDF w systemie Odra. Praktyczna wartość tego środowiska w dużej mierze zależeć będzie od zbudowania kompletnego zrębu programistycznego, bibliotek sbql poprzez rozszerzenie kodu bibliotecznego RDF.sbql o nowe klasy i mechanizmy charakterystyczne dla danych RDF.

W aktualnym stanie programista języka SBQL ma możliwość załadowania danych zapisanych w formacie RDF, a następnie przetwarzania wg przygotowanej struktury grafu skierowanego. Elementy danych RDF posiadają swoje odpowiedniki w obiektach przygotowanych klas języka SBQL

Autor zwraca uwagę na ograniczenia związane z przyjętym założeniem, potrzebę obsługi typów podczas formułowania zapytań do grafu, oraz z aktualną implementacją systemu Odra.

Wyzwaniem i jednocześnie szansą promocji koncepcji SBA jest przygotowanie przykładów użycia zrębu z wykorzystaniem specyfiki systemu opartego na podejściu stosowym. W szczególnym przypadku godnym zainteresowania jest opracowanie zapytań do grafu RDF z wykorzystaniem operatorów tranzytywnego domknięcia oraz równań stałopunktowych, dających możliwość wykorzystania mocy algorytmicznej w przetwarzaniu danych o strukturze grafu w stopniu przewyższającym możliwości systemów SQL. Zestawienie tak przygotowanych zapytań z możliwościami systemów produkcyjnych SQL może stanowić doskonałe źródło promocji koncepcji SBA w środowiskach skupionych wokół aplikacji przetwarzających dane RDF oraz szerzej, środowiskach skupionych wokół idei Semantic Web.

Bibliografia

[1] Kazimierz Subieta. *Teoria i konstrukcja obiektowych języków zapytań*. Wydawnictwo PJWSTK, Warszawa 2004, ISBN 83-89244-29-2

[2] Tim Berners-Lee *Managing and Enhancing Information: Cultures and Conflicts,* 2004-11-15 http://www.w3.org/2004/Talks/1115-asis-tbl/

[3] Shelley Powers *Practical RDF* O'Reilly 2003, ISBN 0-596-00263-7

[4] Kazimierz Subieta *Słownik terminów z zakresu obiektowości.* Akademicka Oficyna Wydawnicza PLJ, Warszawa 1999, ISBN 83-7101-407-4

[5] Kevin Wilkinson, Craig Sayers, Harumi Kuno, Dave Reynolds (Hewlett Packard Laboratories) *Efficient RDF Storage and Retrieval in Jena2*

[6] Ljiljana Stojanovic, Juergen Schneider, Alexander Maedche, Susanne Libischer, Rudi Studer, Thomas Lumpp, Andreas Abecker, Gerd Breiter, John Dinger *The Role of Ontologies in Autonomic Computing Systems*IBM Jurnal 2004/03

[7] Frank Manola, Eric Miller, W3C *RDF Primer* 2004 http://www.w3.org/TR/REC-rdf-syntax/

[8] Tim Bray, Dave Hollander , Andrew Layman, Richard Tobin *Namespaces in XML 1.0 (Second Edition)* 2006 http://www.w3.org/TR/REC-xml-names/

[9] Web Ontology Working Group, W3C *OWL Web Ontology Language Overview* 2004 http://www.w3.org/TR/owl-features/

[10] Graham Klyne, Jeremy J. Carroll, *W3C Resource Description Framework (RDF): Concepts and Abstract Syntax* 2004 http://www.w3.org/TR/2004/REC-rdf-concepts-20040210

[11] Patrick Hayes, W3C *RDF Semantics* 2004 http://www.w3.org/TR/rdf-mt/

[12] Dave Beckett, W3C *RDF/XML Syntax Specification* 2004, http://www.w3.org/TR/rdf-syntax-grammar/

[13] Eric Prud'hommeaux, Andy Seaborne, W3C *SPARQL Query Language for RDF* 2008 http://www.w3.org/TR/rdf-sparql-query/

[14] Ivan Herman, W3C *Introduction to the Semantic Web* Bangalore 2007 http://www.w3.org/2007/Talks/0221-Bangalore-IH/

[15] Brian McBride, HP *Jena: Implementing the RDF Model and Syntax Specification* http://www.hpl.hp.com/personal/bwm/papers/20001221-paper/

[16] Tim Berners-Lee, James Hendler, Ora Lassila *The Semantic Web* Scientific American May 2001

[17] Tim Berners-Lee, James Hendler *Managing and Enhancing Information: Cultures and Conflicts* World Wide Web Consortium Talks 2004-11-15

Dodatek 1. - Importer RDF dla Odra, instrukcja

Import danych z linii poleceń CLI

Importer danych RDF zaimplementowany jest zgodnie z przyjętą konwencją implementując interfejs *odra.filters.DataImporter.* Importer zarejestrowany jest jako plugin w *odra.system.Config* pod nazwią *RDFImporter*

Struktura wywołania pluginu ma postać:

```
load "<zasób>" using <nazwa plugin'u>
```

lub:

```
load "<zasób>" using <nazwa plugin'u>("parametry")
```

zasób – ścieżka do pliku rdf

nazwa plugin'u – nazwa pluginu dla komendy *load.* Dla filtru RDF aktualnie zaimplementowany jest jeden plugin o nazwie *RDFImporter.*

parametry – lista parametrów rozpoznawanych przez importer rozdzielonych znakami:

[spacja] „,”, „;”, \n, \t, \r, \f

Parametry RDFImporter

Aktualny stan implementacji przewiduje parametry formy uri z jakimi zostaną impotowane zasoby i predkaty rdf do składu systemu Odra:

FullURI – uri identyfikujące zasoby oraz predykaty rdf zapisywane będą w pełniej formie, pominięcie tego parametru w wywołaniu procedury importu spowoduje import danych rdf z polami uri w skróconej formie tzw. *Qname*

Importer dostarcza również możliwość eksperymentalnego importu danych rdf w formie serializacji innej niż RDF/XML:

N3 – wywołanie importu przy użyciu parsera rdf formy serializacji N3

N-TRIPLE - wywołanie importu przy użyciu parsera rdf formy serializacji N-Triple

Przykłady wywołania importu

Import pliku „*C:\rdf\ms_7.4_2.rdf*"

```
load "C:\rdf\ms_7.4_2.rdf" using RDFImporter
```

Import pliku „*C:\rdf\ms_7.4_2.nt*" z pełną formą URI, w formacie N-Triple

```
load "C:\rdf\ms_7.4_2.nt" using RDFImporter("FullURI, N-TRIPLE")
```