UNIVERSITATEA DIN BUCUREȘTI
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
MASTER – INGINERIE SOFTWARE

PARADIGMA PROGRAMĂRII ORIENTATE PE OBIECTE ȘI LIMBAJELE SEMANTICE

Profesor coordonator
CONF. DR. TĂTĂRÂM MONICA

Masterand
IORDACHE RAUL-MIHAI

BUCUREȘTI, 2018

Cuprins

1. Introducere în Semantic Oriented Framework (SOF)[1]

Programarea orientată pe obiecte este una dintre principalele paradigme folosite în dezvoltarea și în design-ul software. Introducerea **semanticii** în contextul programării este o necesitate pentru a putea manipula informații eterogene, existând totuși câteva provocări: de exemplu utilizarea metodelor pentru manipularea datelor de tip RDF(Resource Description Framework), conversia automată în format RDF, suportul multi-limbaj, etc. De asemenea, limitările din cadrul modelului obiectual (mai exact, din cadrul relațiilor dintre diferite entități) le depășesc pe cele are RDF, împiedicându-se o mapare directă, 1:1, între modelul object-oriented și Semnatic Web.

Noul concept SOF (Semantic Object Framework) combină design-ul orientat pe obiect cu facilitățile Semantic Web. SOF utilizează comentarii integrate în codul sursă pentru a descrie relațiile semantice dintre clase și atribute.

Pornind ca o extensie a World Wide Web (WWW), Semantic Web utilizează relații semantice între date, pentru a oferi servicii automate de analizare și de furnizare a datelor. Aplicațiile utilizează adesea tehnici de automatizare a proceselor, de căutare a datelor (*data searches*), de integrare a datelor precum și de reutilizare a acestora. Resource Description Frameworks (RDFs) reprezintă o modalitate de a reprezenta modele de date în cadrul Semantic Web. Un document de bază RDF conține subiecte, predicate și obiecte. Inginerii folosesc acest instrument de reprezentare pentru a proiecta procese și produse cu scopul de a maximiza schimbul de cunoștințe și informații. Majoritatea instrumentelor de proiectare existente se bazează pe o paradigmă orientată pe obiecte (O-O), dar neconcordanța dintre O-O și Semantic Web împiedică integrarea fără cusur a instrumentelor de proiectare curente în modele de date semantice (bazate pe Semantic Web).

Majoritatea dezvoltatorilor de software utilizează paradigma de proiectare orientată pe obiecte (OOP), dar aceasta este în mod evident inadecvată pentru prelucrarea datelor de tip SemanticWeb.

Cea mai folosită arhitectură de divizare a funcțiilor pentru proiectarea claselor OOP este modelul model-view-controller (MVC). Există mai multe instrumente de mapare obiect-relațională care pot converti modele asociate claselor în formate de înregistrare pentru bazele de date relaționale. Deoarece RDF folosește instrucțiuni triplu-orientate pentru formatarea datelor, aceasta diferă semnificativ de modelul MVC. În plus, clasele orientate pe obiecte nu pot fi folosite pentru a descrie relațiile semantice între atributele clasei, făcând astfel mai dificilă sarcina de a transforma obiectele model (din arhitectura OO) în format RDF. Cu cât este mai mare cantitatea de date existente care necesită conversia în format triplu, cu atât sunt mai mari provocările în ceea ce privește performanța și costurile.

Tehnologia O-O ascunde relațiile semantice în datele funcției din codului sursă. Tehnologiile O-O pure nu acceptă raționamentul sau inferența datelor ca în tehnologia Web Semantic. De asemenea, este greu pentru O-O să gestioneze surse de date eterogene, fără tehnologie de tip Semantic Web. Programarea orientată pe obiecte este matură și există multe modele de design care pot ajuta programatorii să scrie un cod sursă reutilizabil. Semantic Web poate publica informații pe Internet ca surse de date reutilizabile. Este un mijloc puternic de integrare a beneficiilor O-O (stilul de codare prietenos) și tehnologia Web Semantic (pagini web care pot fi citite automat).

1.1. Modelarea relațiilor semantice în context SOF - exemplu

Datele din agenda de adrese utilizată în următorul exemplu este acceptată de Gmail. Folosind limbajul Python ca exemplu specific, abordarea SOF este de a adăuga relații semantice la clase și atribute **atunci când sunt declarate**. Înainte de a efectua o interogare unitară în diferite agende (baze de date), un utilizator trebuie să definească mai întâi o clasă numită "Contact" pentru a partaja atributele comune. Din perspectiva semantică, această clasă este moștenită de la GContact (Gmail Contact).

```
class Contact(Model):
partOfName=''
partOfAddress=''
#owl:InverseFunctionalProperty Contact_email
email=''
phoneNumber=''

#Contact_officePhoneNumber rdfs:subClassOf Contact_phoneNumber
officePhoneNumber=''

#Contact_homePhoneNumber rdfs:subClassOf Contact_phoneNumber
homePhoneNumber=''

#Contact_mobilePhoneNumber rdfs:subClassOf Contact_phoneNumber
mobilePhoneNumber=''

#Contact_faxPhoneNumber rdfs:subClassOf Contact_phoneNumber
faxPhoneNumber=''
```

Conform modelului de proiectare MVC, clasa *Contact* aparține clasei *Model*, prin urmare clasa Contact (Model) este declarată ca reprezentând un *Contact* moștenit de la clasa Model.

Semnificația atributului "partOfName" este numele persoanei de contact, și conține un nume de familie / nume complet (nume+prenume) / poreclă etc. În acest caz, partOfName poate să reprezinte un nume complet sau orice segment dintr-un nume. Dacă semantica oricărui alt atribut este moștenită din partOfName, atributul este folosit pentru a identifica șirul de caractere care corespunde numelui.

În Python, semnul diez (#) desemnează un comentariu. Deoarece sintaxa SOF este încorporată în comentarii, orice instanță a "owl:" sau "rdfs:" inclusă într-un comentariu semnifică o declarație este specifică SOF. De exemplu, "#owl: InverseFunctionalProperty Contact_email" utilizează sintaxa OWL pentru a-și modifica semantica, ceea ce înseamnă că valorile șirului de caractere ale lui *Contact_email* trebuie să fie unic. În situațiile în care două obiecte de tipul Contact au același șir de caractere pentru „e-mail”, sistemul SOF identifică obiectele conflictuale și notifică programatorii, care pot aplica diverse strategii pentru a rezolva semantica „ilegală”. Instrucțiunile OWL[4] sunt utile pentru programatori atunci când sintaxele sunt bogate, pentru a preîntâmpina limitările din cadrul limbajului obiectual.

Numele de atribute pentru identificarea e-mailurilor diferă între aplicații. Exemplele din programele software includ: E-mail, e-mail, mail, Mail, emailAddress și EmailAddress - toate cu semantică identică. Pentru a afișa toate valorile atributelor pentru toate e-mailurile din agendele eterogene, toate atributele legate de e-mail trebuie să fie moștenite din clasa *Contact_email*.

```
#GContact rdfs:subClassOf Contact
class GContact(Model):
#GContact_name rdfs:subClassOf Contact_partOfName
name=''
#GContact_email rdfs:subClassOf Contact_email
email=''
#GContact_phone rdfs:subClassOf Contact_officePhoneNumber
#GContact_phone rdfs:subClassOf Contact_homePhoneNumber
phone=''
#GContact_mobile rdfs:subClassOf Contact_mobilePhoneNumber
mobile=''
#GContact_fax rdfs:subClassOf Contact_faxPhoneNumber
fax=''
company=''
title=''
#GContact_address rdfs:subClassOf Contact_partOfAddress
address=''
```

2. Probleme asociate integrării Semantic Web – OOP

Vom prelua din lucrarea *Integrating Semantic Web and Object-Oriented Programming for Cooperative Design,* o scurtă sumarizare a 7 probleme aferente integrării Semantic Web cu arhitectura de tip *object-oriented* (comparatie între diferite tehnologii semnatice):

Problemă	Jena	ActiveRDF	D2R	Eclass	***SOF***
Utilizarea metodelor pentru a manipula RDF	Nu	Da	Nu	Da	**Da**
Conversie automată în format RDF	Nu	Nu	Da	Da	**Da**
Suport pentru mai multe limbaje de progr.	Nu	Nu	Da	Nu	**Da**
Utilizarea directivelor pentru descrierea claselor și a semanticii atributelor	Nu	Nu	Nu	Da	**Da**
Actualizarea fișierelor de descriere semantică	Nu	Nu	Nu	Da	**Da**
Suport pentru surse de date eterogene	Nu	Nu	Nu	Nu	**Da**
Verificarea consistenței datelor (și a semanticii)	Nu	Nu	Nu	Nu	**Da**

2.1. Utilizarea metodelor pentru manipularea RDF

Chiar dacă interfețele de programare pentru framework-ul RDF furnizează soluții de scriere/citire/interogare, nu există mijloace de a utiliza obiecte pentru manipularea datelor în contextul RDF. Ca urmare, duratele de dezvoltare sunt mai lungi, codurile programului sunt mai mari, iar întreținerea este mai dificilă. Sistemul SOF utilizează proiectarea OO pentru a abstractiza API-urile RDF, cu scopul de a facilita scrierea de cod. În mod specific, sistemul acceptă utilizarea API-urilor OO pentru efectuarea de interogări, rezultatele de interogare corespunzătoare fiind returnate sub formă de obiecte.

2.2. Conversia automată a datelor în format RDF

Deși unele API-uri RDF sunt capabile să stocheze date triplu-orientate pentru scopuri de interogare semantică, dezvoltatorii trebuie să transforme obiectele model în format triplu-orientat - o sarcină destul de anevoioasă. Astfel, orice arhitectură de dezvoltare capabilă să convertească în mod automat obiectele în format RDF va ajuta dezvoltatorii să câștige timp și bani.

2.3. Suportul pentru mai multe limbaje de programare

În locul integrării sintaxei SOF într-un limbaj specific de programare orientat pe obiect, se poate adopta o strategie de utilizare a comentariilor care descriu semantica de clase și atribute pentru a susține utilizarea parserului SOF (cu modificări minime) pentru multiple limbaje de programare. În consecință, programatorii vor fi obligați să învețe SOF doar pentru a dezvolta aplicații.

2.4. Utilizarea directivelor pentru descrierea semanticii claselor și a atributelor

Cea mai simplă modalitate de a combina caracteristicile de design Semantic Web și OO este de a descrie semantica atributului sau al clasei, de preferință, în același timp în care sunt definite clasele. Cu toate acestea, definirea semanticii (atât pentru clasă cât și pentru atribute) necesită de obicei modificarea sintaxei limbajului de programare. Pentru a aborda această problemă fără a afecta în mod negativ sintaxa de programare inițială, sistemul SOF permite înglobarea de comentarii care facilitează utilizarea limitată a sintaxelor RDF și OWL.

2.5. Actualizarea fișierelor de descriere semantică

Unele soluții de implementare Semantic Web oferă fișiere de descriere independente care modifică în mod continuu relațiile dintre datele existente. Acest lucru necesită menținerea actualizărilor sincronizate între fișiere pentru a preveni neconcordanțele. Documentul API al programului și fișierele de cod sursă sunt independente, iar actualizarea documentelor de descriere este adesea ignorată, ajungându-se astfel la descrieri învechite și eronate. JavaDoc folosește comentarii încorporate pentru a preveni neconcordanțele dintre documentele API și codurile sursă ale programului, ceea ce face mai ușor pentru programatori să mențină coerența.

2.6. Suport pentru surse de date eterogene

Inconsecvența referitoare la numele coloanelor din diferite baze de date este frecventă (de exemplu, baza de date A poate folosi termenul "E-mail" și baza de date B "e-mail"). Pentru a efectua interogări consecvente care implică toate e-mailurile stocate în două baze de date, semantica celor doi termeni trebuie să fie clar definită, astfel încât computerele să le recunoască drept egale. Nu există în prezent nicio arhitectură pentru definirea relațiilor semantice între clase și atribute în codurile OOP care permite unui sistem să recunoască automat numele de atribute diferite cu semnificații identice. Probleme apar de asemenea atunci când se efectuează interogări unificate pe surse de date eterogene. Sistemul SOF permite utilizarea de comentarii pentru a menține o relație de moștenire între atribute și permite dezvoltatorilor să facă interogări unificate pe modele eterogene de date.

2.7. Verificarea consistenței datelor și a semanticii

Pot apărea conflicte între obiecte și semantică. De exemplu, atribuirea unui e-mail către un cont unic într-un sistem de gestiune poate avea ca rezultat un conflict ulterior atunci când două conturi au aceeași șir de caractere pentru câmpul e-mail. Sistemul SOF oferă API-uri pentru interogarea obiectelor pe care dezvoltatorii le pot utiliza pentru a efectua verificări de consistență semantică.

3. Programarea semantică orientată pe obiecte-SOOP[2]

Spre deosebire de abordarea SOF (menționată mai sus), în cadrul SOOP semantica devine o componentă conexă în cadrul unei paradigme (respectiv cea obiectuală).

Ideea fundamentală din spatele SOOP este de a crea sau de a popula o ontologie din interiorul C++ utilizând clase și obiecte obișnuite ale limbajului C++.

Aceasta înseamnă că realizăm maparea datelor în direcția opusă în comparație cu direcția abordată anterior. Ca avantaj al acestei abordări, eliminăm multe dintre problemele obișnuite care rezultă din încercarea de a mapa date de la o reprezentare mai puternică a datelor într-una mai puțin puternică. În plus, abordarea SOOP ne permite nu numai să păstrăm conceptul de *static-type*, dar și să folosim activ sistemul orientat pe ontologie. Este posibilă utilizarea entităților C++ ca tipuri regulate C++ și în afara contextului ontologiei.

Ontologie → OOP	**OOP → Ontologie**
Maparea relațiilor dinamice a entităților într-un tip static - **imposibilă**	Maparea tipului relației (al unui tip static) într-o ontologie dinamica – **posibilă (f. ușor)**
Obiecte arbitrare pot fi legate utilizând relații arbitrare. Deci, **translatare grea** (în atribut obiectual)	Conexiunile atributelor sunt destul de simple în OOP, și **ușor de mapat** într-o ontologie
O ontologie este un hipergraf care prezintă (în general) un set complex de relații. Foarte puțin probabilă **maparea** (chiar dacă limbajul suportă moștenirea multiplă)	Moștenirea directă, aciclică din OOP este defapt un subgraf (al unei ontologii mari – hipergraf -)
Mecanismul ontologiei este tocmai evoluția dinamică a structurii. Un lucru destul de greu de translatat într-un limbaj static.	Structurile moștenite (din OOP) sunt foarte ușor de transformat într-o ontologie dinamică

În prezent, interacțiunea dintre ontologii și limbaje de programare generale constă, în cea mai mare parte, dintr-o mapare indirectă între ontologie și limbajul de programare. Limbajul și bibliotecile sunt adesea utilizate, în principal, ca editor de ontologii. Vom prezenta o abordare diferită, care creează o ontologie din obiectele obișnuite de date într-un limbaj *static-typed,* și anume C++. Deoarece în cazul de față maparea pornește de la limbajul de programare orientat pe obiect (OOP) la ontologie, putem evita multe probleme specifice abordărilor care pornesc din direcția opusă. Mai mult, interacțiunea dintre domenii devine mai directă și poate fi în mare parte păstrată, fără construcții de limbaj neobișnuite. Paradigma de programare declarativă, pe de o parte,

și paradigma imperativă (+ orientată pe obiecte), pe de altă parte, sunt combinate astfel în cadrul paradigmei programării semantice orientate pe obiect (SOOP). Ca rezultat, SOOP permite programatorilor să utilizeze direct tehnologii semantice, în special raționamente, pentru obiectele lor obișnuite din limbajul C++.

O ontologie este o specificare formală și explicită a unei conceptualizări comune. Aceasta înseamnă că este o modalitate adecvată de a reprezenta date și raționamente pentru teoreticieni, astfel încât să se poată căuta răspunsuri, în mod algoritmic, la întrebările exprimate în limbajul ontologic. De exemplu, un subiect modelat folosind o ontologie, poate fi verificat din punctul de vedere al consistenței. În mod similar, setul de date care satisface o formulă poate fi extras.

Din moment ce două domenii diferite sunt reunite, tehnologiile semantice, pe de o parte, și programarea orientată pe obiecte (OOP), pe de altă parte, trebuie să definim anumiți termeni: **atomul** se referă la o entitate logică unică; o axiomă este o afirmație logică care este fie adevărată, fie falsă. **Predicatul** și cuantorul sunt folosite în același mod ca și în calculul predicatelor sau în logica formală. De asemenea, **variabila** este folosită în contextul calculului predicatelor. Termenii de *clasă* și *obiect* au același înțeles ca în OOP, ei referindu-se la definirea unui tip de date și a instanțelor sale. Entitatea este utilizată în ambele domenii: se poate referi la un atom (domeniul semantic) sau la obiecte (domeniul OOP).

OOP	Vocabular comun	Ontologii
Clasă **Obiect (instanță)**	**Entitate**	**Atom, Axiomă, Variabilă (cu sens diferit față de OOP), Predicat, Cuantor**

Vocabularul utilizat în OOP și în domeniul semanticii

Cea mai obișnuită metodă de a reprezenta o ontologie (împreună cu datele asociate) este în prezent, prin intermediul unui **editor de ontologii** sau a unei interfețe de programare a ontologiilor (*ontology API*). Cu toate acestea, chiar și API-urile care permit definirea ontologiilor în cadrul limbajelor de programare nu acceptă o combinație directă a structurilor de date existente în limbaj cu cele din ontologie. Rezultatul este o diviziune puternică între datele brute și datele semantice. Datele brute sunt ușor accesibile din limbajele de programare (de exemplu: afișare, sortare, etc.). Datele semantice, prezente în ontologie, sunt ușor accesibile *gândirii umane*, dar sunt greu de transpus în diferite limbaje de programare.

Un exemplu pentru acest tip de decalaj este API-ul OWL *(OWL-API)*. În timp ce datele semantice sunt ușor accesibile din Java, obiectele Java care reprezintă entități semantice nu au decât un singur tip (specific limbajului Java) - și astfel nu au nici o semantică din punctul de vedere al limbajului de programare: Entitățile au tipuri (ca de ex. *OWLLiteral* sau *OWLNamedIndividual*); acest lucru înseamnă că utilitarul de verificare a tipurilor statice nu poate identifica multe dintre erorile evidente (cum ar fi o adresa de e-mail atribuită unui nume de utilizator).

Scopul SOOP este de a dezvolta o nouă abordare care îmbină ontologiile împreună cu limbajele de programare, evitând în același timp pierderea informațiilor, realizând totodată o integrare mai profundă în comparație cu metodele existente. Pentru a evita problemele datorate puterii expresive mai mari ale ontologiilor comparativ cu limbajele de programare orientate pe obiecte (OOP), în special cele care utilizează tehnica *static-type*, SOOP inversează direcția de mapare: În loc de a crea ontologii cu un editor de ontologie independent și de a importa ulterior descrierea rezultată în limbaje de programare, prezenta abordare construiește direct schema în manieră OOP. În acest scop, s-au extins structurile de date ale limbajului de programare astfel încât

acestea să poată fi folosite ca entități semantice și combinate cu ontologii existente. Deoarece ontologiile au o putere expresivă mai mare decât limbajele OOP, ele pot reprezenta cu ușurință toate conexiunile semantice care există în OOP.[3] [5]

În prezent, ontologiile sunt create în primul rând cu ajutorul editorului de ontologii. În mod special, cunoscut în acest context este Protege. În ceea ce privește utilizarea din cadrul limbajelor de programare, există o gamă relativ largă de abordări.

În primul rând, vor fi descrise acele abordări în care entitățile unei ontologii sunt reprezentate în OOP printr-un set de clase generice. Scopul nu este de a crea reprezentări exacte ale ontologiei în limbajul OOP, ci de a oferi un instrument programabil pentru editarea acestora. Aceste abordări sunt numite și *abordări indirecte*. Cele mai importante biblioteci de software care pot fi menționate aici sunt API-ul OWL și Jena din proiectul Apache.

API-ul OWL este o bibliotecă Java, care permite accesul la tripletele OWL cu un nivel relativ scăzut de abstractizare. API-ul oferă atât posibilitatea de a pune întrebări prin interogări, cât și de a edita ontologia. Schimbările efectuate in bibliotecă pot fi redactate pentru a putea fi utlizate ulterior. API-ul OWL este folosit ca backend de Protege. Apache Jena urmează o abordare similară, dar oferă un nivel mai ridicat de abstractizare. Ideea de bază a Jena este reprezentarea datelor și a tipologiilor de ontologii ca obiecte ale clasei derivate din *OntRessource* (de exemplu *resurse*) și utilizarea acestora prin metode generice. Dezavantajul evident atât al API-ului OWL, cât și al lui Jena este lipsa de suport în ceea ce privește obiectele de ontologie din mediul Java și utilizarea șirurilor de caractere în locul obiectelor din limbajul nativ.

Mai mult decât atât, este posibilă realizarea unor interogări simple în Jena cu ajutorul limbajului Java nativ (de exemplu, pentru a itera clasele). Cu toate acestea, metoda recomandată pentru interogările complexe este folosirea limbajului de interogare SPARQL (Protocolul SPARQL și RDF Query Language). SPARQL este un limbaj standardizat de W3C și modelat după SQL (Structured Query Language). Acesta permite formularea unor interogări semantice relativ complexe.

La celălalt capăt al spectrului de abordări existente, există încercări de a traduce datele din ontologii în limbaje obișnuite pentru OOP, cunoscută și sub denumirea de "direct approach". Din păcate, există unele probleme fundamentale ale acestui tip de traducere, datorită faptului că ontologiile sunt mult mai generale și mai expresive decât majoritatea limbajelor OOP. Cu toate acestea, câteva abordări interesante pot fi găsite în literatura de specialitate.

ActiveRDF, de exemplu, ocolește multe dintre limitările cunoscute, folosind Ruby ca limbaj de implementare a acestuia, în loc de Java, care este folosit pe scară largă. Ca urmare, multe dintre problemele comune create de un sistem *static-type* dispar, deoarece Ruby nu este doar *dynamic-typed*, ci permite, de asemenea, să alerteze orice încercare de a utiliza funcții inexistente. Cu toate acestea, tocmai acest sistem *dynamic-typed* implică mai multe dezavantaje pentru această abordare. Cel mai important lucru este lipsa verificării la compilare(*compile-time check*), aspect care este prezent în cadrul sistemelor *static-type*.

3.1. Prezentare generală

Pentru a structura conceptul SOOP, acesta este împărțit în următoarele submodule:

1. Un modu de demonstrație.
2. Un modul ontologic care gestionează cunoștințele și creează interogări pentru modulul de demonstrație

3. Un modul care implementează clasa de bază a tuturor entităților care sunt susținute de un obiect C++: *entity.*
4. Un modul care implementează formule într-un mod în care se păstrează semantica lor într-o manieră utilizabilă, oferind în același timp o sintaxă simplă în C++ care seamănă foarte mult cu calculul unui predicat.

Clasa de bază *entity* stochează un pointer la ontologia asociată și un număr întreg care servește ca unic ID în cadrul ontologiei. Datorită semanticii neclarite a copierii, nu este furnizat niciun constructor de copiere. În schimb, sunt permise operațiunile de mutare prin informarea ontologiei despre noua adresă a entității mutate. Fiecare clasă care se moștenește din *entity* poate fi utilizată ca entitate a contextului ontologic. Prin urmare, este normal ca aceste clase să aibă membri și metode de date obișnuite, a căror semnificație sau chiar existență să nu fie formalizată în cadrul ontologiei. În plus, permite utilizarea membrilor prin intermediul semanticii limbajului de programare, cum ar fi funcțiile de sortare sau de afișare.

Pentru a sprijini utilizarea tipurilor de date deja existente, este furnizat un șablon de clasă ***e*** care împachetează (*wrapping*) tipurile existente astfel încât versiunea înfășurată (e <T>) să moștenească *entity* și astfel să devină utilizabilă în cadrul ontologiei.

Predicatele în SOOP constau in două părți: în primul rând, un șablon de clasă, ale cărui instanțe reprezintă predicate plus argumentele sale. În al doilea rând, un șablon de funcție care instanțiază șablonul de clasă. Prin convenție, șablonul funcției este denumit identic cu predicatul (de exemplu, f) iar șablonul de clasă primește sufixul *_template*. Prin încorporarea/imbricarea unor astfel de predicate, este posibil să se creeze expresii și tipuri mai complicate.

Clasa *ontology* stochează o listă a tuturor obiectelor C++, care sunt în același timp entități ale acestei ontologii. Aceeași clasă stochează și lista tuturor axiomelor asociate acestei ontologii. Verificările unui set specific de axiome (dacă este satisfăcut sau nu grupul de axiome) se procesează prin crearea unei reprezentări textuale complete a întregii ontologii, care este apoi trimisă la modulul de demonstrație, a cărui răspuns este returnat apelantului în C++. În general, este posibilă nu numai verificarea satisfacerii unei formule, ci și solicitarea setului de atribuiri de variabile din model și returnarea obiectelor C++ care sunt asociate cu acestea. Solicitările pentru aceste entități sunt procesate în mod similar cu verificarea condițiilor de satisfacere, cu excepția faptului că este definită o funcție suplimentară care mapează toate entitățile la SOOP ID-ul corespunzător.

Pentru a profita de puterea completă a logicii descriptive, sunt necesare în plus cuantificatori și variabile. Abordarea prezentă creează tipuri unice pentru toate variabilele, oferind un șablon de clasă *variable* care primește identificatorul ca argument al șablonului și o clasă *bound_vars* care primește un număr volatil de variabile în constructorul său și creează în interior o reprezentare textuală a acestei liste ca șir de caractere. Astfel, este posibil să se implementeze *cuantificatorii* în același mod ca și predicatele, cu excepția faptului că de data aceasta, cuantificatorii primesc o instanță de tipul *bound_vars* ca prin argument al constructorului.

Aceste caracteristici combinate permit crearea ușoară a formulelor complexe fără a necesita o sintaxă neobișnuită atât din prisma logicii descriptive cât și a limbajului C++. Putem presupune, de exemplu, că un predicat f este tranzitiv, prin intermediul următorului cod C ++ :

forall ({x,y,z}, implies (and_(f(x,y),f(y,z)), f(x,z))).

3.2. Implementare

Pentru a crea un predicat cu numele *predicate*, trebuie să realizăm un șablon *predicate_template* și o funcție *predicate*.

```
template<typename T>
class predicate_template : soop::is_predicate {
public:
    predicate_template(const T& arg) : arg{ arg } {}
    void collect_entities(std::vector<std::size_t>& ids);
    void stream(std::ostream& out, const std::vector<std::string>& names) const;
private:
    T arg;
};

template<typename T>
auto predicate(const T& arg)
-> predicate_template<soop::to_bound_type<T>> {
    return { arg }
}
```

Tipul *soop:is_predicate* este moștenit pentru oferi facilitatea de a afișa și de a colecta argumentele în concordanță cu șablonul de funcție utilizat. Metodele *collect_entities* și *stream* sunt necesare pentru managementul diferitelor formule. Ideea de bază din spatele acestei arhitecturi este de a permite utilizatorului de a imbrica predicate, astfel încât fiecare *șablon_predicat* să fie instanțiat cu tipul de date dat ca argument.

Deoarece definiția predicatelor necesită scrierea unui număr mare de linii de cod, SOOP oferă patru macrocomenzi care permit definirea unui predicat nou într-o singură declarație:

SOOP_MAKE_TYPECHECKED_RENAMED_PREDICATE(Id, Name, Rank, ...)
SOOP_MAKE_RENAMED_PREDICATE(Id, Name, Rank)
SOOP_MAKE_TYPECHECKED_PREDICATE(Id, Rank, ...)
SOOP_MAKE_PREDICATE(Id, Rank)

3.3. Relația șef-subaltern modelată în context SOOP (C++)

Pentru a ilustra integrarea informațiilor semnatice într-un limbaj orientat pe obiecte, luăm ca punct de reper relația șef-subaltern. Astfel, arhitectura SOOP rezultată este:

```cpp
#include <iostream>
#include <string>
#include <soop/onto.hpp>

class angajat {
public:
    angajat(std::string nume) : nume{ std::move(nume) } {}
    std::string nume;
};

using angajat_e = soop::e<angajat>;
SOOP_MAKE_TYPECHECKED_PREDICATE(parent_of, 2, angajat_e, angajat_e);
//predicatul parent_of cu sensul relatiei: sef-angajat

int main() {
using namespace preds;
using namespace soop::preds;
soop::ontology o{};
o.add_type<angajat_e>();
o.add_predicate<parent_of_t>();
soop::variable<'sef'> sef;
soop::variable<'subaltern'> subaltern;

// seful si subalternul sunt angajati:
o.add_axiom(forall({ sef, subaltern }, implies(parent_of(sef, subaltern), and_(
        instance_of(sef, soop::type<angajat_e>),
        instance_of(subaltern, soop::type<angajat_e>)))));

// axioma: seful unui subaltern, nu este subalternul unui sef:
    o.add_axiom(forall({ sef, subaltern }, implies(parent_of(sef, subaltern),
        not_(parent_of(subaltern, sef)))));
    angajat_e PopescuSef{ o, "Popescu Alexandru" };
    angajat_e IonescuSubaltern{ o, "Ionescu Adrian" };
    o.add_axiom(preds::parent_of(PopescuSef, IonescuSubaltern));
    soop::variable<'s'> s;
    const auto& sef = std::get<0>(
        o.request_entities<angajat_e>(
            exists({ subaltern }, parent_of(s, subaltern)), s));
    std::cout << sef->nume << " este un sef.\n";
}
```

Chiar dacă unele detalii sau axiome au fost omise, exemplul de mai sus oferă o vedere de ansamblu asupra SOOP (așa cum este implementat în C++). În principal, există două alternative pentru SOOP, cu proprietăți foarte diferite. Putem evidenția atât utilizările alternative ale ontologiilor cât și abordările care nu utilizează ontologii.

3.4. Avantaje și dezavantaje ale SOOP

În ceea ce privește primul aspect, SOOP integrează ontologiile și limbajele obișnuite pentru OOP mult mai stricte decât abordările existente. Puterea expresivă este suficient de mare pentru a descrie chiar și probleme complicate (cum ar fi problema NP-completitudine), încă asigurându-se că fiecare afirmație poate fi reprezentată în C++. În plus, SOOP nu necesită niciodată compromiterea siguranței de tip (*type-safety*) sau tratarea explicită a traducerii datelor din ontologie (de obicei, șiruri de caractere) în obiecte OOP.

Excluzând funcționalități precum parsarea intrărilor, care sunt întotdeauna necesare, crearea unui software de lucru cu SOOP a necesitat mai puțin de o sută de linii de cod pentru logica programului real. Efortul este mai mic sau egal decât efortul necesar prelucrării cu alte API-uri

pentru ontologii .

Ideea principală atunci când lucrăm cu SOOP este formalizarea problemei, care, de altfel, nu este un pas absolut necesar. Cu toate acestea, acest lucru are un preț mic, deoarece formalizarea este întotdeauna recomandabilă atunci când rezolvăm problemele complexe, dacă se dorește o soluție corectă. Utilizarea SOOP pentru a formaliza problema are avantajul suplimentar că este mai ușor să observăm formele incomplete, deoarece cazurile simple de testare pot produce rezultate evident greșite. Deoarece totul este efectuat în cadrul C++, cazurile de testare pot fi generate în același mod și cu aceleași instrumente ca și pentru testarea unitară (*unit-testing*).

O restricție privind gradul de utilizare a SOOP este că formalizarea problemelor poate fi extrem de dificilă, ceea ce înseamnă că o implementare imperativă poate fi mai ușoară pentru multe probleme simple. Cu toate acestea, argumentul anterior se aplică, în general, ontologiilor și tehnologiilor semantice care deseori își manifestă potențialul doar în domenii cu un anumit nivel de complexitate.

Un avantaj al SOOP constă, cu siguranță, în faptul că permite amestecarea cu ușurință a soluțiilor imperative cu raționamentul ontologic. Astfel, diferite aspecte ale aplicației pot fi rezolvate în domenii diferite (OOP sau raționamentul semantic declarativ), în funcție de dorința programatorului (după ce analizat ambele posibilități de abordare). Acest lucru întărește ideea de *multiparadigmă* a limbajului C++.

Mai mult, bibliotecile care utilizează SOOP prezintă o conceptualizare comună pe două nivele: Din perspectiva OOP, clasele SOOP pot fi conectate la tipuri de date create de utilizatori prin moștenire și agregare, și astfel reutilizate la fel ca și bibliotecile non-SOOP. În același timp, datorită legăturii dintre tipurile OOP și o ontologie SOOP, diferite biblioteci derivate din aceeași superclasă sunt de asemenea compatibile la nivel semantic.

4. Concluzii

Există două abordări, prezentate succint în lucrarea de față:

- ➔ SOF (Semantic Oriented Framework), arhitectură care utilizează strict comentarii pentru adăugarea semanticii în cadrul aplicației (dezvoltată, în mod uzual, în manieră OOP)
- ➔ SOOP (Semantic Object Oriented Programming), concept care unește logica semantică, logica declaratică cu limbajul de programare orientat pe obiecte.

Ambele tehnologii au avantaje și dezavantaje, motiv pentru care, rămâne la latitudinea programatorului să aleagă metoda convenabilă aplicației acestuia. Chiar dacă a doua variantă este o abordare destul de versatilă, aceasta nu este încă matura, astfel încât să fie aplicată pe scară largă. Totuși, atunci când este necesară combinarea logicii semantice în diverse aplicații (dezvoltate în manieră OO), conceptele prezentate mai sus pot reprezenta soluții viabile.

Bibliografie

1: Petr Jezek, Roman Moucek, Semantic framework for mapping object-oriented model to semantic web languages, 2015,https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4340193/
4: Mikhail K. Levin and Lindsay G. Cowell, owlcpp: a C++ library for working with OWL ontologies, 2015,https://jbiomedsem.biomedcentral.com/articles/10.1186/s13326-015-0035-z
2: Heinrich C. Mayr,Martin Pinzger, , 2016,https://subs.emis.de/LNI/Proceedings/Proceedings259/313.pdf
3: , A Semantic Web Primer for Object-Oriented Software Developers, 2006,https://www.w3.org/TR/sw-oosd-primer/
5: , Semantic Oriented Programming, ,https://en.wikipedia.org/wiki/Semantic-oriented_programming