Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs
Rendszerek Tanszék

Ontológia kezelő modul tervezése szöveges információt kezelő informatikai rendszer számára

Diplomaterv
2004

Förhécz András

nappali tagozat
műszaki informatika szak
V. évfolyam

Konzulens: Strausz György

Nyilatkozat

Alulírott Förhécz András, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2004. május 14. ....

Förhécz András

Kivonat

Napjainkban, az információs társadalom korában az Internet rendkívül gyors ütemű fejlődése következtében temérdek kiaknázandó információ halmozódott fel, mely segíthetne a felhasználóknak munkájuk gyorsabb elvégzésében. Azonban a Web hatalmas mérete és a dokumentumok strukturálatlansága miatt nagyon nehéz megtalálni a megfelelő anyagokat. Jelenleg a World Wide Web jelentős részét emberek számára olvasható formában helyezték el, nem pedig úgy, hogy számítógépek tartalmilag helyesen kezelhessék.
Ennek megoldására információ kinyerő rendszereket használnak, melyek automatikusan elvégzik a szövegek szemantikus indexelését, lehetővé téve a tartalmukon alapuló keresést. A nemzetközi IKF projekt célja a fenti feladatra egy keretrendszer kialakítása.
Az információ kinyerés azonban általában nem oldható meg hatékonyan, csak egy behatárolt tárgyterületen belül, melynek fogalmi rendszerét formálisan meg kell fogalmazni a számítógép számára. Diplomamunkám során az IKF rendszerprototípus azon komponensét készítettem el, mely a cél környezetet leíró fogalmi rendszert, az ontológiát képes kezelni és a többi komponens számára elérhetővé tenni.

Abstract

Nowadays, in the age of information revolution, the rapid growth of the Internet has led to the accumulation of enormous amounts of publicly available information waiting for utilization, which may help out people with their daily tasks. However it's difficult to find the appropriate material, caused by the huge number and unstructured form of documents. Most of the Web's content today is designed for humans to read, not for computer programs to manipulate meaningfully.

To overcome this problem, information extraction methods are used, which make semantic indexing automatically, enabling content-based queries. Within the framework of the international Information and Knowledge Fusion (IKF) Project these techniques are investigated, developing a knowledge-based information retrieval system.

However without restrictions information extraction can't be solved efficiently, only in the scope of a narrower application domain. The domain model should be formalized for the computer. This master thesis discusses the design and implementation of the component responsible for the formal domain model, the ontology, managing and making it available for the other components of the IKF framework.

Tartalomjegyzék

Bevezetés 1

1. Szemantikus Web 4

1.1. A Szemantikus Web architektúrája . . . 5

1.2. Mi az ontológia? . . . 8

2. Az IKF rendszerprototípus 10

2.1. A háttértudás tárolásának lehetőségei . . . 12

2.2. Az ontológia szerepe az IKF-ben . . . 14

3. Ontológia technológiák 15

3.1. Ontológia nyelvek . . . 15

3.1.1. RDF és RDF sémák . . . 15

3.1.2. KAON ontológia nyelv . . . 18

3.1.3. DAML+OIL . . . 21

3.1.4. DL-workbench . . . 24

3.1.5. OWL . . . 26

3.2. Ontológia szerkesztők . . . 27

3.2.1. OilEd . . . 27

3.2.2. KAON OI-modeler . . . 28

3.2.3. Protégé . . . 29

3.3. Ontológia programozói felületek . . . 31

3.3.1. KAON API . . . 31

3.3.2. Jena . . . 31

3.3.3. Protégé . . . 32

3.3.4. OWL API . . . 32

3.4. Technológiák választása . . . 33

4. Következtetés ontológiával 35

4.1. Az ontológiai következtetés fajtái . . . 35

4.2. A következtetés célja . . . 36

4.3. Leíró logikák . . . 37

4.3.1. $\mathcal{SHIQ}$ nyelv . . . 40

4.4. Következtetőgépek . . . 42

4.4.1. FaCT, FaCT++ . . . 42

4.4.2. RACER . . . 43
4.4.3. Vampire . . . 44
4.4.4. Pellet . . . 45
4.4.5. TRIPLE . . . 45
4.4.6. Választás indoklása . . . 46

5. Tervezés **48**
5.1. Modul specifikáció . . . 49
5.2. Architektúra . . . 51
5.3. Függőségek kezelése . . . 52
5.4. Ontológia lekérdező API . . . 53
5.5. Konkurens kérések kiszolgálása . . . 54

6. Implementáció **56**
6.1. Választott technológiák . . . 56
6.1.1. Java . . . 56
6.1.2. OWL API . . . 56
6.1.3. RACER . . . 57
6.1.4. SOAP . . . 57
6.2. Ontológiák kezelése . . . 58
6.3. Grafikus felhasználói felület . . . 60
6.4. RACER kezelése . . . 61
6.5. API szerver . . . 63
6.6. Végleges architektúra . . . 65
6.7. Demonstráció . . . 66
6.8. Teljesítményelemzés . . . 69

7. Továbbfejlesztési lehetőségek **71**

8. Összegzés **73**

Függelék **75**
A. Szolgáltatott primitívek leírása . . . 75
B. Grafikus felhasználói felület . . . 78
C. Demonstrációs ontológiák . . . 79

Irodalomjegyzék **84**

Ábrák jegyzéke

1. A Szemantikus Web rétegei . . . 6
2. Az IKF rendszer felépítése . . . 11
3. Az RDF gráfos ábrázolási módja . . . 16
4. Az RDFS és az objektum-orientált nyelvek kapcsolata . . . 17
5. A KAON entitásainak bemutatása . . . 20
6. Az OIL dialektusok és az RDFS viszonya . . . 21
7. A DL-workbench platform architektúrája . . . 24
8. Az OI-modeler ontológia szerkesztő gráf alapú ábrázolása . . . 28
9. A Protégé OWL Plugin képes az OWL kifejezések kezelésére . . . 30
10. Leíró logikán alapuló tudásábrázoló rendszer architektúrája . . . 37
11. Az IKF ontológia modul architektúrája . . . 52
12. A modul konfiguráció (*config* csomag) UML diagramja . . . 59
13. Az OWL és RACER-kezelés (*owl, racer* csomagok) UML diagramja . . . 62
14. Az ontológia szerver (*api* csomag) UML diagramja . . . 63
15. Konkurens kérések kiszolgálásának pszeudo-kódja . . . 64
16. Az IKF ontológia modul részletes architektúrája . . . 65
17. A modulok függőségi gráfja . . . 66
18. Figyelmeztetés inkonzisztens ontológia esetén . . . 67
19. Klasszifikált ontológia a teszt kliens kimenetén . . . 68
20. A sárgabarack és a sárgás gyümölcs a Protégé OWL Plugin jelöléseivel . . 68
21. Teljesítmény mérése . . . 70
22. Az ontológiák kezelését végző felhasználói panel . . . 78
23. Az ontológia modulok függőségeit ábrázoló felhasználói panel . . . 78
24. Az API szerver működését, terhelését jelző felhasználói panel . . . 78

Táblázatok jegyzéke

1. Az $\mathcal{AL}$ nyelv konstruktorai . . . 38
2. Néhány DAML+OIL axióma $\mathcal{SHIQ}$ logikai megfelelője . . . 41
3. A teljesítményelemzéshez használt ontológiák mérete . . . 69
4. Fogalmak lekérdezése, igaz-hamis értékű függvények . . . 75
5. Fogalmak listájával visszatérő lekérdezések . . . 75
6. Tulajdonságok lekérdezése, igaz-hamis értékű függvények . . . 76
7. Tulajdonságok listájával visszatérő lekérdezések . . . 76
8. Példányok lekérdezése, igaz-hamis értékű függvények . . . 77
9. Példányok listájával visszatérő lekérdezések . . . 77

Bevezetés

Az informatika fejlődése folyamatosan maga után vonja az Internet fejlődését is. A rohamos növekedést önmagában az is jelentősen segíti, hogy a szakterületen dolgozók túlnyomó többségének gyakorlatilag munkatere a Világháló. Az elmúlt években azonban más területeken dolgozók is részben vagy teljesen „felköltöztek” a Webre: leginkább akkor képesek hatékonyabban dolgozni, ha komoly előnyökkel jár a tetemes mennyiségű információ gyors mozgatásának, megosztásának képessége.

Az információ fogyasztókkal egy időben megjelentek az információ termelői is: számtalan hírforrással, akár ingyenesen elérhető folyóirattal, néha még könyvekkel is találkozhatunk a legkülönbözőbb tudományterületekről. A temérdek anyag közül azonban koránt sem egyszerű feladat a hasznosak kiválogatása. A World Wide Web méretének becslésével foglalkozók megkülönböztetik a Web felszínét (*surface Web*) a mély Webtől (*deep Web*). Az előbbibe csak az egyedien előállított, tartalmilag kézzel begépelt – technológiai szempontból nem feltétlenül statikus – oldalak tartoznak bele, míg utóbbi tartalmazza az adatbázisokból automatikusan generált oldalakat is. Ugyan a mély Web becslések szerint 4–500-szor nagyobb, a tartalmilag talán igényesebb, több értékes, naprakész információval szolgáló felszín is oldalak milliárdjait tartalmazza.

A releváns információ megtalálása a rengeteg oldal között már csak azért is reménytelen vállalkozás, mert a legnagyobb keresők, a Google és az Inktomi (MSN és Yahoo) sem képesek lépést tartani a növekedéssel. Mára azonban a keresés teljessége, azaz minden releváns oldal visszaadása nem is lehet cél, helyette a pontosságon próbálnak javítani.

A webes kulcsszavas keresők a relevancia megállapításakor az oldalakon lévő szavakra vonatkozó statisztikai módszereket, például az egyes szavak relatív gyakoriságát, együttes előfordulások számát, és *page ranking*-et, azaz a linkek szerkezetéből számolt fontossági mértéket használnak. A találatok és a keresőkérdés tartalmi (szemantikus) feltérképezésével, értelmezésével nem is próbálkoznak: tehetetlenek az azonos alakú szavakkal szemben, nem ismerik a szavak szinonimáit, és nem adhatjuk meg a kulcsszavak közti összefüggéseket sem. Általános esetben reménytelen vállalkozás is lenne, mivel az oldalak annyira heterogének mind a nyelvet, mind pedig a szavaknak vélt betűösszetételek jelentését tekintve (egy szó lehet, hogy valójában egy alkatrész kódját jelöli), hogy a szemantikán alapuló módszerek túl sok bizonytalanságot rejtenek magukban.

A Szemantikus Web kezdeményezés a keresési problémát más szolgáltatásokkal együtt úgy próbálja megoldani, hogy a természetes nyelvű dokumentumok írása

mellett a weblapok készítőire további feladatokat ró: a számítógép számára is értelmezhető módon, formálisan is meg kell fogalmazniuk a kulcsfogalmak jelentését és viszonyait. A formális leírások logikai állításokká alakíthatók, így egzakt módon, a logika eszközeivel kezelhetőek a keresések – persze a teljesítmény által behatárolt keretek között. A technológia új lehetőségeket teremthet, megválaszolhatóvá tehet olyan keresőkérdéseket, mint például „keresd meg az összes Kovács vezetéknevű férfit, akiknek Mária nevű lányuk van, és vízparton laknak". Sajnos a fejlődés jelenlegi ütemét nézve a megvalósulás még várat magára, a technológia széles körű elterjedése pedig kétséges, mivel a meglévő dokumentumok átalakítását igényli.

A nemzetközi IKF projekt hasonló célok teljesítését egészen más megközelítéssel próbálja elérni. Behatároljuk a számunkra érdekes szakterületet, más néven a probléma tárgyterületét, és fogalmait megpróbáljuk formálisan minél pontosabban leírni. A kísérleti alkalmazás során a gazdasági területre esett a választás, mivel az információforrások és az igények szempontjából is kiemelt jelentőségű. A fogalmi rendszer ebben az esetben például a gazdasági szerepeket, a pénzpiaci eszközöket, a gazdasági társaságok lehetséges tagjait, cselekvéseit és mindezek viszonyát tartalmazza.

A keresés során figyelembe vett dokumentumokat a szakterületnek megfelelő kevésbé heterogén, ellenőrzött csoportra korlátozzuk, kezdetben például néhány gazdasági hírportál tömör cikkeiből indulunk ki. Végül a formális fogalmi rendszer segítségével információt nyerünk ki a dokumentumokból. A forrásdokumentumok gondos megválasztásával az információforrás tartalmi és formai szempontból is megbízhatóbb, várhatóan eredményesebben kezelhető a szemantikus információ.

A szerzett tudást az IKF rendszer tudásbázissá szervezi, melynek tartalmát közvetlenül is lehet adminisztrálni. A felhasználók kulcsszavakkal kereshetnek a tudásbázisban, vagy űrlapok kitöltésével előre definiált, sablon lekérdezéseket futtathatnak.

A diplomaterv célja a tudáskinyerés során felhasznált fogalmi rendszer kezelését végző komponens létrehozása. A fogalmi rendszert ontológia segítségével írhatjuk le formálisan, a számítógép számára is értelmezhető módon. A moduláris ontológiák kezelése, a tárolt információ konkurens, következtetéseket igénylő lekérdezése által felvetett problémákat kell megoldani.

A dolgozat áttekintése. Először az IKF megközelítésétől eltérő szemléletű Szemantikus Web kezdeményezést mutatom be, mivel eredményeiből sokat merít az IKF projekt is. Bevezetem az ontológia fogalmát, majd a 2. fejezetben részletesen ismertetem az IKF rendszerprototípus szerkezetét. Megindokolom, miért ontológiát használunk a fogalmi rendszer ábrázolására, majd leírom, a keretrendszerben hogyan segítheti az ontológia a tudáskinyerést és lekérdezést.

A komponens megtervezése előtt, a 3. fejezetben bemutatom az ontológiához szorosan kapcsolódó technológiákat, majd kiválasztom a megvalósítás során alkalmazni kívántakat. Ezek közül az ontológia nyelv megválasztása a legfontosabb, de az ontológia kezelését megkönnyítő programozói felületet is kiválasztom, és javaslatot teszek a fogalmi rendszer tervezésekor használt ontológia szerkesztőre is.

Az ontológiát beágyazó modul feladata a fogalmi rendszer lekérdezhetőségének biztosítása. A lekérdezések azonban nem csak az ontológia leírásában közvetlenül szereplő állításokra terjednek ki, hanem azokból következtetőgép segítségével kell megválaszolni a kéréseket. A 4. fejezetben bemutatom az ontológia alapú következtetés fajtáit, majd részletesen ismertetem a leíró logikákat, melyek a választott reprezentáció, az OWL nyelv logikai megfeleltetéséül szolgálnak. Bemutatom a komponens kiszolgálására alkalmas következtetőgépeket, majd kiválasztom a feladatnak legmegfelelőbbet.

Az ontológia kezelő modul tervezését a feladat elemzése után a részletes specifikáció elkészítésével kezdem az 5. fejezetben. Külön foglalkozom a komponens lényeges feladataival: az egyes ontológia modulok függőségeinek kezelésével, a lekérdezéseket kiszolgáló programozói felülettel és a konkurens kérések kiszolgálásával. A komponens megvalósításának folyamatát a 6. fejezetben ismertetem, majd demonstrálom annak helyes működését. A dolgozatot a jövőre vonatkozó tervek, továbbfejlesztési lehetőségek zárják.

1. Szemantikus Web

A World Wide Web szülőatyja, Tim Berners-Lee 1990 októberében elkezdte fejleszteni a *WorldWideWeb* nevű programot. Mai szemmel hihetetlen, de a mára már többmilliárd[1] dokumentumot magában foglaló WWW születése a *httpd* nevű web szerverrel és a névadó hypertext böngészővel kezdődött.

Berners-Lee akkoriban a CERN fizikai kutatóintézetben dolgozott, ahova a világ minden tájáról sereglettek kutatók, mindenki igyekezett magával hozni a saját számítógépét – talán a nagyszámítógépen dolgozók kivételével. Az információ átvitele egyik gépről a másikra a legtöbb esetben külön program írását igényelte. Helyette a problémát egy közös protokoll (HTTP) és közös nyelv (HTML) használatával sikerült megoldania, melyek a weblapok eszperantójává váltak.

A Világháló fejlődése mindenki számára nyilvánvaló, a Szemantikus Web viszont immár három éves, de a fejlesztőin kívül kevesen hallottak róla. Ugyan a magyar sajtóban néha feltűnik egy-egy cikk,[2] személyes tapasztalatom szerint még az informatikusok körében is általában ismeretlen a fogalom. Szintén Tim Berners-Lee-t tekintik megalkotójának, aki 2001-ben víziójának olvasmányos bevezetését jelentette meg az American Scientistben [Berners-Lee01].

Képzeljük el, hogy meg szeretnénk találni egy Erzsébet nevű hölgy honlapját, akivel egy konferencián találkoztunk. Tudjuk róla, hogy cégünk egyik ügyfelénél dolgozik, illetve a fiával egy iskolába jártunk, de a vezetéknevére sajnos nem emlékszünk. Ennyi információ valószínűleg elegendő, mégsem tudunk a jelenlegi, kulcsszavakon alapuló keresők segítségével a nyomára bukkanni. Az ilyen típusú adatokat nem tudjuk leírni a számítógép számára, azt pedig egyelőre végképp nem várjuk el tőle, hogy feldolgozza és értelmezze őket.

A Weben található információk jelentős része természetes nyelvű szöveg, vagy olyan egyéb média (kép, videó, hang), amelyek jelentése a számítógép számára nem feldolgozható. A HTML nyelvet és a hozzá kapcsolódó technológiákat arra tervezték, hogy emberek számára közvetítsen információt. A Szemantikus Web segítségével anélkül szeretnénk megoldani a fenti feladatot, hogy a Csillagok Háborúja droidjainak mesterséges intelligenciájára lenne szükségünk. A természetes nyelvű szövegek értelmezése tehát nem járható út.

Ha azonban a tárolt adatokhoz pontosan definiált jelentést (szemantikát) társí-

---

[1] A Web méretére nincsenek friss statisztikák. A legnagyobb kereső, a Google jelenleg több, mint négy milliárd dokumentumot járt be.
`http://www.viz.co.nz/internet-facts.htm`
Cyveillance, „Sizing the Internet", `http://www.cyveillance.com/web/forms/request.asp?form_type=wp&wp_name=sizing_internet`, 2000. július

[2] `http://index.hu/tech/net/webjovo/`

tunk, a gépi értelmezés és következtetés lehetővé válik. Ilyen, ún. tudásreprezentáción alapuló rendszereket már jóval a Web létrejötte előtt is használtak, azonban centralizált formában: mindenkinek ugyanazokra a jól definiált fogalmakra kellett építkeznie, folyamatosan ügyelve a rendszer konzisztenciájára. A rendszer növekedésével azonban a karbantartás költségei egyre nagyobbak, egy idő után menedzselhetetlenné válnak.

A Szemantikus Web célja hasonló, strukturált információforrást és hozzá tartozó következtetési szabályokat létrehozni, de a Webhez hasonlóan, decentralizált módon. Olyan közös nyelvet – akár többet – kell létrehozni, mellyel mindez leírható, és segítségével a legtöbb létező tudásreprezentáció lefordítható. Így a meglévő tudásbázisok exportálhatóak lesznek a Webre.

Berners-Lee már 2000-ben előrevetítette, hogy ebben a technológiában meghatározó szerepe lesz a XML és RDF nyelveknek, valamint egy logikán alapuló ontológia nyelvnek. Az XML maximálisan hordozható jelölőnyelv, mellyel struktúrát adhatunk a dokumentumokhoz, bár a struktúra jelentéséről semmit nem árul el. Az RDF-fel egyszerű, elemi állításokat fogalmazhatunk meg *(alany, állítmány, tárgy)* hármasok formájában.

A rendszerek közötti átjárhatóság eléréséhez le kell írnunk az építőkövekként használt fogalmak jelentését. Az *ontológia* a fogalmak közötti relációk, kapcsolatok leírására képes, akár egy rendszeren belül, vagy különböző rendszerek terminológiáját összekapcsolva. Az explicite leírt relációkon túl következtetéseket is levonhatunk, a logikai ontológia nyelvek lehetővé teszik *következtetőgépek* használatát, melyekből lekérdezésekkel (*query*) juthatunk hozzá az implicit információkhoz.

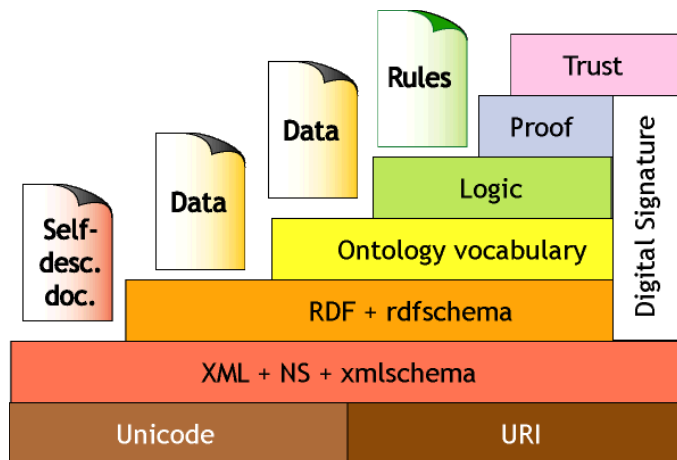
1.1. A Szemantikus Web architektúrája

A Szemantikus Web kapcsán kialakuló szabványokra nagy hatással van a World Wide Web szabványait megalkotó és gondozó szervezet, a W3C.[3] Alapítója, Tim Berners-Lee 2000 decemberében az XML 2000 konferencián[4] körvonalazta a Szemantikus Web funkcionális architektúráját, mely azóta is aktuális és követendő (1. ábra).

Ez a megközelítés funkcionális abban az értelemben, hogy a kifejező erőt nyújtó elemek (*expressive primitives*) inkrementálisan kerülnek bevezetésre a legalsó rétegtől a legfelsőig, így az egyes rétegek – a hozzájuk tartozó nyelvekkel – más-más funkcionalitást hivatottak megvalósítani [Pan01]. Röviden áttekintem az egyes rétegeket és a megvalósításukkor felhasználásra kerülő technológiákat.

[3] World Wide Web Consortium, `http://www.w3.org/`

[4] Tim Berners-Lee prezentációja az XML 2000 konferenciáról: `http://www.w3.org/2000/Talks/1206-xml2k-tbl/`



1. ábra. A Szemantikus Web rétegei

Unicode és URI. Amíg a World Wide Web mögötti technológia alapjául az URL szolgált, vagyis az a gondolat, hogy minden oldalhoz hozzárendelünk egy, az egész világon egyedi azonosítót, addig a Szemantikus Web alapja az URI. Az előbbi fogalom általánosítása, nem csak weblapokhoz, hanem bármilyen erőforráshoz rendelhetünk egyértelmű azonosítókat, statikus leképezést hozva létre. Ezáltal az erőforrások halmaza az alkalmazások számára is navigálható térré válik, melybe az elektronikusan hozzáférhető dolgokon túl beletartozik minden, amiről állításokat szeretnénk megfogalmazni: például tárgyak, személyek, szerepek.

A *Unicode* a dokumentumok alapértelmezett kódkészlete, mely igyekszik magában foglalni a világon használt összes nyelv valamennyi karakterét. Ezáltal globálisan, kultúrától független módon terjeszthetőek a dokumentumok.

XML, névterek, XML sémák. A programok által is könnyen feldolgozható tartalommal feltöltött Szemantikus Web másik alapköve a jól ismert XML[5] technológia. Az információ cseréjéhez szükség van a dokumentumok szintaktikájának olyan globális szabványára, mely architektúrától függetlenül ábrázol strukturált információt. Az XML adatstruktúrája ugyan nagyon kötött (csak fába rendezett elemeket tudunk ábrázolni), de segítségével közvetve már bármilyen, összetettebb szerkezetű adatokat is leírhatunk. Emberek számára is könnyen olvasható, így a dokumentumok áttekinthetőek maradnak.

A *névterek* szintaktikai édesítőszerek, segítségükkel anélkül biztosíthatjuk a dokumentumokban szereplő elemek neveinek egyediségét, hogy azok olvashatóságukból, tömörségükből veszítenének.

Az *XML sémák* a dokumentumokban szereplő adatok szerkezetét írják le, segít-

[5]eXtensible Markup Language, `http://www.w3.org/XML/`

ségükkel definiálhatjuk az XML dokumentumok nyelvtanát – kizárólag szintaktikai szempontból. Ellenőrizhetővé (validálhatóvá) teszik a dokumentumokat.

RDF és RDF séma. Az adatok akkor válnak gépi feldolgozásra alkalmassá, ha képesek vagyunk leírni, mit jelentenek. Az ilyen információkat, adatokról szóló adatokat hívjuk metaadatoknak, ábrázolásuk a Szemantikus Web központi problémája. Például metaadatnak számít, ha megjelöljük egy cikk absztraktját, így feldolgozáskor tartalmi összegzésként kezelhetjük.

Az RDF[6] a Szemantikus Web metaadat rétegének elterjedt, egyszerű modellje, mely csak az *„erőforrás”* és *„tulajdonság”* fogalmakat vezeti be.

A séma rétegben egyszerű Web ontológia nyelveket vezetnek be, melyek képesek a fogalmak és tulajdonságok hierarchiájának leírására (*is-a* reláció). A metaadat réteg általános modelljét használják az ontológia nyelvek alapvető metamodellezési képességeinek definiálásához. Az RDF séma nyelv[7] ezt a szerepet tölti be, melyet 2003. február 10-én W3C ajánlássá nyilvánították.

Ontológiák. Az ontológia egy tudásábrázolási forma, legtöbbször egy adott alkalmazásra vonatkozó tárgyterület fogalmainak – ritkábban általános fogalmaknak –, a köztük értelmezhető relációknak, valamint az ezekből képezhető állításoknak a formális leírása. Az ontológia fogalmát részletesen bevezetem az 1.2. fejezetben, itt a Szemantikus Web ontológia rétegének alaptulajdonságait foglalom össze.

A Szemantikus Web ontológia nyelvei a modellező elemek (*modelling primitives*) sokkal gazdagabb halmazát vonultatják fel, mint az őket megalapozó séma réteg metamodelljei. Általában a bevezetett szemantikák valamelyik leíró logika kifejező erejének feleltethetőek meg, ezzel lehetővé válik az ontológiák átírása leíró logikai kifejezésekké.

A Szemantikus Web jelenleg elterjedt ontológia nyelvei a DAML+OIL (3.1.3. fejezet), utóda, az OWL (3.1.5. fejezet) valamint a KAON (3.1.2. fejezet).

Logikai réteg. Mivel az ontológiák alkalmazásával az adatokhoz pontos szemantikát rendelünk, az implicit információkon túl következtetések levonásával további, explicit tartalmat társíthatunk. A logikai réteg képes összekapcsolni és manipulálni az így kinyert információt.

[6] Resource Description Framework, `http://www.w3.org/RDF/`

[7] RDF Vocabulary Description Language 1.0: RDF Schema, `http://www.w3.org/TR/rdf-schema/`

Bizonyítás. A helyes logikai állítások, következtetések levezetése sok erőforrást igénylő feladat. A következtetés eredményét azonban feltétlenül meg akarjuk osztani másokkal, sőt azt szeretnénk, ha helyességükről a másik fél is meggyőződhetne. Például a Szemantikus Web kooperatív ágensei is így tudnának információt cserélni.

A megosztott tudás igaz voltát azonban csak a hozzá kapcsolódó bizonyítások, levezetések átadásával tudjuk igazolni a másik félnek anélkül, hogy neki is el kelljen végeznie az összes műveletet. A bizonyítás átadásával már csak egy gyors ellenőrzést kell lefuttatnia, hogy valóban nincs a levezetésben logikai hiba. A bizonyítás rétege azokat a jövőbeli nyelveket foglalja magában, melyek levezetések általános leírására alkalmasak.

Bizalom. A végső cél a bizalom elérése, amikor az idegen forrásból szerzett információ igazságtartalmában biztosak lehetünk. Ehhez szükség van a digitális aláírásokra, melyek az előző rétegeken áthaladó információ sértetlenségéről gondoskodnak. A gyakorlatban a következtetőgépet az aláírás hitelesítését végző rendszerrel kell összekapcsolni. A dokumentumok nem csak állításokból fognak állni, hanem leírják, mely állításokat kik írták alá, kik szavatolják helyességüket.

1.2. Mi az ontológia?

Az ontológia fogalma a filozófiából származik, a létező szisztematikus, tapasztalati úton történő kategorizálásával foglalkozó tudományra, a lételméletre utal. A mesterséges intelligencia területén valamely, tipikusan háttértudást igénylő feladat tárgyterületének fogalmi világát modellező leírás. Az ontológia (*ontology*) szót régebben meglehetősen következetlenül, számtalan értelemben használták. Ezeket világosan elhatárolja egymástól Guarino cikke, mely a filozófiai diszciplínán kívül még hét különböző jelentést ismertet [Guarino95].

Hagyományosan a mesterséges intelligenciában a tudásábrázolás során a funkcionalitásra és a következtetésre fektetik a hangsúlyt. Az ontológia ezzel szemben a modellezést helyezi előtérbe: nem csak önmagában a problémát, hanem a kontextust és a megoldási folyamatot modellezhetjük segítségével [Kankaanpää99]. A problémamegoldás szempontjából lényeges tárgyterület terminológiáját rögzítjük, a fogalmak és relációk mindenki számára egyértelmű felfogását adjuk meg explicit formában az ontológiában, ami egyébként implicit módon minden tudásalapú rendszernek része.

Az ontológia jelölhet informális fogalmi rendszert, ahol az egyes fogalmak vagy egyáltalán nincsenek definiálva, vagy csak természetes nyelvű állítások segítségével fogalmazzuk meg őket. A formális ontológia ezzel szemben egy tárgyterület leírására

alkalmas mérnöki termék, amelyben a szavakhoz (fogalmakhoz) explicit jelentést társítunk, és ezt a jelentést formálisan (például formális logika használatával) ábrázoljuk. A továbbiakban ontológián az utóbbi, formális fogalmi rendszert értem.

Az ontológiák fő építőkövei a **fogalmak** (*concepts*) és a **relációk** (*relations*). A fogalmak merev, állandó tulajdonságot kifejező típusokat (pl. *férfi*, *nő*) vagy változó szerepeket (pl. *tanuló*) jelölhetnek. A típusokat és a szerepeket az ontológia nyelvek azonban sokszor nem különböztetik meg. A relációk alkalmasak a fogalmak összekapcsolására (pl. egyetem *tanulója* tanuló). Legtöbbször csak bináris relációkat használunk, mivel jól áttekinthetőek és elegendőek a többes kapcsolatok ábrázolására is – ha szükséges, a többes reláció helyett külön fogalmat vezethetünk be. Sokszor célszerű bevezetni a **példányokat** (*instances*) is, melyek a fogalom osztályok konkrét egyedeinek tekinthetők. A példányok modellezhetőek a fogalmak hierarchiájának alsó szintjével is, ezért néha eltekintenek a megkülönböztetésüktől.

Az eddigieket a jól ismert taxonómiákról is elmondhatnánk, az ontológiák modellező képessége azonban tágabb. A taxonómiák ugyanis csak a klasszifikációt, az öröklődés relációt használják a fogalmak leírására, míg ontológiák esetén tetszőleges számú, különböző relációt felvehetünk. Minden ontológia egy nézeteként származtatható tehát az a taxonómia, melyet a fogalmi hierarchia figyelembe vételével képezünk.

Ontológiák fő alkalmazási területei. Az ontológiák bevezetését leginkább az az igény motiválta, hogy lehetővé tegyék a tudás megosztását a közös nyelv, illetve a kommunikáció megvalósításához. Elosztott ágens rendszerekben az ügynökök olyan üzenetekkel kommunikálhatnak, melyek a közös ontológiában specifikált fogalmakra hivatkoznak.

A vállalati ontológiák egy vállalat különböző szempontok szerint felépített modelljeit integrálják. A vállalati modellezés célja a teljes szervezetet átfogó modell kialakítása, mely elősegíti az üzleti tervezést, a kommunikációt és a vállalaton belüli eszközök közötti átjárhatóságot. Kifejezetten ehhez az igényhez kialakított ontológia rendszerről, a *KAON Tool Suite*-ról majd a 3.1.2. fejezetben írok.

Az ontológia számunkra is lényeges alkalmazási területe az információ kinyerés, az Internetes forrásokon alkalmazott információ visszakeresés területe. A hagyományos, napjainkban használt kereső szolgáltatások a webes dokumentumokat nem adekvát módon, az emberi felhasználásra tervezett, természetes nyelvű szövegeken futtatott statisztikai módszerek segítségével hasonlítják össze. Nem rendelkeznek a szavak szemantikus értelmezéséhez szükséges fogalmi rendszerrel, aminek az alapjául éppen az ontológiák szolgálhatnának.

2. Az IKF rendszerprototípus

Amint már a bevezetőben említettem, a diplomaterv célja az IKF kutatási projekt keretében a fejlesztett prototípus rendszer ontológiát beágyazó komponensének tervezése és megvalósítása. Ebben a fejezetben bemutatom az IKF projektet, céljait és a keretrendszer legfontosabb tulajdonságait.

Az Információ és Tudás Tárház (angolul *Information and Knowledge Fusion*, IKF) nemzetközi EUREKA projekt célja olyan intelligens tudás beszerző, elemző és kezelő rendszerek tervezése és implementálása, amelyek különböző, meghatározott alkalmazási területeken nyújtanak hatékony segítséget a tudásmenedzsmentben és az üzleti intelligencia megvalósításában [Varga03]. Az IKF célterületei többek között a pénzügyi (bankok és biztosítótársaságok), egészségügyi és oktatási szféra.

A projekt magyar résztvevőinek (IKF-H projekt, többek között a BME, a Morphologic Kft. és a Multilogic Tanácsadó és Informatikai Kft.) feladata tudásalapú információ-visszakereső rendszer kifejlesztése a pénzügyi szektor számára. A rendszer a különböző elektronikus forrásokból (Internetről, intranet erőforrásokból, adattárházakból) beszerzett adatokat elemzi, majd strukturált formában szolgáltatja a felhasználók felé.

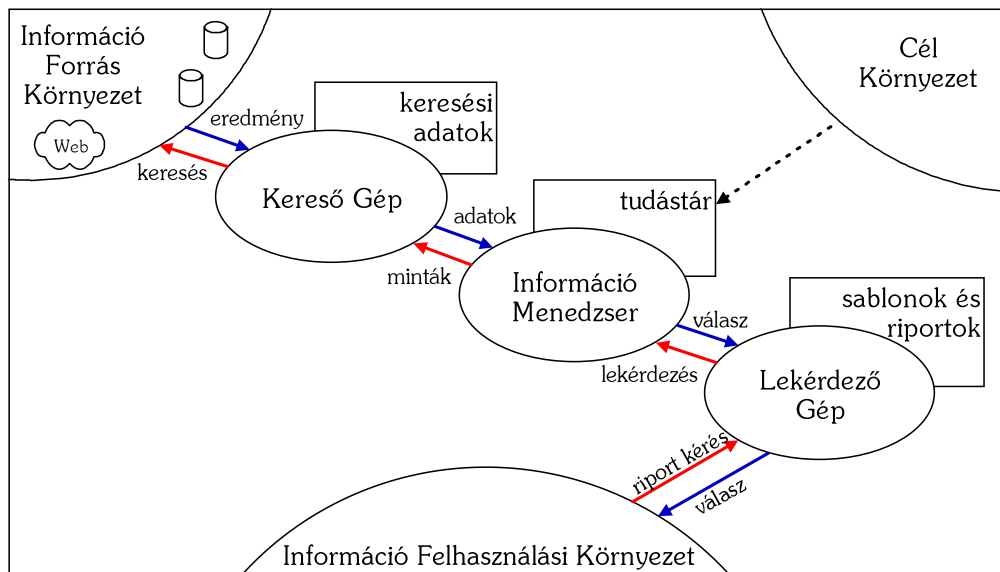
Az információ-visszakereső rendszer a projekt során mint *IKF keretrendszer* valósul meg. Ebből a konkrét problémák megoldásához, a feladatoknak megfelelő testreszabás után születnek meg speciális példányai, az *IKF alkalmazások*, melyek tükrözik a szóban forgó IKF architektúrát.

A következőkben áttekintem a magyar projekt keretein belül létrejövő IKF keretrendszer céljait, a rendszer környezeti modelljét és szerkezeti felépítését. Mivel a diplomamunka célja a rendszeren belül a tárgyterület modellezésére használt ismeretanyag kezelése, áttekintem, melyek a háttértudás ábrázolásának lehetőségei. Ismertetem a tárgyterületi tudás szerepét az IKF-ben, majd megindokolom, miért az ontológia a megfelelő reprezentációs séma annak leírására. Végül ismertetem, hogyan illeszkedik az ontológia kezelő komponens a teljes rendszerbe.

A magyar IKF projekt célja. Az IKF-H projekt elsősorban pénzügyi cégek, bankok és biztosító társaságok számára fejleszt információ-beszerző és -elemző keretrendszert. Az IKF rendszer strukturálatlan vagy részben strukturált információforrásokból nyer ki téma-specifikus információkat, majd ezeket kérésre strukturált formában nyújtja a felhasználóknak. Például cégek alapadatainak (neveinek, részvényük nevének stb.) kézi bevitele után gazdasági témájú cikkek alapján képes megbecsülni egy cég piaci pozícióját, vagy felsorolni az utolsó két hét legfontosabb

történéseit. Az adatokhoz tipikusan a kulcsszavas keresés mellett felhasználói űrlapok kitöltésével lehet hozzáférni, melyekre a rendszer generált riportok formájában válaszol.

A fenti célok megvalósításához összetett részfeladatokat kell megoldani. A tudáskinyerés és keresés támogatásához szemantikus modelleket kell létrehozni, amihez a pénzügyi információs világ mélyreható elemzése szükséges. Megfelelő információ kinyerési, következtetési és tudásábrázolási technológiákat kell választani, illetve ezek működését integrálni kell.



2. ábra. Az IKF rendszer felépítése

A rendszer felépítése. Az IKF rendszer környezet modellje három fő részre bontható [Mészáros01]. A *cél környezet* a tárgyterülethez kapcsolódó tudás fizikai forrásait, objektumait tartalmazza. A rendszer működéséhez szükséges tudásmodell ennek modellezésével jöhet létre. Az *információ forrás környezet* tartalmazza azon elektronikus forrásokat, melyekben rejlő információt a felhasználóknak nyújtani kell. Az *információ felhasználási környezet* a felhasználókat és a rendszer karbantartását, finomhangolását végző rendszermenedzsereket foglalja magába.

A felhasználó lekérdezésére adott válasz a 2. ábrán látható információs folyamnak megfelelően, a rendszer három alrendszerén keresztül érkezik vissza a felhasználási környezetbe. A *kereső gép* közvetlenül a forrás környezetből szerzi be azokat a dokumentumokat, melyek alkalmasak releváns adatok kinyerésére. A dokumentumokat nem eredeti formában adja tovább, hanem strukturális elemzéseket, előfeldolgozást végez.

Az *információ menedzser* végzi a klasszikus információ kinyerési feladatot: a be-

szerzett dokumentumok alapján tudásbázist épít, mely már strukturált formában tárolja az adatokat. Egyik környezettel sem áll közvetlen kapcsolatban, de az információ kinyeréshez és a tudástár kezeléséhez a cél környezet minél pontosabb modelljére van szüksége. A modell kezelése, tartalmának következtetőgépen keresztüli szolgáltatása a diplomamunka termékeként született ontológia kezelő modul feladata.

A *lekérdező gép* tart közvetlen kapcsolatot a végfelhasználókkal. Értelmezi a felhasználói kéréseket, majd az információ menedzser válaszai alapján emberek számára is jól áttekinthető riportokat generál.

Az IKF rendszer célja tehát a Weben, vagy hasonló dokumentumtárakban megbúvó információ gazdag kereshetőségének biztosítása. Azonban a Szemantikus Web szemléletével ellentétben nem a bemeneti dokumentumok jellegének reformjával akarja ezt elérni. Helyette a meglévő források intelligens indexelésével saját maga állítja elő a strukturált, metainformációkban gazdag adatokat. Ezáltal a napjainkban is hozzáférhető temérdek információt a gépi feldolgozásra is alkalmas belső ténybázisban tárolja, amin már pontos lekérdezések futtathatók.

2.1. A háttértudás tárolásának lehetőségei

A háttértudás olyan előzetes ismeretek gyűjteményét jelenti, melyek segíthetik a kitűzött információfeldolgozási feladat hatékonyabb és pontosabb elvégzését. Az IKF keretrendszerben mind a funkcionális jellegű általános, mind a cél környezetről, az alkalmazási területről szóló specifikus tudás az IKF alkalmazás részeként jelenik meg, az információ kinyerést és elemzést segítő háttértudás [Varga03].

Ennek az ismeretanyagnak a tárolását egy tudásábrázolási séma segítségével tudjuk megoldani. A tudásábrázolás fő feladata, hogy a problémára vonatkozó minden fontos információt le tudjon írni, és könnyen hozzáférhetővé tegye a problémát megoldó folyamat számára. Fő sémái a következők [Kankaanpää99]:

logikai reprezentációs séma: formális logika segítségével írják le a tudást és a következtetési szabályokat;

procedurális reprezentációs séma: az információt parancsok halmazaként tárolják, a többi reprezentációval ellentétben nem deklaratív leírások, hanem arra koncentrálnak, hogyan oldhatjuk meg a problémát;

hálózati reprezentációs séma: gráf alapúak, ahol az egyes fogalmakat a gráf pontjaiként ábrázoljuk, az élek a köztük fennálló relációkat jelölik – ember számára könnyebben kezelhető, vizuális reprezentációk;

strukturált reprezentációs séma: az előző kiterjesztései, ahol a fogalmakhoz

részletesebb adatstruktúrákat csatolhatunk – értékeket, hivatkozásokat más fogalmakra, vagy akár procedurális információt.

A tudáskinyerés folyamatát és a tárolt információk értelmezését támogató reprezentációt a feladat jellegének megfelelően kell megválasztanunk. Előtte megvizsgáljuk, milyen szempontok játszanak szerepet a mérlegeléskor, hol milyen követelményeket támasztunk a tudásábrázolással szemben. Randall a reprezentációk legfontosabb szerepeit, betöltendő feladatait az alábbiakban foglalja össze [Randall93]:

helyettesítő: a reprezentáció feladata, hogy helyettesítse a külső világban előforduló dolgokat belső megfelelőkkel, amiken a következtetések végrehajthatóak;

ontológiai elkötelezettség: az ontológiai elkötelezettség határozza meg, milyen megközelítésben, milyen részletességgel reprezentáljuk a világ egyes részeit;

részleges következtetési elmélet: a tudásreprezentációk általában az emberi gondolkodás mibenlétéről, működéséről megfogalmazott feltételezésen alapulnak, valamilyen – akár csak implicit – következtetési mechanizmust mindig magukban hordoznak;

eszköz a hatékony számításra: a következtetés számítási feladat, a reprezentáció feladata olyan hatékony modellezést nyújtani, mely segítségével a következtetés megoldható;

emberi kifejezésmód: az emberi kifejezés eszköze, ezért a megfogalmazott tudást emberek számára is olvasható, áttekinthető, karbantartható és módosítható módon kell tárolnia.

A tárgyterületi tudás ábrázolásánál elengedhetetlen a pontos szemantika és a hatékony következtetési mechanizmusok megléte. Mivel a háttértudás a cél környezetet modellezi, lényeges, hogy a reprezentáció képes legyen a valóságot a részletezettség különböző szintjein is leírni, hogy megalkotásakor az egyes részeit azok relevanciája szerint vehessük figyelembe.

Mivel a tárgykörnyezetet minden IKF alkalmazásra külön-külön le kell írni, és ennek a modellnek az előállítása előre láthatólag költséges és körülményes feladat, a vizualizáció is előtérbe kerül. A modell létrehozásakor és karbantartásakor annak emberközeli, áttekinthető ábrázolhatósága elősegíti a pontosabb modellezést, mivel kisebb az emberi hiba valószínűsége. A tisztán logikai és a procedurális rendszerek ezt egyáltalán nem, vagy csak kis mértékben támogatják.

A háttértudás tárolására az IKF projekt a logikai kifejezőerővel is bíró, mégis strukturált, így jól vizualizálható reprezentációs séma, az ontológia alkalmazása mellett döntött. A választás további előnye, hogy a Szemantikus Web térhódításával hosszú távon lehetővé teszi annak rendszerei és az IKF alkalmazás közti könnyebb átjárhatóságot.

Az ontológia nyelvek közül a hozzájuk kapcsolódó technológiák ismertetése után, a 3. fejezet végén választom ki a feladathoz illőt.

2.2. Az ontológia szerepe az IKF-ben

Az IKF rendszerben az ontológia feladata az információkinyerés támogatása, illetve a begyűjtött információ tárolásának, későbbi értelmezésének lehetővé tétele.

Az információ forrás természetes nyelvű szöveg, így értelmezéséhez a nyelvtani előfeldolgozás (szegmentálás, mondatelemzés) után a szóalakokhoz (szótövekhez) fogalmakat kell társítani, valamint a fogalmak nyelvtani viszonyát szemantikus kapcsolataik által lehet tényekké alakítani. Mindkét lépés az ontológia és a ténybázis együttese segítségével lehetséges.

A természetes nyelvű forrásszövegből logikai állításokat szeretnénk kinyerni, azonban a szöveg nyelvtani elemezhetősége a magyar nyelv sokrétűsége miatt nehézkes. A folyamat támogatásához így a manuálisan készített, *a priori* információnak a lehető legpontosabb szemantikus reprezentációval, gépi értelmezhetőséggel kell bírnia. Az ontológiai formalizmus esetében erős logikai támogatással, pontos szemantikával rendelkező nyelvet kell választanunk.

3. Ontológia technológiák

Az IKF keretrendszer tárgyterületi tudásának ábrázolására és kezelésére választanunk kell az ontológiával kapcsolatos technológiák közül. A legfontosabb, hogy elkötelezzük magunkat egy konkrét ontológia nyelv, vagy egymással kompatibilis nyelvek mellett. Ezért először bemutatom a napjainkban elfogadott ontológia nyelveket, kialakulásuk szerinti sorrendben.

A háttértudás elkészítésének és karbantartásának eszközeit, az ontológia szerkesztőket is felvonultatom, majd az ontológiák beolvasását, manipulálását lehetővé tevő programozói könyvtárakat ismertetem.

Végül kiválasztom a feladathoz legjobban illő ontológia nyelvet, javaslatot teszek az IKF szempontjainak leginkább megfelelő szerkesztőre és kiválasztom a kezelő komponens belső ontológia manipulációs könyvtárát.

3.1. Ontológia nyelvek

A fejezet során áttekintem a jelenleg elterjedt ontológia nyelveket: a már 1998 óta készülő RDF(S)-t, majd a rá épülő KAON, DAML+OIL és OWL nyelveket, valamint bemutatok egy tetszőleges formalizmus használatát lehetővé tevő platformot (DL-workbench). A fejezet végén értékelem a felsorolt lehetőségeket, kiválasztva az IKF ontológia kezelő modulja által támogatott reprezentációt.

3.1.1. RDF és RDF sémák

Az RDF adatmodell metaadatok definiálását és feldolgozását teszi lehetővé, a W3C szabvány elkészítését már 1997-ben megkezdték. Az URI-kkal azonosított erőforrásokra vonatkozó nevesített tulajdonságokat és tulajdonság értékeket képes leírni. Adatmodellje alapvetően három objektum típust ismer [Pan01]:

erőforrások *(resources)* Amint azt már az 1.1. fejezetben az URI-k bevezetésénél leírtam, bármi lehet erőforrás, amiről állításokat kívánunk megfogalmazni. Azonosításukra az URI-kat használjuk az RDF dokumentumon belül is.

tulajdonságok *(properties)* Az erőforrások valamely aspektusát, attribútumát, kapcsolatát azonosítják. Ezek is erőforrások, megnevezésükre az állítások leírásánál van szükség.

állítások *(statements)* Valamely erőforrás egy megnevezett tulajdonsággal és a tulajdonsághoz tartozó értékkel együtt alkot egy RDF állítást. Tulajdonképpen egy rendezett hármas *(erőforrás, tulajdonság, érték)* alakban, ahol a részekre

3. ábra. Az RDF gráfos ábrázolási módja

mint az állítás *alanyára*, *állítmányára* és *tárgyára* is szoktak hivatkozni. A tárgy lehet erőforrás vagy literál értékű is.

Az RDF tehát egy olyan mechanizmus, amely az állításoknak megfelelő hármasok kezelésére képes. Ezeket a hármasokat több módon is ábrázolhatjuk. Leírhatjuk bináris predikátum formájában, például az alábbi állítás szerint én az IKF projekt résztvevője vagyok:

```
resztvevo(http://ikf.mit.bme.hu/,
        http://home.mit.bme.hu/~fandrew/)
```

Használhatjuk a 3. ábrán látható vizuális, sokkal áttekinthetőbb irányított gráfos ábrázolási módot, vagy az RDF átvitelére használt szabványos XML formátumot:

```
<rdf:Description rdf:ID="http://ikf.mit.bme.hu/">
    <ikf:resztvevo rdf:ID="http://home.mit.bme.hu/~fandrew/"/>
</rdf:Description>
```

Az RDF adatmodellje önmagában tehát alkalmas attribútumok (literál tárgyú állítások) illetve relációk (erőforrás tárgyú állítások) leírására. Gyenge tipizálásra is képes: a beépített *rdf:type* tulajdonság segítségével egy erőforráshoz hozzárendelhetünk egy másikat, vagy egy literált, mint az ő típusa. Ez a képesség abban az értelemben gyenge, hogy az *rdf:type* relációnak nincs a szabvány által definiált értelmezése (szemantikája).

További erőssége a reifikáció támogatása: állításokat fogalmazhatunk meg állításokról, mivel az RDF állításai önmagukban is erőforrások. Az így konstruált kijelentéseket magasabb rendű állításoknak nevezzük, melyek értelmezése nehezen megoldható: logikai átfogalmazásukhoz magasabb rendű logikákra van szükség, melyek alkalmazása a következtetési képességekkel szembeni megalkuvásokat kíván. Az RDF megalkotásakor azonban tudatos tervezési döntés volt, hogy egy RDF dokumentum bármit le tudjon írni [Haarslev03a]. A kifejezőerő szűkítésére és pontosabb, gazdagabb szemantika hozzáadására az RDF sémákra építkező logikai ontológia nyelv definiálásával van lehetőség, amire több példát is fogunk látni a fejezet hátralevő részében.

RDF sémák. Az RDF alapvető modellezési képességeit egészíti ki séma nyelve, az RDFS. A két technológiára együtt RDF(S)-ként hivatkoznak. Fogalmakat és példányokat rendelhetünk össze a jól ismert *is-a* reláció mentén, az erőforrások leírására tulajdonságokat definiálhatunk, valamint a fogalmak és tulajdonságok hierarchiába rendezhetőek.

Minden fogalom az *rdfs:Class* beépített osztály példánya, az RDF erőforrásai az *rdfs:Resource* osztályba tartoznak. Az osztályba tartozás jellemzésére az *rdfs:type*, az osztály- és tulajdonsághierarchiát az *rdfs:subClassOf* illetve *rdfs:subPropertyOf* jelöli.

A sémák modellezési képességeit szemléletesen az objektum-orientált (OO) programozási nyelvekkel mutathatjuk be. Az osztályok és példányok jelentése nagyon hasonló, bár a sémák a legtöbb OO nyelvvel ellentétben lehetővé teszik a többszörös öröklődést és a többszörös tipizálást (egy példány több osztályhoz tartozik) is. Lényegi különbség mutatkozik az OO nyelvek attribútum-orientált és a sémák tulajdonság-centrikus szemlélete között, ahogy az a 4. ábrán látható.

```
// osztályok
class Stílus {
        String neve;
}

class ZeneiStílus extends Stílus {
}

class Zeneszerző {
        ZeneiStílus zeneiStílusa;
}

// példányok
ZeneiStílus klasszikus
        = new ZeneiStílus();
Zeneszerző Mozart = new Zeneszerző();
Mozart.stílusa = klasszikus;
```

(a) Java

```
<!-- osztályok -->
<rdfs:Class rdf:id="Stílus"/>
<rdfs:Class rdf:id="ZeneiStílus">
        <rdfs:subClassOf rdf:resource="#Stílus"/>
</rdfs:Class>
<rdfs:Class rdf:id="Zeneszerző"/>

<!-- tulajdonságok -->
<rdf:Property rdf:id="StílusNeve">
        <rdfs:domain rdf:resource="#Stílus"/>
        <rdfs:range rdfs:Literal/>
</rdf:Property>
<rdf:Property rdf:id="zeneiStílusa">
        <rdfs:domain rdf:resource="#Zeneszerző"/>
        <rdfs:range rdf:resource="#ZeneiStílus"/>
</rdf:Property>

<!-- példányok -->
<ZeneiStílus rdf:id="klasszikus"/>
<Zeneszerző rdf:id="Mozart">
        <zeneiStílusa rdf:resource="#klasszikus"/>
</Zeneszerző>
```

(b) RDFS

4. ábra. Az RDFS és az objektum-orientált nyelvek kapcsolata

Az objektum-orientált nyelvekben az osztályok leírására attribútumokat definiálhatunk, melyek aztán a példányok szintjén értékeket kaphatnak. Az attribútumok azonban az osztálydefiníciókkal szorosan összetartoznak. Az RDF sémák esetében a tulajdonságokhoz rendeljük az osztályokat a megadható kényszerek (*rdfs:ConstraintProperty*) segítségével. A leggyakrabban használt kényszerek az értelmezési tartomány (*rdfs:domain*) és értékkészlet (*rdfs:range*). A tulajdonságok a sémák esetén az osztályok szintjére emelkednek: míg például Java-ban (4(a). ábra)

a zeneszerző stílusát az osztálydefinícióban kell bevezetnünk, RDFS esetén a tulajdonság definíciója független.

A tulajdonságok függetlenítése segíti a Szemantikus Web céljaként megfogalmazott elosztott tudásábrázolás elérését, mivel segítségével anélkül bővíthetjük ki egy fogalom jelentését, hogy módosítanunk kellene a definícióját.

3.1.2. KAON ontológia nyelv

A KAON (*KArlsruhe ONtology*) a németországi Universität Karlsruhe AIFB Intézetének[8] és az FZI kutatóintézet tudásmenedzsment kutatócsoportjának[9] együttműködésével jött létre, a *The Karlsruhe Ontology and Semantic Web tool suite* nevű eszközkészlet saját ontológia nyelve.

A *KAON tool suite*-ot azzal a céllal hozták létre, hogy vállalati környezetben létrehozandó tudásbázisok ontológia hátterének kialakítását tegye lehetővé. Egymásra épülő modulokból áll, melyek így magukban foglalják a szükséges architektúra számtalan elemét: RDF alapú perzisztens adatbázist (*RDF Server*), KAON tudásbázis szervert (*KAON Server*), csoportos ontológia fejlesztés támogatást (*Engineering Server*), ontológia szerkesztőt (*OI-modeler*), ontológia portált (*KAON Portal*) [Oberle04].

A nyelv lehetővé teszi osztályok (fogalmak), relációk (tulajdonságok) és példányok (egyedek) definiálását. Az osztályokat és a relációkat hierarchiába rendezhetjük, a többszörös öröklődés megengedett. A tulajdonságokhoz értelmezési tartományt és értékkészletet rendelhetünk, ezek a hierarchia mentén csak szűkülhetnek, valamint a tulajdonságokat példányosíthatjuk az egyedek közti kapcsolatok ábrázolásához. Lehetőségünk van továbbá moduláris ontológia létrehozására: az egyes modulokba másikakat importálhatunk, és az így létrejövő összefésült ontológián futtathatunk lekérdezéseket, vagy következtethetünk.

A nagyvállalati alkalmazhatóság érdekében a KAON ontológia nyelv tervezését más szempontok vezérelték, mint a következtetésen alapuló ontológia nyelvekét [Motik02]. Az üzleti alkalmazások igényeinek megfelelően a figyelembe veendő technikai követelmények a skálázhatóság, konkurens hozzáférés támogatása, megbízhatóság és a könnyű integrálhatóság már meglévő adatforrásokkal. A megfelelő teljesítmény elérése érdekében nem egy leíró logikából és a mögé felépített következtetési algoritmusokból indultak ki, annak ellenére, hogy ennek már akkor részletes

[8]Institut für Angewandte Informatik und Formale Beschreibungsverfahren
`http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/WBS/english`

[9]Forschungszentrum Informatik, Die Forschungsgruppe Wissensmanagement
`http://www.fzi.de/wim/eng/`

irodalma volt. Elvetve ezt a lehetőséget csak egyszerű következtetési mechanizmusokat használtak (*light-weight inferences*), mely csak alapvető, axiómákon alapuló, de kezelhetőbb tulajdonságokat tesz lehetővé. Ezek jelenleg a szimmetrikus, tranzitív és inverz relációk, melyek lényegében egyszerű gráf-algoritmusok futtatásával kezelhetőek.

A KAON ontológia nyelv első implementációja az RDF(S)-en alapul, mely nem véletlen: segítségével a KAON könnyen ábrázolhatóvá válik. Az RDFS legtöbb elemét átvették, de természetesen több megszorítást és pontosabban definiált szemantikát alkalmaztak.

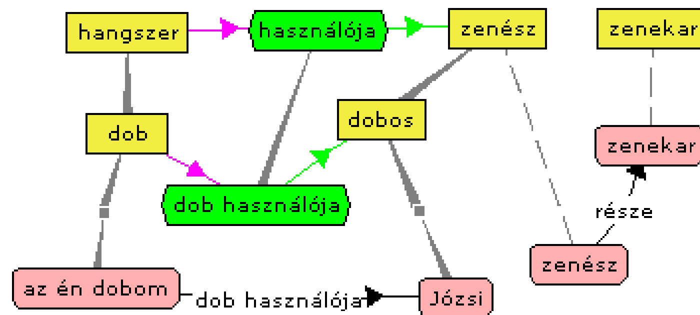
Metamodellezés. Az RDFS-t bírálták a végtelen metamodellezési képességei miatt, melyek egyrészt nehezen értelmezhetővé teszik [Pan01], másrészt bizonyos körülmények között még Russel-paradoxon kialakulásához is vezethetnek.[10] Modellezési primitíveket használ (pl. *rdf:type*, *rdf:domain* stb.), melyekkel további modellezési primitívek definiálhatók, így akár ontológia nyelvet is definiálhatunk a segítségével. Azonban a saját primitívei nehezen értelmezhető relációban állnak egymással: az *rdf:Class* (osztály) leszármazottja az *rdf:Resource*-nak (erőforrás), miközben az *rdf:Resource* maga is az *rdf:Class* példánya. Hasonlóan furcsa, hogy az *rdf:Class* önmaga példánya. Ennek kiküszöbölésére vezették be az RDFS(FA) nevű fix négy rétegű metamodellezési architektúrát, mely élesen elkülöníti egymástól a metanyelvet, az ontológia nyelvet, az osztályokat és a példányokat.

A KAON kevésbé szigorú megszorítással él, három rétegű architektúrát vezet be (metamodell, ontológia, példányok). A metamodell elhatárolódik, így primitívei nem érhetőek el a modellből – ellentétben az RDFS-sel, ahol hivatkozható például a hierarchiát jelölő *rdfs:subClassOf* entitás.

A nyelv szemantikája szerint az interpretációt nem entitásokhoz rendelték, hanem az egyes osztályokhoz, példányokhoz illetve tulajdonságokhoz. Ezeknek a halmaza viszont nem feltétlenül diszjunkt: csak az osztályok és tulajdonságok halmaza határolódik el. Egy entitás tehát lehet egyszerre osztály és példány, ilyenkor azonban két külön interpretáció tartozik hozzá. Bevezették a pszeudo-példány (*spanning instance*) fogalmát, mely ezek kapcsolatát fejezi ki. A nyelv modellezési képességeit szemlélteti az 5. ábrán látható ontológia részlet.

A metamodellezés szükségességére lássunk egy példát: próbáljuk meg leírni a „zenész", „hangszer" és „cselló" szavak jelentését. A zenész olyan művész, aki hangszeren

[10] *Russel-paradoxon:* legyen R az önmagukat nem tartalmazó halmazok halmaza. Kérdés: R tartalmazza-e önmagát? Ha megfogalmazhatóak ilyen és ehhez hasonló kérdésfelvetések, nehezebb logikai következtetéseket levonni.



5. ábra. A KAON entitásainak bemutatása

A fogalmakat sárga, a tulajdonságokat zöld, a példányokat rózsaszín doboz jelöli. A szaggatott nyilak a fogalmakhoz a pszeudo-példány megfelelőiket kötik.

játszik. A hangszer itt attribútum, a zenészhez egy tulajdonság-egyed köti, szükségszerűen példány. A cselló viszont a hangszerek halmazának egy egyede, kézenfekvő, hogy a hangszer osztály példánya legyen. A két interpretáció közötti különbség az eltérő nézőpontban keresendő, más értelmezés tartozik hozzájuk.

Ez a metamodellezés az ontológia értelmezésével kapcsolatos problémát vet fel. Nem tudjuk megválaszolni azt a kérdést, hogy „Mit jelent a hangszer entitás?", csak a konkrétabb „Mit jelent a hangszer, mint fogalom?" kérdésre tudunk válaszolni. Mindig csak a modell valamelyik rétegét tudjuk értelmezni, összességében az egész modellt nem. Persze válthatunk a különböző rétegek között, egy entitás egyik interpretációjáról a másikra, de köztük a KAON egyelőre semmilyen szemantikát nem definiál. (Később tervezik a nyelv ilyen irányú bővítését, például alapértelmezett értékek bevezetével.[11])

Lexikai tartalom illesztése. A KAON rendszerben a nyelvi modell struktúráira épül egy lexikai réteg, mely speciális entitások használatával („*System Objects*") lehetővé teszi úgynevezett „*lexical entry*"-k, szöveges bejegyzések hozzáfűzését az egyes entitásokhoz. Ezek a címke (*label*), mely az entitás megjelenített neve, a szótövek (*stem*), a szinonimák (*synonym*) és a dokumentáció (*documentation*). Az utóbbi három tetszőleges célra felhasználható, egyelőre a KAON részéről nem kötődik hozzájuk egyedi funkcionalitás. A modell tehát a lexikai elemeket is magában foglalhatja, melyet más nyelvek esetében magunknak kell megoldanunk.

A KAON további előnye, hogy rendelkezik programozási felülettel (*KAON API*), magas szintű Java objektumok segítségével böngészhetjük és szerkeszthetjük az ontológiát.

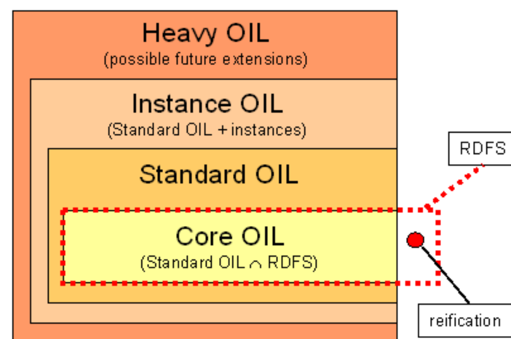
[11] Document on KAON Supported Modeling Primitives and Design Rationales, October 2002 `http://kaon.semanticweb.org/Members/motik/KAON-SupportedPrimitives_DesignDecisions.pdf`

3.1.3. DAML+OIL

A DAML+OIL ontológia nyelv két független projekt találkozásaként, fúziójaként jött létre. A DAML[12] a *DARPA Agent Markup Language* rövidítése, az Amerikai Védelmi Minisztérium (DARPA) fejlesztette ki. Az OIL[13] az *Ontology Inference Layer* rövidítése, egy független kezdeményezés Web alapú reprezentációs és következtetési képességekkel bíró ontológia kifejlesztésére. Először bemutatnám a DAML+OIL gyökereit, „szüleinek" azon tulajdonságait, melyekből sokat meríthettek az újabb ontológia nyelvek alkotói.

OIL. Az OIL számos egyetem kutatóinak közös munkájaként született az 5. Európai Keretprogram IBROW és On-To-Knowledge projektjeinek támogatásával. Tervezésekor három terület eredményeit próbálták ötvözni annak érdekében, hogy létrehozzák a Szemantikus Web általános célú jelölőnyelvét [Fensel02]. Ezek a keret alapú rendszerek, a leíró logikák és az XML és RDF(S) nyelvek (az OIL az RDF(S) kiterjesztése).

Míg az XML és RDF alkotják a Szemantikus Web adatrétegét, az RDFS a séma réteget, az OIL az RDFS segítségével definiált logikai réteg: formális szemantikát vezet be és lehetővé teszi a formális következtetést.



6. ábra. Az OIL dialektusok és az RDFS viszonya

A nyelv négy elkülönülő dialektusát definiálták (6. ábra), így alkalmazásai nagyban eltérhetnek az igényelt kifejező erő illetve a rendelkezésre álló erőforrások tekintetében. A *Core OIL* az RDFS leszűkítése: a reifikációt teljes mértékben mellőzték a nyelvből a hatékony következtetési algoritmusok alkalmazhatósága érdekében. A *Standard OIL* a leíró logikák azon legbővebb osztályának felel meg, melyre még léteznek teljes és hatékony következtetők. A további két réteg a példányok kezelését (*Instance OIL*), illetve azóta sem specifikált, mélyebb kifejező erejű szemantikát (*Heavy OIL*) ad hozzá a nyelvhez.

[12] `http://www.daml.org/`

[13] `http://www.ontoknowledge.org/oil/`

Megszületésével párhuzamosan kezdetét vette az OILedit ontológia szerkesztő fejlesztése, mely az OilEd (3.2.1. fejezet) elődje.

DAML. A DAML kezdeményezés célja az SGML és a HTML után egy újabb jelölőnyelv létrehozása, mely lehetővé teszi a Weben található információ gépi feldolgozását, azaz a Szemantikus Web létrejöttét.

Kezdetektől fogva a már sok erőfeszítés eredményeként született technológiákra próbáltak építkezni, mint az XML, RDF, OIL és a HTML kiterjesztéseként született SHOE[14]. A 2000 októberében született kezdeti specifikáció[15] az OIL-hoz hasonlóan az RDF-en alapult, majd a két projekt 2000 decemberében egyesült.

Logikai kifejezőerő. Az ontológiák kézi elkészítése egyszerű és gyors módja az ontológiaépítésnek, de az ilyen megközelítések a méretbeli növekedéssel tarthatatlanná válnak. A DAML+OIL nyelvvel lehetővé vált, hogy a leíró logikák segítségével az ontológia építése során annak igazságát, teljességét és konzisztenciáját automatikusan ellenőrizzük. Stevens és munkatársai sikeresen alkalmazták a nyelvet a GONG[16] projekt keretében, ahol géneket leíró ontológiát írtak át a fogalmak tulajdonságainak logikai leírását is támogató DAML+OIL nyelvre [Stevens03].

A DAML+OIL nyelv az RDF(S)-hez hasonlóan osztályokat és tulajdonságokat képes leírni állítások és axiómák segítségével [Gomez-Perez02]. Az axiómák azonban nem csak atomi fogalmakra hivatkozhatnak, hanem halmazműveletek segítségével (metszet, unió, komplemensképzés) komplex operandusokat építhetünk fel [Bechhofer01a]. A fogalmakhoz kötött tulajdonságokhoz további megkötéseket tehetünk, melyek lehetnek kardinalitásbeliek illetve a betöltő fogalomra vonatkozó Bool-kifejezések. A nyelv képességei jól áttekinthetőek a hivatalos honlapon található, jegyzetekkel ellátott példa ontológián.[17] Néhány saját példán keresztül én is bemutatom a nyelv fő sajátosságait.

A halmazművelettel definiált osztály bemutatására definiálhatjuk a **nőstény**eket mint a **nem hím** állatokat, használva a komplemensképzést:

```
<daml:Class rdf:ID="nőstény">
  <rdfs:subClassOf rdf:resource="#állat"/>
  <daml:disjointWith rdf:resource="#hím"/>
</daml:Class>
```

[14] http://www.cs.umd.edu/projects/plus/SHOE/
[15] DAML-ONT Initial Release, http://www.daml.org/2000/10/daml-ont.html
[16] Gene Ontology Next Generation, http://gong.man.ac.uk/
[17] Annotated DAML+OIL Ontology Markup, http://www.daml.org/2001/03/daml+oil-walkthru

A **ragadozók** olyan **állatok**, akik **állatokat esznek**, ez az **eszik** tulajdonság értékkészletére vonatkozó megkötés (*daml:Restriction*):

```
<daml:Class rdf:ID="ragadozó">
  <rdfs:subClassOf rdf:resource="#állat"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#eszik"/>
      <daml:toClass rdf:resource="#állat"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

A tulajdonsághoz rendelt kardinalitás segítségével leírhatjuk, egy egyedhez minimum, maximum vagy pontosan hány, a tulajdonságnak megfelelő attribútum kapcsolódhat. Például ábrázolható, hogy minden **embernek** két **szülője** van:

```
<daml:Class rdf:ID="ember">
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="2">
      <daml:onProperty rdf:resource="#szülője"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

A tulajdonságok lehetnek tranzitívek, injektívek illetve megadható az inverzük. Értékkészletük az osztályokon illetve osztályokból felírt kifejezéseken túl felvehet konkrét adattípusokat is, az XML sémákra hivatkozva. Dátumok ábrázolásához, például a születési idő megadásához az *xsd:date* lesz az értékkészlet típusa:

```
<daml:DatatypeProperty rdf:ID="születésiIdő">
  <rdfs:domain rdf:resource="#ember"/>
  <rdfs:range
     rdf:resource="http://www.w3.org/2000/10/XMLSchema#date"/>
</daml:DatatypeProperty>
```

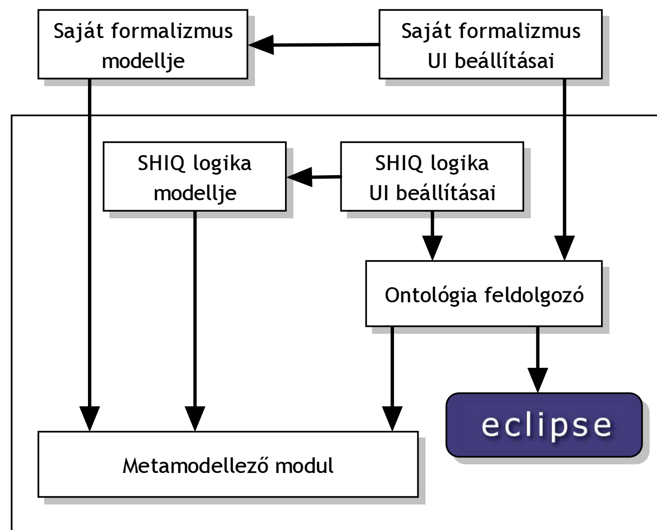
Az osztályok közötti ekvivalenciát (*daml:sameClassAs*) is kifejezhetjük, illetve léteznek további beépített állítások, melyek osztályok pontos definícióját adják. Nem csak szükséges feltételeket írnak le, hanem egyben elégségeseket is. Ilyenek az osztályok metszete (*daml:intersectionOf*), ellentettje (*daml:complementOf*), illetve a diszjunkt unió (*daml:disjointUnionOf*), utóbbi tekinthető szintaktikai édesítőszernek is (*daml:sameClassAs* + *daml:unionOf* + *daml:disjointWith*).

A szükséges és elégséges feltételek megadása lehetővé teszi az osztályok automatikus hierarchiába sorolását, ezt a műveletet nevezik klasszifikációnak. Például ha tudjuk, hogy aki nem eszik húst, az vegetáriánus, és a tehenek csak füvet esznek, valamint a fű nem hús, akkor a tehenek vegetáriánusok.

A Szemantikus Web logikai rétegének nyelveit a kezdetektől tudatosan az $\mathcal{SHIQ}$ leírólogika kifejező erejéhez igazították, ami nem véletlen. Már az OIL nyelvcsalád Standard OIL és Instance OIL rétege is leírható az $\mathcal{SHIQ}(D)$ logikával, mely az előbbi kis mértékű kiegészítése [Horrocks02]. Az ontológia nyelvek és a leíró logikák pontos kapcsolatát a 4.3. fejezetben részletezem.

3.1.4. DL-workbench

Az Open Cascade S.A.,[18] francia székhelyű, főleg szimulációs eszközöket készítő nemzetközi IT vállalat és a francia CNRS Laboratoire PSI[19] közös fejlesztésének eredménye a DL-workbench ontológia-szerkesztő platform. Azért tárgyalom mégis az ontológia nyelvek körében, mert ugyan az eszköz alapvetően ontológia manipulálására hivatott, felépítésében egyedülálló: a nyelvet leíró metamodellt teljes mértékben elkülönítették a többi komponenstől.



7. ábra. A DL-workbench platform architektúrája

Három különálló modulra választották az ontológia kezelő rendszert [Kazakov03a]: a metamodell szolgál az ontológiai formalizmus definiálására (7. ábra). Egy programozói felületen keresztül szolgáltatja az ontológia entitásain végezhető műveleteket.

[18] `http://www.opencascade.com/`

[19] Centre National de la Recherche Scientifique, Laboratoire Perception Systémes Information `http://psiserver.insa-rouen.fr/psi/`

A második a formalizmustól teljesen független felhasználói felület, a harmadik pedig az $\mathcal{SHIQ}$ leíró logikán alapuló következtető.

Azáltal, hogy a támogatott ontológia nyelvet leíró metamodell elválik a platform többi részétől, két egyedülálló lehetőséget tud nyújtani: tetszőleges új formalizmust lehet implementálni és kipróbálni a platform segítségével, így megnyílik az út a kísérletezésre; másrészt azonos körülmények között hasonlíthatóak össze a már létező formalizmusok.

Az új formalizmus elkészítéséhez a metamodellt felépítő Java interfészeket kell megvalósítani. A két alapvető interfész a meta-fogalom és a meta-tulajdonság. A meta-fogalomból származtatott interfész definiálhatja például a DAML+OIL nyelv *daml:Class* entitását. A Java interfészek taxonómiája határozza meg a meta-fogalmak hierarchiáját, így kihasználható az objektum-orientált nyelv öröklődés általi kód-újrafelhasználó hatása és a forráskód is áttekinthetőbb. A meta-tulajdonságok értelmezési tartománya konzisztencia-ellenőrzésre használható, az értékkészlet hordozza a tulajdonság korlátozó hatását a metamodell egyszerű típusrendszere segítségével. A meta-tulajdonságokhoz elő- és utófeldolgozó eljárások társíthatóak, az előfeldolgozó például ellenőrizheti, hogy egy fogalom azonosítója szabályos URI típusú. A metamodell magja támogatást nyújt a leíró logikai következtetőgép kezeléséhez és a modell leíró logikai megfeleltetéséhez is, így ha megfelelően használjuk a formalizmusokat, logikai következtetéseket is végrehajthatunk.

A DL-workbench készítői a DAML+OIL nyelvet formalizálták a metamodell segítségével és a RACER következtetőgépet (4.4.2. fejezet) kapcsolták a platformhoz, így módosítások nélkül használható DAML+OIL ontológiák készítésére [Kazakov03b]. Az ontológia szerkesztő modult mint Eclipse[20] plug-int implementálták, mely a felhasználói felület űrlapjait – melyekkel az ontológia entitásai manipulálhatóak – a metamodelltől lekérdezett információk alapján állítja össze. Az implementáció minden része elérhető egy Java API-n keresztül, így könnyen illeszthetőek hozzá további komponensek.

A platform tehát nagyon rugalmas a felhasználható modellek tekintetében, valójában az eddig tárgyalt ontológia nyelvek általánosításának is tekinthető. Ellene szól azonban, hogy ugyan nyílt forráskódú, dokumentációja nagyon hiányos. Prototípus állapotban van, így számtalan hibát tartalmazhat, de sem kielégítő felhasználói dokumentációt, sem jól dokumentált forráskódot nem tartalmaz. Akkor érdemes alkalmazni, ha nagyon speciális, egyedi ontológia nyelvet akarunk használni, melyhez nem hozzáférhetők a szükséges eszközök (szerkesztés, következtetés, API); vagy feltétlen szükséges egyszerre több formalizmust párhuzamosan, egységes keretek között

[20]IBM Eclipse Platform, nyílt forráskódú univerzális fejlesztőfelület, `http://www.eclipse.org/`

kezelnünk, például integrálási feladatoknál.

3.1.5. OWL

Az OWL[21] egy szemantikus jelölő nyelv, mely az RDF(S) kiegészítéseként teszi lehetővé ontológiák leírását. Lényegében a DAML+OIL nyelvből származtatták, annak a W3C WebOnt[22] csoportja által készített továbbfejlesztett, szabványosított megfelelője. A munkacsoport 2001 novemberében kezdte meg munkáját, melynek fontos mérföldköve, hogy 2004 februárjában elkészült az OWL W3C ajánlás.

A nyelvnek az OIL-hoz hasonlóan nyelvjárásai vannak, melyek célja a különböző kifejezőerőhöz társuló eltérő következtetési komplexitás felsorakoztatása [OWL referencia]. Az *OWL Lite* azokat a felhasználókat célozza, akiknek csak hierarchiára és egyszerű megkötések bevezetésére van szükségük. Például a kardinalitásra vonatkozó megkötések csak 0 vagy 1 értékűek lehetnek. Cserébe a következtetés teljes és gyors, nem igényel sok erőforrást.

Az *OWL DL* kifejezőerejét pontosan úgy határozták meg, hogy a lehető leggazdagabb leírásokat tegye lehetővé, de még teljes és eldönthető logika tartozzon hozzá. (A teljesség az igaz állítások garantált levezethetőségét, az eldönthetőség minden számítás időben véges lefolyását jelenti.) Nevét onnan kapta, hogy az OWL DL ontológiák leíró logikai (*description logics*, DL) állításokra fordíthatóak le. A leíró logikákról, illetve arról, hogy a nyelvjárás mekkora részére léteznek jelenleg valóban hatékony, gyakorlatban is használható következtetési algoritmusok, a 4.3. fejezetben írok részletesen.

Az *OWL Full* a nyelv legbővebb osztálya, a legmagasabb szintű szabadságot és kifejezőerőt biztosítja, de nem szavatol semmit számítási komplexitásáról. Rendelkezik az RDFS teljes metamodellezési képességével, például osztályok lehetnek egyben példányok is.

A DAML+OIL és az OWL nyelv közti különbségek listáját megtalálhatjuk az OWL referenciában [OWL referencia, D függelék]. A nyelv szemantikája sokat változott, a DAML+OIL leginkább az OWL DL-hez áll közel. Egyébként a változtatások jelentős része arra terjed ki, hogy a DAML+OIL entitások helyett ahol lehet, azok RDF(S) megfelelőit használják (pl. *daml:subClassOf*), a félrevezető megnevezéseket lecserélték (pl. *daml:sameClassAs* helyett *owl:equivalentClass*), illetve a redundáns elemeket eltávolították (pl. *daml:disjointUnionOf* leírható *owl:unionOf* és *owl:disjointWith* segítségével).

[21] OWL Web Ontology Language, `http://www.w3.org/TR/owl-features/`
[22] Web-Ontology Working Group, `http://www.w3.org/2001/sw/WebOnt/`

3.2. Ontológia szerkesztők

A következőkben bemutatok néhány, a felsorolt ontológia nyelvekhez tartozó népszerű ontológia szerkesztőt. Amíg a szerkesztők az egész fejlesztési folyamatot végigkísérik, egy-egy speciálisabb részfeladat megoldására azonban további eszközöket érdemes igénybe venni.

Az ontológia fejlesztés számtalan különböző aspektusára léteznek software eszközök. Ilyenek a külső forrásból felhasznált ontológiák összehasonlítását, fogalmaik egymáshoz rendelését, ellentmondás-mentességének vizsgálatát, összefésülését vagy konverzióját végző programok. A választott ontológia szerkesztő mellett további software-ek segíthetnek az ontológia áttekintésében, lekérdezésében, vizualizációjában. Az ontológia fejlesztést támogató összes eszközt viszont nehéz lenne áttekinteni, mivel egészen egyedi részfeladatokat is segíthetnek.

Az ontológia szerkesztők alapos összehasonlítását végezte el Denny, cikkében ötvennyolc különböző eszköz képességeit tekinthetjük át táblázatos formában [Denny02].

3.2.1. OilEd

Az OilEd[23] ontológia szerkesztőt kezdetben az OIL nyelv támogatására, képességeinek demonstrálására kezdték el fejleszteni a Manchesteri Egyetemen. Később vált képessé DAML+OIL nyelvű ontológiák szerkesztésére. Készítésénél nem volt cél egy teljes ontológia fejlesztő platform előállítása, nem alkalmas nagy, ipari méretű ontológiák készítésére, nem támogatja ontológiák összevonását sem. Sajnos a fejlesztését abbahagyták. (A szerkesztő honlapján az OWL nyelv támogatására képes új verzióját 2003 elejére ígérték, ami azonban nem valósult meg.)

Az OIL nyelv képességeihez híven lehetővé teszi logikai következtetőgép használatát, mellyel felfedhetők az inkonzisztenciák, és az ontológiában megfogalmazott implicit relációk is lekérdezhetővé válnak. A kezdetektől a FaCT következtetőt használja, de az utolsó változat már támogat egy általános interfészt, melyen keresztül a RACER következtetőgép is elérhető.

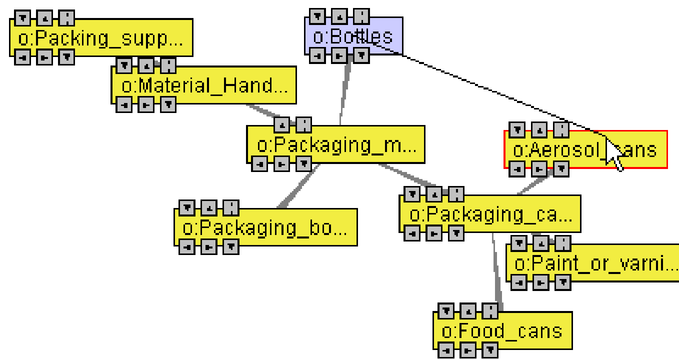
Az ontológia bővítéséhez külön felület szolgál az osztályok, tulajdonságok, példányok definícióinak és az axiómák leírásának szerkesztésére. Csak korlátozott méretű ontológiák szerkesztésére alkalmas, mivel az egyes kategóriákon belül az összes entitás listájából kell választanunk, ami több száz elem esetén kényelmetlen lenne. Helyette a mai eszközökben már fa-struktúrába szervezett hierarchiákban navigálhatunk, a hierarchiát az OilEd-ben csak külön ablakban tudjuk megtekinteni.

[23] `http://oiled.man.ac.uk/`

Az eszköz erőssége az OIL logikai állításainak támogatása. Külön kifejezés szerkesztőt tartalmaz a megszorítások összeállítására. Mára elavult eszközzé vált, a későbbi eszközök azonban átvették számos előnyös tulajdonságát.

3.2.2. KAON OI-modeler

A *KAON tool suite* egy teljes infrastruktúrát foglal magába ontológiák készítéséhez, menedzseléséhez és ontológia-alapú alkalmazások építéséhez. Részét képezi az ontológia szerkesztő, az OI-modeler is, mely önmagában is működőképes, de a többi komponenssel összekapcsolva egyedi, extra szolgáltatásokat nyújt.



8. ábra. Az OI-modeler ontológia szerkesztő gráf alapú ábrázolása

A legnagyobb előnye a többi szerkesztővel szemben a sokkal fejlettebb kezelőfelülete, vizualizációs képessége. Az ontológia entitásait nem a hierarchiának megfelelő lineáris nézetekben, hanem minden relációt feltüntető, gráf alapú ábrázolásban választhatjuk ki, és ugyanez a felület szolgál az új kapcsolatok felvételére is, mint ahogy a 8. ábrán látható.

Nagy méretű ontológiák építésére is tökéletesen alkalmas. A KAON keretrendszert az ipari igényeknek megfelelően alakították ki, így minden komponense jól skálázható. A szerkesztő esetében a moduláris ontológiák kezelése, az entitások keresését lehetővé tevő query ablak is ezt a célt szolgálja.

A háttértudás összeállításánál és későbbi karbantartásánál is fontos a konzisztencia fenntartása. A KAON ontológia nyelv korlátozott kifejező erejének köszönhetően olyan mechanizmusokat építhettek a szerkesztőbe, melyek képesek minden változtatás során automatikusan kikövetkeztetni azokat a módosításokat, amik a konzisztens állapot fenntartásához szükségesek [Maedche03a]. Ezeket megtekintve dönthetünk, hogy valóban el akarjuk-e végezni a – most már változtatások sorozatává kiegészített – műveletet.

A keretrendszer részeként a csoportmunka támogatására is fejlett segítséget nyújt. Több felhasználó munkáját segíti összehangolni az *Engineering Server* modul, mely

valós időben jeleníti meg a másik szerkesztőjében az őt érintő módosításokat. Valójában az ontológia módosítását végző programozói interfész, a KAON API egy külön implementációja [Maedche03b]. Az RDF helyett a gyakori ontológia módosítások figyelembe vételével tervezett adatbázis sémákra építkezik, így hatékonyabb az ontológia lekérdezésére és példányok módosítására optimalizált *KAON Servernél.* Lehetővé teszi a konkurens hozzáférést és a műveletek visszavonását, mivel a módosító műveletek a szerver tranzakcióinak felelnek meg.

Az *OI-modeler* a *KAON tool suite* többi részéhez hasonlóan nyílt forráskódú, jól dokumentált, Java nyelven íródott eszköz. Kiegészítések (*plug-in*) készítését egyelőre nem támogatja, de a nyílt fejlesztés – kicsit több munkával ugyan – bárki számára lehetővé teszi a szerkesztő egyedi funkciókkal való bővítését.

3.2.3. Protégé

A tudás alapú rendszerek fejlesztését, kutatását támogató *Protégé* platformot[24] már több, mint másfél évtizede fejlesztik a Stanford Egyetemen. A platform első ránézésre egy szokványos ontológia szerkesztő, mely a szerkesztett modellt szöveges file-ba, RDF(S) állományba vagy JDBC kompatibilis adatbázisba is képes menteni.

Azonban ennél sokkal többre is képes, mivel a bővíthetőség, testreszabhatóság jegyében úgy alakították ki, hogy funkcióit minden irányban *plug-in*-ek írásával bővíthessük [Natalya01]. A belső modellje helyett más modellt is használhatunk, ha a megfelelő bővítéseket elkészítjük. Ehhez az adattárolás módját (*back end*) és a felhasználói felület kiegészítéseit kell implementálnunk.

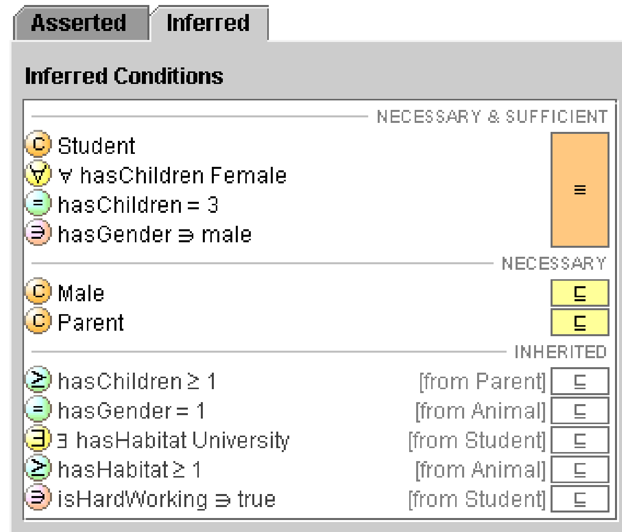
Utóbbira három lehetőségünk is kínálkozik. Ha csak az entitások tulajdonságait tároló elemi egységeknek, a *slotoknak* akarunk egyedi felületet készíteni, azt *Slot Plug-in* írásával tehetjük meg. Ha a felhasználói felület egy teljes paneljét akarjuk testre szabni, *Tab Plug-in*-t készíthetünk. A harmadik lehetőség, ha a *Protégé* szerkesztő felületét teljes egészében elvetjük, független alkalmazást fejlesztünk a tudásbázis kezelésére. Ilyenkor a *Protégé* belső modelljének API-jára építkezhetünk.

Mindezek tükrében a *Protégé* nem egyszerűen ontológia szerkesztő, hanem egy Java nyelvű, nyílt forráskódú eszköz, mely bővíthető architektúrát kínál egyedi igényekhez igazított tudás alapú alkalmazások fejlesztéséhez.

Nagy múltjának és nyílt szemléletének köszönhetően nagy felhasználói bázisra tett szert, mely mára már hatvan ingyenesen hozzáférhető kiegészítéssel járult hozzá a rendszerhez. Ezek különböző tudásreprezentációk támogatásán túl egészen eltérő fel-

[24]The Protégé Ontology Editor and Knowledge Acquisition System,
`http://protege.stanford.edu/`

adatokat is ellátnak, például fejlettebb vizualizációt, speciális következtetést nyújtanak, vagy a tudásbázisban történő keresést, navigációt segítik.



9. ábra. A Protégé OWL Plugin képes az OWL kifejezések kezelésére

Számunkra a legfontosabb kiegészítése az *OWL Plugin*,[25] mellyel a *Protégé* a Szemantikus Web első számú ontológia nyelvét is támogatja. Az OWL ontológia konzisztenciáját leíró logikai következtetőgéppel is ellenőrizhetjük, illetve megtekinthetjük, milyen információk vezethetők le az osztálydefiníciókból. A *plug-in* lehetőséget ad OWL kifejezések felépítésére, valamint hasonló formában képes megmutatni a következtetőgép kimenetéül kapott információkat (9. ábra).

[25]`http://protege.stanford.edu/plugins/owl/index.html`

3.3. Ontológia programozói felületek

Az IKF keretrendszer ontológia kezelő komponensének megvalósításához szükséges egy programozói felület, melyen keresztül elérhető, lekérdezhető az ontológia. Rövid áttekintést nyújtok tehát a szóba jöhető technológiákról, az ontológia modul esélyes ontológia nyelveit támogató API-król. Végül az ontológia nyelv megválasztása után kiválasztom a használni kívánt felületet is.

3.3.1. KAON API

A *KAON tool suite* részét képező API KAON nyelvű ontológia elérését teszi lehetővé. Támogatja ontológiák módosítását, moduláris ontológiák kezelését, a konkurens hozzáférést és a tranzakció-kezelést is.

Az API Java interfészekből áll, melyek implementációját a tárolási mechanizmusnak megfelelően kell kiválasztani. Ezzel az alkalmazás tervezőjét függetleníti az ontológiák fizikai elérhetőségétől, ugyanaz az alkalmazás RDF állományokban, vagy relációs adatbázisban tárolt ontológiát is fel tud dolgozni.

A programozói felület a KAON nyelv szerkezetét tükrözi. A moduláris ontológia részeit az *OI-model* interfész reprezentálja, ezek tehát más OI-modelleket foglalhatnak magukban. Az entitások leírására a *Concept* (fogalom), *Property* (tulajdonság) és *Instance* (példány) interfészek szolgálnak.

A módosítások elvégzésénél a KAON API automatikusan fenntartja az ontológia konzisztenciáját. Például nem enged létrehozni körkörös öröklődést, valamint minden osztálynak a *Root* ősosztály leszármazottjának kell lennie. Ennek érdekében minden műveletet kiegészít további szükséges műveletekkel, a viselkedés hangolásához fejlődési stratégiák (*evolution strategies*) közül választhatunk. A stratégia része például, hogy fogalom törlésekor annak gyerekei a törölt fogalom szüleinek, vagy a *Root* fogalom közvetlen leszármazottai legyenek, esetleg az eltávolított fogalom minden leszármazottja is törlődjön.

Az API további sajátossága az *Engineering Server* használatakor a felhasználók értesítése az ontológia változásairól, az *OI-modeler* szerkesztő is ennek segítségével képes fejlett csoportmunka támogatásra.

3.3.2. Jena

A Jena[26] egy Java nyelvű keretrendszer Szemantikus Web alkalmazások készítéséhez, melyet a Hewlett Packard fejlesztő laboratóriumában[27] készítettek.

[26]Jena Semantic Web Framework, `http://jena.sourceforge.net/index.html`
[27]HP Labs Semantic Web research group, `http://www.hpl.hp.com/semweb/`

Kezdetben kizárólag RDF gráfok létrehozását és kezelését támogatta [McBride01]. Az RDF állításokat a memóriában vagy SQL adatbázisban is képes tárolni, lekérdezésükhöz erőforrás-centrikus és állítás-centrikus (tripleteket kezelő) függvények is a rendelkezésünkre állnak.

Később RDFS, DAML+OIL és OWL támogatással is ellátták, így ontológiák kezelésére is alkalmas lett. Azonban az ontológiát nem szemantikus szinten, hanem a leírásához hűen tudjuk csak kiolvasni: valójában az RDF állítások lekérdezését teszik kényelmesebbé az ontológia entitásainak megfelelő osztályok implementálásával.

Egyszerű következtetési képességekkel is rendelkezik: egy szabály alapú következtetőt tartalmaz, melyet az RDFS és az OWL Lite nyelvjárás támogatásához szükséges szabályokkal töltöttek fel. Ennek ellenére a Jena egy ontológia beolvasásakor nem annak szemantikus modelljét, hanem a szintaktikához hű képét építi fel, így csak korlátozottan alkalmas lekérdezések futtatására.

3.3.3. Protégé

Az ontológia kezelő komponenst a 3.2.3. fejezetben ismertetett *Protégé* ontológia szerkesztő belső tudásbázisát felhasználva, a *Protégé API*-ra építkezve is kialakíthatjuk. Ennek legfőbb előnye, hogy felhasználhatjuk a szerkesztő valamennyi kezelőszervét, így könnyedén illeszthetünk a modulba az ontológia megjelenítésére vagy módosítására szolgáló paneleket.

Azonban az OWL nyelv támogatásához az *OWL Plug-in*-ra van szükségünk, mely nem támogatja az OWL néhány nyelvi elemét, amihez nem lehet, vagy nagyon nehéz automatikusan űrlapot generálni.

A fenti korlátozások miatt csak akkor célszerű a *Protégé*-re építkezni, ha elsődleges az ontológia vizualizációja, vagy módosíthatóságának megvalósítása, így az automatikus űrlapgenerálásban rejlő előnyöket ki tudjuk aknázni.

3.3.4. OWL API

A *World Wide Web Consortium* ajánlásaként elfogadott, mára a Szemantikus Web ontológia nyelvei közti háború esélyes befutójaként emlegetett nyelv az OWL.

Készítőinek célja egy mindenki számára könnyen elérhető ontológia API-t alkotni, mely hasonló módon járulhat hozzá a Szemantikus Weben az OWL nyelv elterjedéséhez, mint ahogy annak idején a DOM API segítette az XML technológia elterjedését [Bechhofer03].

A Java nyelvű OWL API-t[28] úgy alakították ki, hogy az ontológia feldolgozása közben megőrizze az ontológia alkotójának kifejezésmódját, a szemiotikát (*semiotics*) is. A gépi feldolgozás számára hordozott jelentés szempontjából ennek nincs jelentősége, de az ontológiát szemlélő vagy szerkesztő felhasználót segítheti, ha pontosan látja, a szerző milyen módon modellezte a világot [Bechhofer03, p. 5]. Például egy tartalmazási reláció az osztályok között kifejezhető a gyerek osztály definíciójában, vagy a definícióktól függetlenül, axióma használatával is.

A programozói felület tervezésekor elválasztották egymástól a belső modellt, a változtatások kezelését és a következtetést. Így ha nem kívánjuk módosítani az ontológiát – mint az IKF komponens esetében –, a változtatást kezelő *change* csomagot kihagyhatjuk az implementációból.

Jelenlegi állapotában az API képes a szabványos formátumú OWL ontológiákat olvasni illetve írni, valamint megállapítja, melyik OWL nyelvjárásba tartozik az ontológia.

A következtetés támogatásához elkezdték kialakítani a külső következtetőgépek szolgáltatásainak eléréséhez szükséges osztályokat, de ezek egyelőre csak korlátozottan, nem a teljes nyelvre alkalmazhatóak, illetve nem dokumentáltak. Sajnos általában az egész könyvtárról elmondható, hogy a metódusok pontos működését csak a nevük alapján, vagy rosszabb esetben a példaprogramokra vagy a forráskódra támaszkodva térképezhetjük fel. A közeljövőben azonban várhatóan sokat fog fejlődni az implementáció.

3.4. Technológiák választása

Ontológia nyelv. Az IKF keretrendszer cél környezetét modellező fogalmi rendszert egy ontológia nyelv segítségével ábrázolhatjuk. Ennek a reprezentációnak a kiválasztását a projekt többi résztvevőjével megegyezésben végeztem, a már ismertetett, saját metamodell kialakítására is alkalmas környezet (DL-workbench) és több ontológia nyelv közül.

Az RDF(S) ontológia nyelv további nyelvek alapjául szolgál. Szemantikája nagyon szabad, ezért nehezen értelmezhető. Például korlátlan metamodellezési képességei problémákat vetnek fel, nem teszik lehetővé logikai következtetőgép használatát. A KAON nyelv kifejezőereje lényegesen korlátozottabb, kialakításakor éppen a nagyon hatékony kezelhetőségére törekedtek. Ezért sajnos a nagyvállalati alkalmazhatóságnak hála hiába rendelkezik hatékony vizualizációt támogató szerkesztővel, nem alkalmas komplex logikai állítások megfogalmazására, nem elegendő a tárgykörnyezet

[28] `http://owl.man.ac.uk/api.shtml`

megfelelő pontosságú modellezésére.

A DAML és OIL nyelvekből kialakított DAML+OIL majd OWL nyelvek viszont sokkal több különböző nyelvi elemmel teszik lehetővé a fogalmi rendszer leírását. Az OWL nyelvben ráadásul e mellett az OIL-hoz hasonlóan három nyelvjárás kialakításával eltérő kifejezőerejű nyelvváltozatokat is bevezettek, melyek közül az OWL DL felel meg legjobban az IKF projekt céljainak.

A DL-workbench platformot csak akkor célszerű alkalmazni, ha több reprezentációt együttesen vagyunk kénytelenek használni, vagy elkerülhetetlen saját adatmodell készítése. Az IKF esetén azonban inkább elkötelezzük magunkat egy nyelv mellett, így a választásunk az OWL nyelvre esett. Fenti előnyei mellett nemrég W3C szabvánnyá választották, ezért rohamos elterjedésére lehet számítani. Nőni fog tehát az OWL nyelvű szabadon hozzáférhető ontológiák száma, és nagyobb átjárhatóságot biztosít a Szemantikus Webre is.

Ontológia szerkesztő. Az OWL ontológiák szerkesztéséhez a szabadon hozzáférhető alkalmazások közül ma a *Protégé* nyújtja a legtöbb eszközt. Leginkább ez nyitott architektúrájának, egyszerű bővíthetőségének köszönhető, folyamatosan nő az ingyenesen hozzáférhető bővítmények, *plug-in*-ek száma, melyek a legkülönfélébb feladatokat látják el.

Ugyan a KAON nyelvhez készített *OI-modeler* áttekinthetőbb vizualizációt tesz lehetővé, továbbra is csak a KAON nyelvet támogatja, és fejlesztése az utóbbi időben egyébként is lelassult. A *Protégé* egy bővítménye segítségével kezeli az OWL nyelvet, és kényelmesen csatlakoztatható következtetőgéphez is, így az IKF ontológiájának fejlesztésére a leginkább alkalmas eszköz. Az ontológia komponens elkészítése után lehetőség nyílik arra is, hogy a kényelmesebb használat végett egy bővítmény elkészítésével az ontológiát a file-rendszer kikerülésével, közvetlenül tölthessük a szerkesztőből a kezelő modulba.

Ontológia programozói felület. Az ontológiák használatát megkönnyítő programozói felületek közül az OWL nyelvet a Jena és az OWL API támogatják, valamint használhatjuk a *Protégé* megfelelő API-ját is a feladatok ellátására. Utóbbit csak akkor célszerű választani, ha az ontológia módosítását is lehetővé kell tenni, melyet a Protégé automatikus űrlapok generálásával praktikusan old meg.

A komponens esetében azonban csak lekérdezésekre kell válaszolni, az ontológia nem módosul. Így a több szolgáltatást nyújtó, eredetileg is az OWL-hez igazított OWL API-t választottam az RDF alapú Jena-val szemben.

4. Következtetés ontológiával

A fejezet során először bemutatom az ontológiai következtetés általános sémáit, majd ismertetem a következtetés céljait az IKF keretrendszer esetére vonatkoztatva. Mivel a választott ontológia nyelvben, az OWL-ben a leíró logikára történő átírása által lehet következtetni, bemutatom a leíró logikákat, különös tekintettel az OWL-nek megfeleltethető kifejezőerejű változatait. Végül a gyakorlati megvalósítás végett a felhasználható következtetőgépeket vizsgálom meg, kiválasztva az ontológia modul céljainak legmegfelelőbbet.

4.1. Az ontológiai következtetés fajtái

A következtetés a különböző tudásreprezentációk közül a logikák esetében részletesen kidolgozott terület, pontosan ismerjük, hogy a különböző logikai reprezentációk milyen kifejezőerővel bírnak, illetve legtöbbször a következtetési mechanizmusok fő tulajdonságai (teljesség, eldönthetőség, futási idő és tárigény) is ismertek.

Az ontológia alapú következtetés annál újkeletűbb, nem ismerjük minden aspektusát. A szemantikus reprezentáció miatt mások az előforduló problémák és a megoldással szemben támasztott elvárások [Sowa01]. Eltérő szemlélet szükséges az ontológiák közötti és egy ontológián belüli következtetés végrehajtásához.

Következtetés ontológiák között. Ontológiák közötti következtetés esetén szemantikus információt képezünk le egyik ontológia reprezentációjából egy másikéra.

Tudás alapú rendszerek együttműködésénél az eltérő fogalmi rendszerekben megfogalmazott állítások átadásához létre kell hozni a különböző ontológiák fogalmai között egy megfeleltetést (*alignment*), vagy be kell vezetni egy köztes ontológiát, ami közvetíti az információt (*integration*). Ontológiák egymásba ágyazásánál vagy összefűzésénél (*merging*) is a fogalmak megfeleltetésére van szükség. Az esetleg megjelenő inkonzisztenciákat meg kell szüntetni a fogalmak finomításával, vagy ha szükséges, akár definícióik megváltoztatásával.

Ontológián belüli következtetés. A hagyományos logikai következtetéssel analóg mechanizmus feladata itt is egy tényhalmazból kiindulva a tárolt háttértudás kiegészítése. Az IKF rendszerben az ontológia kezelő modul feladata egy előre elkészített, a tárgyterület entitásait minél pontosabban leíró ontológia kezelése. Nincs kapcsolata más, ontológiát kezelő komponensekkel, a következtetés tehát egy ontológián belül történik.

Az ontológia perzisztenciája szerint megkülönböztetünk zárt és nyílt rendszert használó következtetést. Zárt rendszer esetén az ontológia tartalma nem változhat, a tényekből kikövetkeztethető tudást kell kinyerjük. A feladat az implicit formában rendelkezésre álló információ explicit visszaadása.

Nyílt rendszert feltételezve a következtetés alapjául szolgáló tények tanulási folyamatot indítanak el, a háttértudásba beépülnek.

Az ontológia kezelő modul esetében a következtetés az implicit megbúvó információk teljes körű visszaadását jelenti. Az ontológia tehát zárt abban az értelemben, hogy azt mindvégig az ontológus által elkészített formában tároljuk, és csak a logikai következtetőgép egészíti ki, bővíti a lekérdezések kiszolgálásakor. A ténybázis, vagyis az IKF rendszer végfelhasználója által lekérdezhető, folyamatosan bővülő tudásbázis azonban nyílt. Az információforrások által szolgáltatott információ mindig beépül a tudásbázisba.

4.2. A következtetés célja

Ontológia alapú következtetésnek számtalan célja lehet, mint ahogy az ontológiák felhasználása is nagyon szerteágazó.

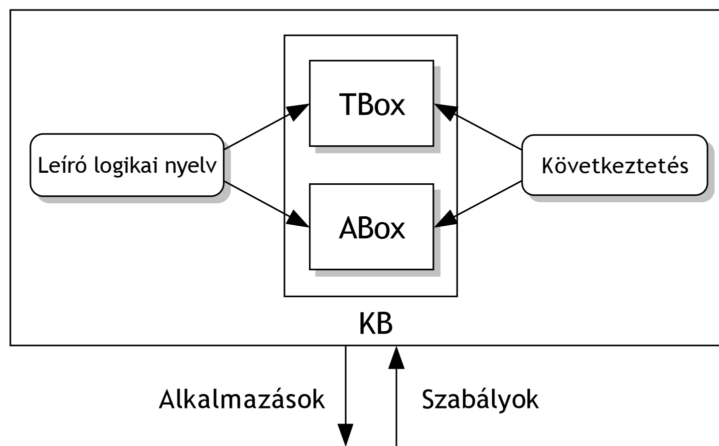
A következtetés fontos az ontológia minőségének biztosításában [Baader03]. Segítségével tervezési időben ellenőrizhető az ontológia ellentmondás-mentessége. Például az ontológia konzisztenciájának feltétele, hogy minden fogalma kielégíthető legyen: ne legyen kizárható, hogy van legalább egy példánya. Nyilván nem lehet szándéka az ontológusnak biztosan üres fogalmakat definiálni. Hasonlóan fontos szolgáltatás az ontológia tervezésekor az implicit relációk feltárása, különös tekintettel a fogalmi hierarchiára. Ha tudjuk, mely fogalmak szinonimái vagy részfogalmai egy másiknak, ellenőrizhetjük, hogy a definíciók a szándékolt jelentést hordozzák-e.

Az IKF keretrendszer esetében az ontológiát több, független funkcionalitás is használja [Varga03]. Ide tartozik az információkinyerés támogatása a tárgyterületi tudással, a keresőkérdés értelmezése és kibővítése. Az ontológia felhasználásakor a rendszer többi komponense az ontológia entitásait lekérdezi, például egy összetett fogalom tartalmazási relációit vizsgálja. Ennek megállapítása azonban az interkategoriális kényszerek miatt nem triviális feladat.

Az ontológiai következtetés célja tehát az ontológiát felépítő relációkból levezethető állítások visszaadása. Az ontológia azon átalakítását, mely az explicit megfogalmazott kapcsolatok mellett az összes implicit relációt is levezeti, nevezzük az ontológia *klasszifikálásának*. Az IKF ontológia komponens a klasszifikált ontológiát nyújtja a többi komponens számára.

4.3. Leíró logikák

A leíró logikák (*description logics*, DL) a tudásreprezentációs nyelvek egy családját alkotják, melyek alkalmasak egy alkalmazási terület tudásanyagának strukturált, formális reprezentálására. Nevüket onnan kapták, hogy a tárgyterület ábrázolásának legfontosabb részei a fogalmak *leírásai*, vagyis atomi fogalmakból és atomi tulajdonságokból, az adott DL konstruktorai segítségével felépített kifejezések. Másrészt a korábbi tudásreprezentációkkal (szemantikus hálók, keret alapú módszerek) ellentétben rendelkeznek egyértelmű, formális, *logikán* alapuló szemantikával [Russel00, p. 395]. (Természetesen a különböző szemantikus hálókhoz is rendelhetünk egyértelmű jelentést, alkalmazásuk azonban legtöbbször nem a logikai megfeleltetésen alapult.)



10. ábra. Leíró logikán alapuló tudásábrázoló rendszer architektúrája

A leíró logikák kategóriákra és azok meghatározására koncentrálnak [Baader02]. A leíró logikán alapuló tudásábrázoló rendszerek lehetővé teszik tudásbázisok létrehozását, manipulálását és a tartalmukon történő következtetést. Általános architektúrájukat mutatja a 10. ábra. A tudásbázis (*knowledge base*, KB) két részből áll: a fogalmak és szerepek leírása, azaz a „szótár" a TBox (*terminology box*), míg a példányokra vonatkozó kijelentéseket az ABox (*assertion box*) tartalmazza.

A fogalmak (kategóriák) példányok halmazát jelölik, mint az ontológiák esetében, a szerepek (tulajdonságok) bináris relációkat jelölnek a példányok között. Az atomi fogalmakon és szerepeken túl minden DL rendszer lehetővé teszi összetett fogalmak és szerepek létrehozását konstruktorok segítségével. A TBox tehát valójában ilyen komplex leírásokhoz rendel elnevezéseket. A DL rendszereket legjobban a leírások készítésére használható konstruktorokkal tudunk jellemezni, ilyen szempontból leíró logikai nyelveket különböztetünk meg.

Jelölés	*Megnevezés*	*Magyarázat*
A	atomi fogalom	egy halmaz jelölése, például *Személyek*
$\top$	univerzális fogalom – *top*	minden példány halmaza
$\bot$	üres fogalom – *bottom*	üres halmaz
$\neg A$	atomi negálás	*Ehetetlen* $=$ $\neg$ *Ehető*
$C \sqcap D$	metszetképzés	*Tinilány* $=$ *Tinédzser* $\sqcap$ *Lány*
$\forall R.C$	érték korlátozás	fogalmak, melyek minden R-je C-beli pl. *LányosApa* $=$ *Apa* $\sqcap$ $\forall$*Gyereke.Lány*
$\exists R.\top$	korl. egzisztenciális kvantor	fogalmak, melyeknek létezik R-je pl. *Házas* $=$ *Ember* $\sqcap$ $\exists$*Házastársa*.$\top$

1. táblázat. Az $\mathcal{AL}$ nyelv konstruktorai

„A” helyén csak atomi fogalom, „C” és „D” helyén összetett fogalom is állhat

Az 1. táblázatban látható konstruktorokat használhatjuk az $\mathcal{AL}$ (*attributive language*) nyelvben, mely a legegyszerűbb, még érdekes leíró logikai nyelv.

Az $\mathcal{AL}$ nyelv formális szemantikáját megadhatjuk a következőképpen: legyen az interpretációk halmaza $\mathcal{I}$, melyek egy nem üres $\Delta^{\mathcal{I}}$ halmazból (tárgyterület, *domain*) és egy $\cdot^{\mathcal{I}}$ interpretációs függvényből állnak. Utóbbi minden A atomi fogalomhoz egy halmazt: $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, és minden R atomi szerephez egy bináris relációt rendel: $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Ezáltal tehát megadja az elemi fogalmak és szerepek egy lehetséges jelentését. Az interpretációt kiterjeszthetjük a fogalmak leírásaira a konstruktorok 1. egyenlet szerinti értelmezésével.

$$
\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \varnothing \\
(\neg A)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \forall b.\ (a,b) \in R^{\mathcal{I}} \to b \in C^{\mathcal{I}}\} \\
(\exists R.\top)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \exists b.\ (a,b) \in R^{\mathcal{I}}\}
\end{aligned}
\tag{1}
$$

Az $\mathcal{AL}$ nyelvnek további kiegészítései léteznek: az $\mathcal{U}$ (diszjunkció), $\mathcal{E}$ (teljes egzisztenciális kvantor), $\mathcal{N}$ (számosság korlátozások), $\mathcal{C}$ (teljes negálás) egyenletekkel leírt konstruktorok bármely részhalmazát használhatjuk, ezzel a $\mathcal{AL}[\mathcal{U}][\mathcal{E}][\mathcal{N}][\mathcal{C}]$ nyelvekhez jutunk. Az első látásra $2^4 = 16$ variáció valójában csak 12 különböző nyelvet takar, mivel $\mathcal{C} \rightsquigarrow \mathcal{U}$ $(C \sqcup D \equiv \neg(\neg C \sqcap \neg D))$, $\mathcal{C} \rightsquigarrow \mathcal{E}$ $(\exists R.C \equiv$

$\neg\forall R.\neg C$), valamint $\mathcal{UE} \rightsquigarrow \mathcal{C}$ (negációs normálformával átalakítható), így az $\mathcal{ALC}$, $\mathcal{ALCE}$, $\mathcal{ALCUE}$, $\mathcal{ALUE}$ nyelvek megegyeznek.

$$
\begin{array}{rcll}
(C \sqcup D)^{\mathcal{I}} & = & C^{\mathcal{I}} \sqcup D^{\mathcal{I}} & \mathcal{U} \\
(\exists R.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} & \mathcal{E} \\
(\geqslant nR)^{\mathcal{I}} & = & \big\{a \in \Delta^{\mathcal{I}} \bigm| |\{b \,|\, (a,b) \in R^{\mathcal{I}}\}| \geq n\big\} & \\
(\leqslant nR)^{\mathcal{I}} & = & \big\{a \in \Delta^{\mathcal{I}} \bigm| |\{b \,|\, (a,b) \in R^{\mathcal{I}}\}| \leq n\big\} & \mathcal{N} \\
(\neg C)^{\mathcal{I}} & = & \Delta \setminus C^{\mathcal{I}} & \mathcal{C}
\end{array}
$$

Természetesen adódik, hogy ne csak a fogalmaknak, hanem a szerepeknek is legyenek konstruktoraik, így atomi szerepek mellett komplex szerepeket is készíthetünk. Felírhatjuk például a szerepek unióját, metszetét, komplemensét és kompozícióját, a szerepek inverzét ($\mathcal{I}$) vagy tranzitív lezártját ($\mathcal{R}^{+}$).

$$
\begin{array}{rcll}
(R^{-})^{\mathcal{I}} & = & \{(a,b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (b,a) \in R^{\mathcal{I}}\} & \mathcal{I} \\
(R^{+})^{\mathcal{I}} & = & \bigcup_{n \geq 1} (R^{\mathcal{I}})^{n} & \mathcal{R}^{+} \\
(\geqslant nR.C)^{\mathcal{I}} & = & \big\{a \in \Delta^{\mathcal{I}} \bigm| |\{b \,|\, (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}| \geq n\big\} & \\
(\leqslant nR.C)^{\mathcal{I}} & = & \big\{a \in \Delta^{\mathcal{I}} \bigm| |\{b \,|\, (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}| \leq n\big\} & \mathcal{Q}
\end{array}
$$

Ha megengedjük az $R \sqsubseteq S$ alakú állítások használatát (minden apa egyben szülő is), eljutunk a szerephierarchiákhoz ($\mathcal{H}$). Az $\mathcal{N}$-nel jelölt számosság korlátozások általánosításai a minősített számosság korlátozások, melyek csak egy adott fogalmon belüli egyedekre vonatkoznak ($\mathcal{Q}$). A felsorolt, az $\mathcal{AL}$ nyelvet kiegészítő konstruktorok nagyon hatékonyak a tárgyterületi tudás megfogalmazására. Például $\mathcal{Q}$ segítségével kifejezhetjük, hogy *a hódítóknak legalább két kék szemük van*:

$$\textit{Hódító} \sqsubseteq\ \geqslant 2\ \textit{Szeme.KékSzem}$$

A leíró logikai rendszerek különböző következtetési szolgáltatásokat nyújtanak, a legfontosabbak a következők:

kielégíthetőség vizsgálat: $\Sigma \not\models C \equiv \bot$, vagyis létezik olyan modell, amelyben C nem üres halmaz;

alárendeltség: $\Sigma \models C \sqsubseteq D$, vagyis minden modellben C részhalmaza D-nek;

információ kinyerés: $\{a \mid \Sigma \models C(a)\}$, fogalom példányainak lekérdezése;
realizáció: $\{C \mid \Sigma \models C(a)\}$, példányt tartalmazó fogalmak lekérdezése.

Egy logikai következtető rendszertől feltétlen elvárjuk a helyességet (*soundness*), vagyis hogy minden kikövetkeztetett állítás igaz. Legyen a teljesség (*completeness*) – minden igaz állítás kikövetkeztetésének képessége – szintén elvárt tulajdonság, sőt, a használhatóság érdekében olyan megoldást keresünk, amivel minden állítás eldönthető a gyakorlatban elfogadható válaszidőkkel. A leíró logikák különböző nyelvei más-más kifejezőerővel bírnak, ennek megfelelően más és más a következtetés komplexitása. Az elmúlt tíz évben sokat fejlődtek a következtető algoritmusok, köszönhetően a tableaux-algoritmus különböző változatainak [Baader03]. A tudományos eredmények tükrében kifejlesztett rendszerek, mint a FaCT, RACE és DLP bebizonyították, hogy ugyan a gazdag kifejezőerejű DL-ek következtető algoritmusai nem polinomiális idejűek, mégis a gyakorlatban még nagy tudásbázisokon is megfelelően viselkednek.

4.3.1. $\mathcal{SHIQ}$ nyelv

A kifejezőerő és a következtetés komplexitása közti kompromisszumot jelenleg azok a nyelvek jelentik, melyeknél a következtetés legrosszabb esetben polinomiális tárhelyigény (PSPACE) mellett ugyan exponenciális idejű (EXPTIME), mégis teljes, és az állítások igaz volta a gyakorlatban elfogadható időn belül eldönthető.

Az ontológia nyelvek tervezésekor is figyelembe kell venni ezeket a szempontokat, kiegészítve azzal, hogy olyan konstrukciókat kell választani, melyek a logikát az ontológiai modellezésre különösen alkalmassá teszik [Baader03].

Ide tartoznak például a minősített számosság korlátozások, melyekkel nem csak azt tudjuk kifejezni, hogy például valakinek *két gyereke van*:

$$(\geq 2 \ \mathit{Gyereke})$$

hanem akár azt is, hogy *legalább egy lánya és egy fia van*:

$$(\geq 1 \ \mathit{Gyereke.Fiú}) \sqcap (\geq 1 \ \mathit{Gyereke.}\neg \mathit{Fiú})$$

A szerepek gazdag leírásai szintén nélkülözhetetlenek az ontológiák készítésekor, amikor egyedek csoportjának közös tulajdonságait akarjuk ábrázolni – a komplex szerepek természetesen felhasználhatók a fogalmak definiálásakor. Az inverz szerep használatával a *Gyereke* mellett használhatjuk a *Szülője* relációt, az *Őse* egy tranzitív tulajdonság, míg a szerepek hierarchiájának leírásával kifejezhető az *Apja* és *Anyja*

viszonya a *Szülője* relációhoz.

A fentieknek megfelelő logika az $\mathcal{SHIQ}$, mely a következőképpen épül fel [Baader03]: a szerepek neveinek halmaza legyen $\mathbf{R}$, melynek két partíciója a tranzitív ($\mathbf{R}_+$) és nem tranzitív ($\mathbf{R}_P$) szerepek. Az összes $\mathcal{SHIQ}$ szerep tartalmazza a nevesített szerepek inverzeit is: $\mathbf{R} \cup \{r^- \mid r \in \mathbf{R}\}$, ahol egy szerep inverzének inverze önmaga ($r^{--} \equiv r$). Használhatunk szerep hierarchiát: megadhatjuk $r \sqsubseteq s$ ($r, s \in \mathbf{R}$) alakú állítások egy véges halmazát.

Az $\mathcal{SHIQ}$ nevében az $\mathcal{S}$ az $\mathcal{ALC}$ logikának a tranzitív szerepekkel történő kiegészítését ($\mathcal{R}^+$) jelenti. A $\mathcal{H}$ a szerep hierarchiára, az $\mathcal{I}$ az inverz szerepekre, a $\mathcal{Q}$ a minősített számosság korlátozásokra utal, ahogy azt az $\mathcal{AL}$ nyelv kiegészítéseinél is jelöltem. Ha korlátlanul használjuk az összes szerep konstruktort a fogalmak létrehozására, nem eldönthető DL-hez jutunk. Ezért bevezetjük az egyszerű szerep fogalmát, mely nem tartalmazhat tranzitív szerepet, és a minősített számosság korlátozások csak egyszerű szerepekre vonatkozhatnak.

DAML+OIL axióma	*DL szintaxis*	*Példa*
subClassOf	$C_1 \sqsubseteq C_2$	$Férfi \sqsubseteq Ember$
subPropertyOf	$P_1 \sqsubseteq P_2$	$Apja \sqsubseteq Szülője$
disjointWith	$C_1 \sqsubseteq \neg C_2$	$Férfi \sqsubseteq \neg Nő$
transitiveProperty	$P \in \mathbf{R}_+$	$Őse \in \mathbf{R}_+$
uniqueProperty	$\top \sqsubseteq (\leq 1 P.\top)$	$\top \sqsubseteq (\leq 1 Anyja.\top)$

2. táblázat. Néhány DAML+OIL axióma $\mathcal{SHIQ}$ logikai megfelelője

A DAML+OIL ontológia nyelvet (3.1.3. fejezet) tervezésekor pontosan ehhez a leíró logikához igazították, állításai és axiómái az egyedek és adattípusok kivételével átírhatók $\mathcal{SHIQ}$ kifejezésekké. Erre adok néhány példát a 2. táblázatban, az átírás a legtöbb esetben nyilvánvaló.

4.4. Következtetőgépek

Az IKF ontológia komponense által beágyazott ontológia klasszifikálását külső következtetőgép használatával tudjuk megoldani. Áttekintem az ingyenesen is hozzáférhető eszközöket, és megvizsgálom, mennyiben tudják kielégíteni az ontológia kifejezőerejével és a komponenshez való integrálásukkal kapcsolatos követelményeket.

4.4.1. FaCT, FaCT++

Ian Horrocks PhD munkájaként készült el a FaCT[29] rendszer, melyet a tableaux-algoritmus optimalizálásával kapcsolatos kutatásai közben, a módszerek teszteléséhez fejlesztett ki [Horrocks00].

Két következtetőt foglal magában, egyik az $\mathcal{SHF}$ logikát ($\mathcal{ALC}$, tranzitív és funkcionális szerepek, valamint szerep hierarchiák), másik az $\mathcal{SHIQ}$ logikát támogatja, helyes és teljes következtetést nyújtva. Az implementáció igazolta, hogy a tableaux-algoritmus számos kiegészítéssel képes alkalmas válaszidőkkel klasszifikálni $\mathcal{SHIQ}$ nyelvű tudásbázisokat annak ellenére, hogy legrosszabb esetben az algoritmusok exponenciális idejűek.

A rendszer nyílt forráskódú (GPL licenszű), LISP-ben készült, de letölthetőek LISP környezet nélkül is azonnal futtatható, bináris változatai Linux és Windows operációs rendszerekre.

A szolgáltatások eléréséhez négy lehetőségünk van: használhatjuk a logikai következtetők körében szabványosnak tekinthető DIG interfészt; a FaCT DIG implementációját Java alkalmazásba belefordítva közvetlen, memórián belül elérhetjük; a FaCT hagyományos, CORBA felületű implementációját távoli eljáráshívással is kezelhetjük; vagy LISP nyelvű implementációval közvetlenül használhatjuk.

A FaCT rendszer hiányossága, hogy csak terminológiai következtetést nyújt, nem támogatja egyedek (ABox) használatát. Másrészt egy kísérleti, a tudományos eredmények empirikus igazolását célzó eszközről van szó, melyet készítője nem kíván továbbfejleszteni. Néhány éve az egyik legjobb alternatívának számított, például az OilEd ontológia szerkesztő (3.2.1. fejezet) használta ontológiák konzisztencia ellenőrzésére, helyét idővel átveszik az újabb megoldások.

A félbehagyott fejlesztés helyét átveszi annak C++ nyelvű, továbbfejlesztett implementációja, mely azonban egyelőre még kísérleti fázisban van.[30] A FaCT++ követ-

[29] **Fa**st **C**lassification of **T**erminologies, `http://www.cs.man.ac.uk/~horrocks/FaCT/`

[30] Reasoner Prototype, FaCT++, `http://wonderweb.semanticweb.org/deliverables/D13.shtml`

keztetőgép egyelőre csak az OWL Lite nyelvet támogatja, de a fejlesztők a végleges változattal az OWL DL-t célozzák meg.

4.4.2. RACER

A RACER leíró logikai következtető rendszert Volker Haarslev[31] és Ralf Möller[32] fejlesztették ki. Különösen érdekes a Szemantikus Web alkalmazásai számára, mivel az $\mathcal{SHIQ}(\mathcal{D}_n)^-$ leíró logikát támogatja, mely jelenleg az $\mathcal{SHIQ}$ egyik legbővebb kiegészítése, amihez még léteznek megfelelően optimalizált következtető rendszerek [Haarslev03b]. A $\mathcal{D}$ a *concrete domains* támogatását jelenti, mellyel kezelhetőek a DAML+OIL illetve OWL adattípusai, mivel lehetővé teszi egész, racionális és valós számok használatát és köztük relációk megadását.

Két interfészen keresztül érhetőek el a szolgáltatásai. Egyik a *FaCT* esetében már említett DIG,[33] mely leíró logikai következtetők és kliens programok összekapcsolására született szabványos protokoll. HTTP protokollon keresztül XML-be csomagolt kéréseket küld, de nem támogathatja az összes, rendszer-specifikus állítást és lekérdezést. Ilyenek például bizonyos lekérdezések hatékonyságát növelő indexszámítások.

Ezért a RACER elérhető egy TCP alapú interfészen is, mely lényegében a KRSS[34] szabványon alapul, néhány kiegészítéssel és megszorítással. Nagy előnye, hogy gyorsabb, mivel mentes a HTTP és XML okozta többletszámítástól (*overhead*), valamint a tömör parancsokat bonyolult kliens nélkül, akár egy *telnet* kliens segítségével is kiadhatjuk. (Utóbbi a parancsok megfelelő ismerete mellett fejlesztéskor a leghatékonyabb módja az ellenőrzésnek.)

A RACER tudásbázisának vizualizációját segíti a párhuzamosan fejlesztett *RICE*, mely a fogalmak taxonómiájának és az ABox-beli struktúrák megjelenítésére képes. TCP interfészen keresztül kapcsolódik, így képes például egyetlen kéréssel lekérni a teljes fogalom hierarchiát, ez a DIG interfésszel csak sok egymás utáni kéréssel – a taxonómia bejárásával – lenne lehetséges.

További egyedi szolgáltatása, hogy a kapcsolódó kliensek lekérdezéseket regisztrálhatnak bizonyos ABox-okhoz (*publish/subscribe interface*). Ilyenkor a *Racer Server* üzenetet küld, ha a korábban regisztrált példány-lekérdezés megoldáshalmaza bővül [Haarslev03a].

A szerver optimalizációját úgy alakították ki, hogy eltérő igényű alkalmazások

[31]Concordia University, Montreal, Kanada

[32]Technische Universität Hamburg-Harburg, Németország

[33]DIG (DL Implementors Group) interface, `http://dl-web.man.ac.uk/dig/`

[34]Knowledge Representation System Specification, `http://www.bell-labs.com/user/pfps/papers/krss-spec.ps`

esetén is megállja a helyét. A kliens lehet, hogy folyamatosan újabb és újabb ABox-okat készít, mindegyiken csak néhány lekérdezést futtatva. Máskor sok lekérdezést futtat egy többnyire statikus tudásbázison. A *Racer Server* mindkét esetet támogatja: alapesetben a lehető legkevesebb erőforrás felhasználásával válaszolja meg a lekérdezéseket, viszont utasítható az indexek előzetes számítására, ha sok lekérdezést kell megválaszolnia. Az ilyen számítások nagy tudásbázisok esetén akár órákig is eltarthatnak, ezért az adatbázisokhoz hasonló módon támogatja a statikus tudásbázisokhoz számított indexek perzisztens tárolását, melyek a szerver újraindításakor újra betölthetők.

A *Racer Proxy* segítségével elosztott rendszert hozhatunk létre: a proxy-n keresztül akár több kliens is elérheti ugyanazt a tudásbázist, a proxy biztosítja a hozzáférések szinkronizálását. Hasonlóan, több *Racer Server*-t is használhatunk egy proxy-n keresztül, ilyenkor a kéréseket mindig szabad szervereknek továbbítja, biztosítva a terhelésmegosztást (*load balancing*).

4.4.3. Vampire

A Vampire[35] egy kísérleti, elsőrendű logikai következtetőgép, melyet Andrei Voronkov és Alexandre Riazanov fejlesztenek. Céljuk egy olyan környezet kialakítása, melyben elsőrendű logikai algoritmusok hatékonyságát tesztelhetik, másrészt időközben gyakorlati feladatok megoldására is alkalmas következtetőgépet sikerült létrehozniuk.

Sikerüket bizonyítja, hogy szoftverük a CASC[36] elsőrendű logikai tételbizonyítási világbajnokság sokszoros győztese.

Hiába léteznek azonban hatékony elsőrendű logikai következtetőgépek, speciálisan a leíró logikákra továbbra sem érdemes alkalmazni őket. Ezt a Vampire és a FaCT++ összehasonlításával Tsarkov és Horrocks kísérletileg is igazolta [Tsarkov03]. A leíró logikai állításokat elsőrendű logikába transzformálták, elvégezve a kézenfekvő optimalizálásokat. Az eredmények alapján a Vampire nagy tudásbázisok esetén két nagyságrenddel lassabbnak bizonyult, ráadásul nem is nyújtott teljes következtetést: memória korlátok miatt a negatív tesztek (nem-tartalmazás megállapítása) harmadát nem tudta megoldani.

A leíró logikai következtetőgépek helyett tehát nem alkalmazhatóak, de velük párhuzamosan annál inkább. A komplex leíró logikákat hibrid következtetőkkel célszerű kezelni, ha még nem léteznek a teljes nyelvet lefedő DL algoritmusok.

[35] `http://www.cs.man.ac.uk/~riazanoa/Vampire/`

[36] The CADE ATP System Competition, `http://www.cs.miami.edu/~tptp/CASC/J2/`

4.4.4. Pellet

A *Pellet*[37] egy OWL DL következtető, melyet a Mindswap Group[38] fejleszt. Céljuk, hogy helyes, teljes, eldönthető és a gyakorlatban hatékony algoritmust adjanak a következtetésre az OWL nyelv Lite nyelvjárásában, és az OWL DL minél nagyobb részében. Az OWL három nyelvjárása, ahogy a 3.1.5. fejezetben leírtam, az OWL Lite, OWL DL és OWL Full. A Lite esetében a kitűzött célok már megvalósultak, a DL-ről még nem tudjuk biztosan, lehetséges-e ilyen következtetőt készíteni. A *Pellet* az $\mathcal{SHIN}(\mathcal{D})$ leíró logikával következtet, mely az OWL DL egy másik maximális, még kezelhető részhalmaza – nem támogatja a *owl:oneOf* és *owl:hasValue* struktúrákat. Az OWL Full meta-modellezési képességei miatt nem eldönthető, de a *Pellet* a jövőben valamilyen szinten támogatni kívánja, például az RDFS(FA) résznyelvét, melynek létezik leíró logikai megfelelője [Pan01]. A készítők célul tűzték ki a W3C OWL teszteseteinek[39] megoldását, melyeket az OWL következtetők hatékonyságának tesztelésére hoztak létre.

A következtetőt Java-ban implementálják, forráskódja hozzáférhető. Készítői hangsúlyozzák, hogy egyelőre befejezetlen állapotban van, a következtető még nem teljes. Két külső könyvtárat használtak: az OWL elemzését végző Jena2-t (3.3.2. fejezet) és a fa struktúrájú absztrakt adattípusok hatékony tárolását végző ATerm könyvtárat.[40]

4.4.5. TRIPLE

Stefan Decker[41] és Michael Sintek[42] közös munkája a TRIPLE[43] RDF lekérdező, következtető és transzformációs nyelv [Sintek02]. Nem rendelkezik fix szemantikával, hanem szabályok segítségével írhatunk le tetszőleges, az RDF fölé építkező nyelvet. Így egyesíti azon adatmodelleket, melyeknek közös reprezentációs nyelve az RDF. Ilyenek lehetnek az UML, a Topic Map-ek, vagy akár a DAML+OIL.

A szemantika leírására a TRIPLE belső nyelve a Horn logikán alapul, melyet kiegészítették az RDF kezeléséhez szükséges hármasok (*triples*, innen kapta a nevét) kezelésével. Ha a megcélzott adatmodell nem írható le Horn logikával, a modellt modulként kell hozzáilleszteni, mely egy külső következtető komponenst használ.

[37] Pellet OWL Reasoner, `http://www.mindswap.org/2003/pellet/index.shtml`

[38] The MINDSWAP Group, `http://www.mindswap.org/`

[39] `http://www.w3.org/TR/owl-test/`

[40] ATerm (Annotated Term) library, `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary`

[41] Stanford University Database Group

[42] DFKI GmbH, Kaiserslautern, Knowledge Management Department

[43] `http://triple.semanticweb.org/`

Az RDF séma például közvetlenül implementálható a TRIPLE belső nyelvével, a DAML+OIL ontológia nyelvet viszont csak külső modul segítségével támogatja, mely a *RACER* vagy *FaCT* következtetőt használja.

A TRIPLE a FaCT-hez hasonlóan szintén nem élő projekt, fejlesztését 2002. márciusában abbahagyták. Így nem várható el, hogy például az újabb ontológia nyelveket, mint az OWL, támogatni fogja.

4.4.6. Választás indoklása

Az IKF ontológia modul feladata OWL nyelvű ontológia klasszifikálása. Az IKF jelenlegi, kísérleti ontológiája csak fogalmakat és tulajdonságokat ír le, de nem zárható ki, hogy a jövőben példányokat is fog tartalmazni. Ezért nem elegendő egy TBox következtető használata, feltétlen hibrid (TBox és ABox támogatást is nyújtó) következtetőgépre van szükség.

Az OWL nyelvjárások közül egészen biztos, hogy az OWL DL-en belül fog maradni az ontológia – most is ebbe az osztályba tartozik. Az információkinyerést és -feldolgozást végző IKF rendszer többi komponense intenzíven fogja használni az ontológiát szolgáltató következtetőgépet, így a megfelelő válaszidők elérése miatt meg kell maradni a kisebb számításigényű DL nyelvnél.

Mindezekből következik, hogy elsőrendű logikára támaszkodó következtetőt – mint a *Vampire* – nem használhatok, mivel az erős modellezési képességekhez nem eldönthető logika, nem elegendő teljesítmény társul.

A *FaCT* nem hibrid következtető, ráadásul idejétmúlt megoldásnak is számít. Készítője kísérleti megoldásnak szánta, annak ellenére, hogy az OilEd ontológia szerkesztő idejében az egyik legjobb alternatívának számított.

A további alternatívák közül a *TRIPLE*-t csak akkor érdemes használni, ha nincs szükség külön következtető bevonására – a választott formalizmus leírható a belső nyelvével –, vagy integrálási feladat keretében több, különböző nyelvű tudásbázist egységes keretben kell kezelni.

A *RACER* kitűnik sokrétű szolgáltatásaival, melyek közül számosat az ontológia modul esetében is fel tudok használni: gyors TCP interfésszel rendelkezik, melyet Java és C++ nyelvű API-val is el lehet érni, valamint OWL ontológiát képes importálni. Tartalmazza a legújabb optimalizálási módszerek jelentős részét és folyamatos fejlesztés alatt áll – körülbelül havonta jelenik meg újabb verziója. Az igénybe vehető számítási teljesítményt az elosztott futtatás lehetősége tovább növeli.

Ugyan nem támogathatja az OWL DL minden állítását, egyes formalizmusok a következtetés komplexitásának kézben tartása végett hiányoznak. Egyelőre nem vár-

ható el egyetlen hasonló sebességű következtetőtől sem a teljes OWL DL támogatása.

A *Pellet* ígéretes megoldás, de egyelőre nem készült el. Ugyan még jó ideig biztosan nem fogja utolérni a sokkal nagyobb múlttal rendelkező RACER teljesítményét, az a tény, hogy az OWL nyelvet célozzák meg, idővel kedvezőbb választássá teheti az OWL használói körében.

Mindezen szempontokat figyelembe véve a RACER-t választottam az ontológia modul következtető motorjául.

5. Tervezés

A diplomaterv kiírásból kiindulva, az IKF rendszerprototípus fejlesztőivel folytatott beszélgetések alapján alakítottam ki az ontológia modul feladatait.

A diplomaterv keretében megvalósítandó feladatom az IKF rendszerprototípus számára olyan komponens létrehozása, mely az információkinyerés és -feldolgozás közben használt ontológia *kezelését végzi*, és ezt az ontológiát a többi modul felé *lekérdezhetővé teszi*.

Függőségek kezelése. Az ontológia kezelése az ontológia betöltését, a több részből felépített, moduláris ontológiák egyes részei közti függőségek kezelését valamint a klasszifikáció elvégzését jelenti. A modulok függőségeit az OWL nyelvű ontológia *owl:imports* kijelentése által értelmezem, mely a következő szemantikával bír [OWL referencia]:

> *„An owl:imports statement references another OWL ontology containing definitions, whose meaning is considered to be part of the meaning of the importing ontology.”*

Tehát ha például egy *B.owl* ontológia tartalmazza az alábbi kijelentést:

```
<owl:Ontology rdf:about="">
  <owl:imports rdf:resource="http://domain.hu/A.owl"/>
</owl:Ontology>
```

akkor az ontológus szándéka szerint csak az *A.owl* ontológia tartalmának figyelembe vételével értelmezendő. Ekkor nevezem a B ontológiát az A-tól függőnek. A függőséget az OWL referenciának megfelelően tranzitív relációként értelmezem.

Ha egy ontológia csak hivatkozik egy másik névtérben található entitásra, akkor azt még nem tekintem kielégítendő függőségnek:

```
<rdf:RDF
  xmlns:ontoa="http://domain.hu/A.owl#">
  <ontoa:alma rdf:ID="golden"/>
</rdf:RDF>
```

Ennek legfőbb oka, hogy nem derül ki, hol található a külső entitás leírása. Csak a konvenciók alapján feltételezhető, hogy ha leválasztjuk az URI fragmentumát (#*alma*), az A ontológia címéhez jutunk. Előfordulhat azonban, hogy egy egész máshol elérhető ontológia írja le a hivatkozott entitást.

Másrészt ha külön-külön is használható ontológiákat néha egy egészként is le akarunk kérdezni, *owl:equivalentClass* definíciókkal összekapcsolhatjuk őket anélkül, hogy a komponens megkövetelné az együttes használatukat.

Klasszifikáció elvégzése. A moduláris ontológia függőségeit figyelembe véve a betöltött modulok egy részét kívánjuk a komponensen keresztül lekérdezhetővé tenni. A lekérdezések kiszolgálását egy következtetőgép biztosítja, mely az ontológiát klasszifikálja, azaz az implicit relációkat kikövetkezteti és közvetlenül elérhetővé teszi.

Először az ontológia konzisztenciájáról győződünk meg, csak konzisztens ontológiát fogadunk el. A TBox konzisztenciájának (fogalmak kielégíthetőségének) és az ABox ellentmondás-mentességének ellenőrzése sokkal gyorsabb művelet, mint a teljes klasszifikáció, ezért célszerű előbb elvégezni az ellenőrzést.

A modul automatikusan klasszifikálja az ontológiát, mivel csak utána futtathatók lekérdezések a következtetőgépen [RACER referencia]. Mivel a komponens szolgáltatásai nem módosítják az ontológiát, illetve a szemantikáját tükröző tudásbázis tartalmát, a nagy tudásbázisok esetén meglehetősen hosszadalmas műveletre csak egyszer kell sort keríteni. A tudásbázis tartalma csak ritkán, az adminisztrátor beavatkozásakor változik, míg a lekérdezések az IKF alkalmazás működése során folyamatosan futnak.

Lekérdezhetőség biztosítása. Azok a modulok, melyek az IKF ontológiáját használni kívánják, az ontológia modulhoz kapcsolódhatnak, így nem kell külön-külön implementálni az ontológia betöltésével kapcsolatos teendőket. Az ontológia lekérdezését tehát minél egyszerűbb, kényelmesebb módon kell lehetővé tenni.

Az IKF rendszer elosztott szemléletét figyelembe véve egy platformfüggetlen, távoli elérésen alapuló interfészt célszerű kialakítani. A klasszifikált ontológiát ezért egyszerű, a leíró logikai következtető szolgáltatásait közvetítő távoli függvényhívásokkal teszem lekérdezhetővé a SOAP protokollon (6.1.4) keresztül.

Az ontológia kezelő komponens szolgáltatásait párhuzamosan több modul is használhatja, ezért ügyelni kell az egy időben befutó kérések helyes kezelésére.

5.1. Modul specifikáció

A modul tervezését az előbbiekben körvonalazott lényeges feladatok alapján a specifikáció elkészítésével kezdtem. Itt pontosabban leírom a megvalósítandó funkciókat, a komponens által nyújtott szolgáltatásokat.

Az ontológiát beágyazó modul feladatai:

- moduláris ontológia betöltése,
- az ontológia modulok függőségének feltérképezése, kezelése,
- a moduláris ontológia betöltése a következtetőgépbe,
- klasszifikáció elvégzése,
- a következtető leíró logikai szolgáltatásainak közvetítése,
- konkurens kérések kiszolgálása.

Ontológia nyelv. Az ontológia nyelve az OWL, mely a Szemantikus Web legújabb, általánosan elfogadott, most már a W3C ajánlásává emelkedett szabványa. Lehetőséget ad az entitások logikai formulákba öntött, pontos szemantikus ábrázolására, valamint támogatja moduláris ontológiák építését, ahol az egyes modulok:

- hivatkozhatnak más modulokon belüli entitásokra,
- egész modulokat illeszthetnek be a saját készletükbe.

Ontológia betöltése. Az ontológia modulokat *file*-ból vagy *http* protokollon keresztül, az URL megadásával tölthetjük be az ontológiakezelőbe. Ha a *file*-ból betöltött ontológia nem definiálja a bázis URI-t (*base URI*), mely a relatív URI-k feloldására szolgál, azt kézzel is megadhatjuk.

Az ontológiaszerkesztő nem része a komponensnek, a későbbiekben azonban lehetőség van annak megoldására, hogy az ontológiát közvetlenül a szerkesztőből tölthessük be. Ez a jelenleg használt szerkesztő (*Protégé*) esetén *plug-in* írásával lenne lehetséges.

Az importált modulokat a komponens lokálisan tárolja, feljegyzi az utolsó frissítés dátumát, és megállapítja az OWL modul nyelvjárását (*Lite*, *DL*, vagy *Full*). A komponens újraindításával visszaállítja az eredeti konfigurációt, így nem szükséges a modulokat újra betölteni. Lehetőség van továbbá a modulok frissítésére és eldobására is.

Függőségek kezelése. A szolgáltatott ontológia-lekérdező API-n keresztül a betöltött modulok összességén futtathatunk lekérdezéseket. A betöltött modulokat azonban a lekérdezhetőségük előtt *aktiválni* kell, ez jelenti a következtetőgépbe való betöltésüket.

A modulok aktiválásához biztosítani kell, hogy az általuk importált modulok szintén be legyenek töltve, illetve aktiválva legyenek. Ennek biztosítását jelenti a *modulok függőségének kezelése.* A függőséget az OWL dokumentum *owl:imports* állításai alap-

ján határozom meg. Az ontológia függőségi gráfja az ontológia komponens grafikus felületén keresztül megtekinthető, illetve az egyes függőségek kikapcsolhatóak.

Egy ontológia függőségei kielégítettek, ha már minden hivatkozott ontológiája aktiválva van. Csak kielégített függőségek esetén lehet egy betöltött ontológiát aktiválni, azaz az API-n keresztül lekérdezhetővé tenni. Az OWL nem zárja ki körkörös tartalmazás használatát, az ilyen szituációt függőségek kikapcsolásával lehet feloldani.

Klasszifikáció. Az ontológia modulok aktiválásukkor betöltődnek a következtetőgépbe, kiegészítve a már aktivált modulokból álló tudásbázist.

A komponens minden aktiválás után automatikusan elvégzi a tudásbázis klasszifikálását és a konzisztencia ellenőrzését. Ha inkonzisztencia lép fel, az aktiválást visszavonja, automatikusan visszaállítva a tudásbázis eredeti állapotát.

Leíró logikai szolgáltatások közvetítése. A modulok aktiválása után rendelkezésre áll a következtetőgépben a klasszifikált, garantáltan konzisztens tudásbázis. A következtetőgépen futtatható, az IKF többi komponense számára érdekes lekérdezéseket szolgáltatja tovább az ontológia komponens.

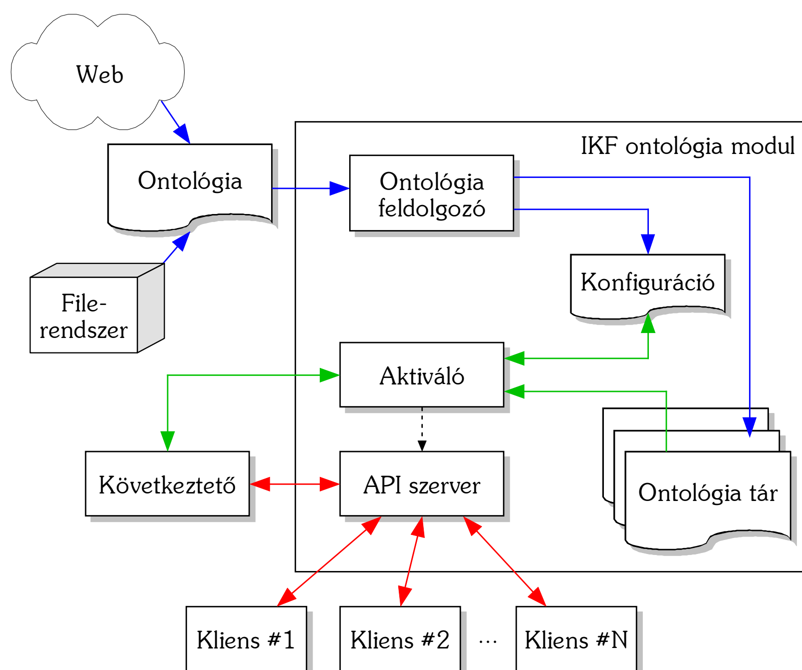
A szolgáltatások a szokásos ontológia-lekérdező primitívekből állnak, melyek alkalmasak az ontológia fogalmainak, tulajdonságainak és példányainak, valamint a közöttük lévő kapcsolatoknak a lekérdezésére.

A modul a szolgáltatásokat olyan interfészen keresztül kell nyújtsa a többi modul felé, melyet különböző platformon futó modulok (pl. Miscrosoft .NET valamint Java alkalmazások) is elérhetnek. Egy időben több kliens is csatlakozhat, ezért konkurens kéréseket is helyesen ki kell tudjon szolgálni.

5.2. Architektúra

A részletes feladat specifikációból már levezethető a modul felépítése. A komponens kisebb, belső egységeit és az adatmozgatás fő irányait mutatja a 11. ábra.

A Webről vagy a filerendszerből beolvasott ontológiát először fel kell dolgozni: a szintaktikai ellenőrzés után meg kell állapítani az ontológia függőségeit. A komponens leállítása majd újraindítása után a beolvasott ontológiának rendelkezésre kell állnia, ezért az egy perzisztens *ontológia tárba* kerül. A beolvasás után egy, a *konfigurációt* karbantartó egység eltárolja az ontológia adatait (neve, forrása, frissítés ideje) és függőségeit. A konfiguráció szintén perzisztens, mivel a komponens leállításakor egy lokális file-ba kerül, indításakor újra beolvasható.



11. ábra. Az IKF ontológia modul architektúrája

A kék nyilak az ontológia betöltését, a zöld nyilak az aktiválást (következtetőbe betöltést), míg a pirosak a kérés-kiszolgálást jelzik.

Az ontológia *aktiválása* előtt a konfiguráció alapján ellenőrizni kell függőségei teljesülését. Ha aktiválható, az ontológia tárból töltődik a *következtetőgépbe*. Az aktiválást végző egység a következtető lekérdezésével ellenőrzi a tudásbázis konzisztenciáját, ellentmondás esetén a műveletet visszavonja. Mivel a tudásbázis a bővítése közben nem a kívánt tartalmat tükrözi, az ontológia szolgáltatását végző API szervert blokkolni kell. Ezt jelzi az ábrán a szaggatott nyíl.

A tudásbázis lekérdezhetőségét biztosítja az *API szerver*. Egyszerre több *kliens* is kapcsolódhat hozzá, de a következtetőgépet csak egy csatornán keresztül használja.

5.3. Függőségek kezelése

A moduláris ontológiák az IKF rendszer tesztelése közben lehetőséget adnak különböző tartalmú – például más-más részletességű – tudásbázisok használatára. Így például kísérletekkel könnyen kimutatható, hogy az ontológia egyes részei hogyan befolyásolják a tudáskinyerési folyamatot, javítják vagy lerontják annak pontosságát, hatékonyságát.

Az ontológia modulok függőségének kezelése biztosítja, hogy ne kerüljön be véletlenül a tudásbázisba egy modul az általa importált modulok nélkül. A függőségek kézbentarthatósága miatt ugyanis az ontológiák beolvasásakor nem próbáljuk meg automatikusan letölteni az *owl:imports* által megadott URI alapján a beillesztett on-

tológiát, hanem azokat a komponens kezelője manuálisan illesztheti be. A nagyobb felhasználási szabadság érdekében a függőségek egyesével kikapcsolhatóak, így az ontológia forrásának módosítása (az *owl:imports* direktíva törlése) nélkül, szabadon kipróbálhatóak például egy kísérleti ontológia moduljainak különböző verziói.

Mint ahogy a fejezet elején, a komponens feladatainak leírásakor kifejtettem, az ontológiák függősége az OWL referenciának megfelelően tranzitív tulajdonság. Egy ontológia modul aktiválásakor ezért biztosítani kell, hogy függőségeinek tranzitív lezártja is aktiválva legyen. Mivel azonban a modulok csak egyesével aktiválhatóak, a közvetlen függőségek aktív állapota már biztosítja az előbbi feltételt.

Persze felmerül a kérdés, miért ragaszkodunk ahhoz, hogy csak egyesével lehessen aktiválni modulokat? Hiányzó függőségek esetén például a komponens felajánlhatná, hogy a tranzitív lezárt nem aktív tagjait automatikusan betölti. Ezzel a körkörös függőségek is kezelhetőek lennének a kör megbontása – megfelelő függőség kikapcsolása – nélkül. A lehetőséget mégis elvetettem, mivel inkonzisztencia fellépésekor nem tudnánk annak pontos okát. Nem kapnánk arra vonatkozó információt, melyik modul milyen már aktivált modulok mellett okozza az ellentmondást.

A függőségek áttekintéséhez kézenfekvő irányított gráffal történő ábrázolásuk. Ezért az egyes ontológia modulok függőségeinek megjelenítésén túl célszerű egy összevont függőségi gráfot is megjeleníteni. A gráfon színekkel vizualizálható a modulok és a függőségi élek állapotai is. A modulok lehetséges állapotai:

nem betöltött olyan modul, ami még nincs betöltve, de egy másik modul függőségei között megtalálható, ezért szerepel a gráfon;

betöltött a komponensbe betöltött, de még nem aktivált modul;

aktivált a következtetőgépbe is betöltött, a tudásbázis részét képező modul.

A függőségi élek állapotai:

kikapcsolt az adminisztrátor szándéka szerint figyelmen kívül hagyandó függőség;

betöltendő a függő modul még nincs is betöltve a komponensbe;

aktiválandó a függő modult aktiválni kell;

teljesülő az importált modul már aktív.

Így azonnal leolvasható a betöltendő moduláris ontológia szerkezete, illetve az egyes modulok aktiválásához szükséges műveletek.

5.4. Ontológia lekérdező API

Az IKF ontológia komponens célja, hogy az aktivált moduláris ontológia tartalmát a többi komponens lekérdezhesse. A szolgáltatott lekérdező felületet tehát úgy kell

kialakítani, hogy a következtetőgépbe töltött ontológia valamennyi lényeges entitásának elérését támogassa. Hasonlóan fontos szempont a távoli elérés és az architektúrától független protokoll, ezáltal az IKF rendszer elosztott jellegének megfelelően bármely másik gépen futó kliensek is kapcsolódhatnak.

A lekérdezéseket a leíró logikai következtetőgép szolgáltatás-primitíveinek megfelelően alakítottam ki. Megvizsgáltam a 3.3. fejezetben bemutatott, ontológiák elérését biztosító API-k függvényeit is, és megállapítottam, hogy a lekérdező primitívek lefedik a szükséges függvények halmazát.

A függvények hívásakor és az eredmények értelmezésekor az entitások az abszolút URI-juk alapján azonosíthatóak: egyértelmű, szabványos karakterfüzérek, melyek egyben a következtetőgép tudásbázisában szereplő entitások azonosítói is. Ha a kliensek valamely névtéren belül dolgoznak, maguknak kell megoldaniuk az URI-k átalakítását.

A komponens API szervere által szolgáltatott függvények teljes listája megtalálható az A. függelékben.

Ha az IKF rendszerben felmerül igényként egy összetettebb lekérdezés egyetlen kérésként történő kiszolgálása – például körvonalazódik a ténybázis típusfájának kezelése –, akkor azt célszerű egy kérésben kiszolgálni, mivel a következtetőgép összetett lekérdezésekre is tud válaszolni.

5.5. Konkurens kérések kiszolgálása

Mivel a következtetőgépek általában nem alkalmasak párhuzamos kérések megválaszolására, a beérkező lekérdezések kiszolgálásához konkurencia-kezelés szükséges.

Ha a következtetőgép foglalt, a kéréseket várakoztatni kell. A kiszolgálásuk sorrendje tetszőleges lehet: mivel a tudásbázist nem módosítják, hatásuk a sorrendtől függetlenül ugyanaz marad.

A lekérdezések sorosítását azonban nem elegendő a hívások kölcsönös kizárásával megvalósítani, mivel attól még fennáll a kiéheztetés lehetősége: elméletileg előfordulhat, hogy valamelyik hívás a következtetőgép foglaltsága miatt soha nem kerül kiszolgálásra.

A probléma áthidalása érdekében kézenfekvő a lekérdezéseket sorba állítani, így a várakoztatásuk közben eltelt idő is hasonló lesz. Minden bejövő kérés egy lista végére kerül, melynek elejéről folyamatosan szolgálja ki őket a szerver. A sorrend tehát FIFO (*first in, first out*) – aki elsőként bejött, elsőként távozik.

Támogathatunk a szerver túlterheltségét kezelő stratégiákat is, a kérés-lista lehetővé teszi a várakozók számának és a várakozási időnek a korlátozását. A várakozók

száma legfeljebb annyi lehet, ahány különböző szálon kérdezik le a kliensek az ontológia modult. A várakozási idő számottevően tehát csak akkor szökhet fel, ha egy lekérdezést nagyon sok ideig tart megválaszolni. Ilyen kérés lehet például az ontológia összes fogalmának lekérése.

Az IKF rendszer helyes működése mellett persze minden lekérdezést meg kell tudni válaszolni, ezért a fentiek miatt legfeljebb egy-egy lekérdezés kiszolgálásának maximális idejét célszerű korlátozni.

6. Implementáció

Az IKF ontológia modult funkcióinak pontos megtervezése után olyan eszközök segítségével implementáltam, hogy az IKF keretrendszerbe könnyen beilleszthető legyen. Először áttekintem, melyek ezek az eszközök, választott technológiák, melyek segítenek a komponens elkészítésében. Utána bemutatom az implementáció lényeges részeinek megvalósítását, UML diagrammokkal szemléltetem az egységek működését.

Végül egy mintapéldán keresztül bemutatom a komponens működését, valamint a teljesítményére vonatkozó méréssel támasztom alá alkalmazhatóságát.

6.1. Választott technológiák

Az ontológiához és az ontológiai következtetéshez kapcsolódó technológiákat már részletesen bemutattam a 3. és a 4. fejezetben. Részleteztem, miért az OWL ontológia nyelvet, az OWL nyelvű dokumentumok kezelésére hivatott OWL API-t és a RACER következtetőgépet választottam feladatom megoldásához. Most a programozási nyelv kiválasztásán túl ezen eszközöknek csak az implementációval kapcsolatos előnyös tulajdonságaira térek ki, valamint röviden ismertetem a szolgáltatott lekérdező interfész megvalósításához használt SOAP technológiát.

6.1.1. Java

A programozási nyelv kiválasztásánál fontos szempont volt az IKF keretrendszer jellegéből adódóan a platformfüggetlenség elérése. Az IKF rendszer fejlesztésének célja ugyanis elkészíteni azokat az eszközöket, melyeket majd a keretrendszerből származtatott alkalmazás létrehozásakor az egyedi igényekhez lehet szabni.

Az ontológia kezelését végző komponens további jellegzetessége, hogy fejlett felhasználói felülettel (GUI) kell rendelkeznie, ehhez a programozási nyelv könyvtárainak megfelelő támogatást kell nyújtaniuk.

A platformfüggetlenséget legnagyobb mértékben jelenleg a Java nyelv támogatja, ráadásul a többi kívánalomnak is megfelel. A Java Swing könyvtárral hatékonyan és könnyen hozhatóak létre fejlett felületek, valamint a nyelv rendelkezik az objektumorientált nyelvekre jellemző valamennyi modellezési előnnyel.

6.1.2. OWL API

Az ontológiákat tároló dokumentumok kezelését megvalósító programozási felületnek a komponensen belül az ontológiák beolvasását, mentését, valamint a függőségek

kinyerését kell támogatnia. Továbbá előnyös lehet, ha a következtetés szempontjából lényeges információkat tud nyújtani az ontológiáról.

Az OWL API támogatja az OWL ontológiák két legelterjedtebb formátumát, az RDF/XML alapú tárolást és az absztrakt szintaxist, és könnyen bővíthető más szintaxis kezelését támogató osztályokkal. A függőségeket jelentő *owl:imports* állítások könnyen lekérdezhetőek, de amint azt a 6.2. fejezetben kifejtem, a függőségek automatikus kezelése miatt módosítanom kellett a könyvtárat. Hasznos tulajdonsága még az OWL nyelvjárás felismerő képessége, így lekérdezhető, hogy a komponensbe töltött ontológia megfelel-e a következtetőgép képességeinek.

A komponensbe könnyen beilleszthető, mivel szintén Java nyelven íródott, így az OWL API függvények közvetlenül meghívhatóak.

6.1.3. RACER

A következtetőgéppel szemben támasztott követelmények a platformfüggetlenségen túl az elegendő teljesítmény nyújtása, az ontológia kifejezőerejének támogatása és a Java nyelvű komponenssel való integrálhatóság.

A platformfüggetlenség csak részlegesen teljesül: a RACER kutatási célokra ingyenesen hozzáférhető bináris változatai a Windows, Linux és Mac OS X operációs rendszereket támogatják. Azonban a TCP interfész miatt távolról is elérhető, nem szükséges az IKF ontológia modullal egy gépen futnia. Ez a komponens használhatóságát is növeli, mivel a felhasználói felület válaszidejét nem növeli meg a következtetőgép nagy terhelése.

A RACER következtető által támogatott leírólogika kifejezőerejét már bemutattam. A Java nyelvvel való integrálhatóságra pedig két lehetőséget is biztosít. A DIG interfészen keresztül elérhető az OWL API, vagy a RACER készítői által nyújtott *DIGRacerClient* segítségével. A gyorsabb és komplexebb lekérdezéseket is támogató TCP interfészhez szintén biztosítanak Java nyelvű API-t, *JRacer* néven.

6.1.4. SOAP

Az ontológiát kezelő komponens integrálhatóságának lényegi kérdése a klienseknek biztosított szolgáltatások elérésének módja. A szolgáltatásprimitív-készlet a leíró logikai következtetőgép szolgáltatásait közvetíti, figyelembe véve a tárolt moduláris ontológia lekérdezhetőségét, ahogy azt az ontológia API ismertetésénél, az 5.4. fejezetben leírtam.

A szolgáltatások technológiai megvalósításakor az elosztott keretrendszeren belüli

heterogén komponensek kiszolgálására, a szolgáltatások egyszerű elérhetőségére és a megfelelő teljesítmény elérésére kell figyelmet fordítani.

Az egyszerű elérhetőség érdekében távoli függvényhívásokkal célszerű megvalósítani a szolgáltatásokat saját protokoll definiálása helyett, ami a kliens készítésekor jóval kevesebb munkát ró a fejlesztőre.

A Java saját könyvtára a Java RMI[44] csak Java virtuális gépek közötti távoli függvényhívást tesz lehetővé, heterogén környezetben nem alkalmazható. A DCOM,[45] a Microsoft ajánlása hasonlóan korlátozottan használható. Az OMG[46] által specifikált CORBA[47] már független a felhasznált implementációs környezettől, az interfész definíciók (IDL) segítségével számtalan programozási nyelvet használhatunk.

Azonban létezik a CORBA-nál egyszerűbb és újabb megvalósítás is, a SOAP,[48] mely 2003 júniusa óta a W3C ajánlása. HTTP protokollon keresztül teszi lehetővé távoli függvények elérését, miközben a kéréseket és válaszokat XML dokumentumba csomagolja. Egy SOAP szerver egyben HTTP szerver, valójában Webes szolgáltatás (*Web service*).

A Web szolgáltatások a CORBA továbbfejlesztésének is tekinthetőek, melyek az objektum-orientált, szkeletonokon és csonkokon alapuló, bonyolult CORBA elérést a Weben már jól bevált, átlátható és közismert technológiákkal helyettesítik, és teljes mértékben platform és nyelvfüggetlenek [Clements02].

A SOAP 1.1 specifikáció szerint a SOAP egy egyszerű protokoll, mely decentralizált, elosztott környezetben teszi lehetővé az információcserét. Több szabadon hozzáférhető implementációja létezik, a Sun hivatalos megoldása a Java WSDP,[49] mely Tomcat Web szerverbe telepíthető SOAP szervert tartalmaz. Én az XSOAP[50] nevű megoldást használtam, így nem kellett külön Web szervert telepítenem, az alkalmazás magában foglalja a HTTP szervert is.

6.2. Ontológiák kezelése

A moduláris ontológiák függőségeinek kézben tartásához, az egyes modulok állapotának követéséhez a tervezéskor bevezettem a komponens konfigurációját (5.2. fejezet, 11. ábra).

[44] Java Remote Method Invocation, `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/index.html`

[45] Distributed Component Object Model, `http://www.microsoft.com/com/tech/DCOM.asp`

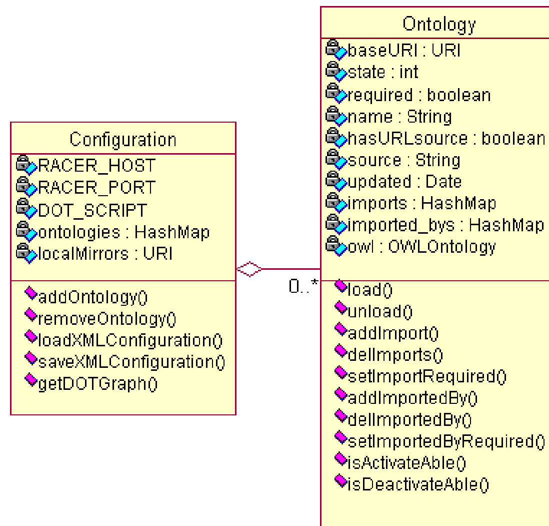
[46] Object Management Group, `http://www.omg.org/`

[47] Common Object Request Broker Architecture, `http://www.omg.org/technology/documents/formal/corba_iiop.htm`

[48] Simple Object Access Protocol, `http://www.w3.org/TR/soap/`

[49] Java Web Services Developer Pack, `http://java.sun.com/webservices/jwsdp/index.jsp`

[50] XSOAP toolkit, régi nevén SoapRMI, `http://www.extreme.indiana.edu/xgws/xsoap/`



12. ábra. A modul konfiguráció (*config* csomag) UML diagramja

Független belső egységet alkot, biztosítva az adatok konzisztenciáját. Az implementáció során külön Java csomagba helyeztem, melynek egyik osztálya egyetlen ontológia, másik osztálya a teljes komponens konfigurációját tartja kézben. Az osztályok fontosabb attribútumait és tagfüggvényeit mutatja a 12. ábrán látható UML diagram.

A teljes konfiguráció XML konfigurációs állományba menthető (*loadXMLConfiguration*) és onnan visszatölthető (*saveXMLConfiguration*). A konfigurációhoz hozzáfűzhetőek az újonnan létrehozott ontológiák (*addOntology*), melyek a következő adatokat tartalmazzák:

baseURI az ontológiát azonosító URI;
state állapot, mely lehet még nem betöltött, betöltött és aktivált;
name neve, mely a felhasználói felületen megjelenik;
hasURLsource, source forrása (file vagy URL), illetve annak címe;
updated utolsó frissítés dátuma;
imports, imported_bys függőségei és tőle függő ontológiák;
owl az ontológia tartalma az OWL API adatszerkezetében.

A függőségek kezelésénél felmerült probléma a következő: mivel ontológia importálásakor az OWL specifikáció elvárja az ontológiát értelmező eszköztől, hogy a hivatkozott ontológia tartalmát az eredeti részének tekintse, minden eszköz ilyenkor automatikusan megpróbálja letölteni azt az *owl:imports* direktívában szereplő URL alapján. Az ontológiát kezelő komponens feladatai között viszont ezen függőségek manuális kezelése szerepel. Például ha valamelyik függőség nem elérhető a Weben, akkor arról ne a tudásbázist böngészve szembesüljünk, hanem a modul feltöltésekor kapjunk hibajelzést.

Az OWL API és a RACER is automatikusan betölti az importált ontológiákat. Ezért az OWL API ontológia betöltő tagfüggvényeit kiegészítettem olyan függvényekkel, melyeknek megadható, kell-e alkalmazniuk az automatikus függőségkezelést (*org.semanticweb.owl.io.owl_rdf* csomag *OWLRDFParser* osztálya, *parseOntology* függvényhívások). A függőségek így nem teljes tartalmukkal, hanem csak hivatkozásként kerülnek az ontológiába.

A perzisztens ontológia-tárba már nem az eredeti ontológia-dokumentumok kerülnek, hanem az OWL API-val képzett, *owl:imports* állításoktól mentes forrás. A RACER következtetőgéphez innen jutnak el a dokumentumok, ezáltal nem fogja automatikusan letölteni a függőségeket.

6.3. Grafikus felhasználói felület

A feladat specifikációjában a grafikus kezelőfelületre vonatkozóan csak annyi megkötést tettem, hogy a modulok közti függőségeket irányított gráf formájában célszerű megjeleníteni. Természetesen a felületnek lehetőséget kell biztosítania az ontológiákhoz kapcsolódó funkciók ellátására (betöltés, aktiválás, frissítés, eldobás) és az ontológiák adatainak megjelenítésére. A komponens teljesítményének mérését megkönnyítendő egy további grafikont is megjelenítettem, mely a másodpercenként kiszolgált kérések számát mutatja az idő függvényében.

A felületet a felsorolt funkcióknak megfelelően három panelre osztottam: ontológiák kezelése, függőségek megjelenítése és az API szerver, mely a teljesítményen túl lehetőséget biztosít a SOAP szerver elindítására és leállítására is. A panelek elrendezését, a rajtuk megtalálható tartalmat és kezelőszerveket mutatja be a B. függelék.

A függőségek megjelenítéséhez az ontológiák viszonyát megjelenítő irányított gráf ábrázolására van szükség. A gráf síkba rajzolását és megjelenítését egy külső könyvtár, a GraphViz[51] segítségével oldottam meg. Az eredetileg C nyelven íródott könyvtárat csak részben fordították le Java nyelvre (Grappa[52] néven), ezért a síkba rajzolást bináris program hívásával, illetve ha az nem elérhető, egy Web szolgáltatás használatával oldottam meg. A függőségi gráfot a GraphViz belső, DOT nevű gráf ábrázoló nyelvén a konfigurációt tároló osztály *getDOTGraph* metódusa generálja.

A testreszabhatóságot segíti, hogy felhasználtam a Java nyelvfüggetlen programok írását lehetővé tevő (*internationalization* és *localization*) szolgáltatásait, így a felület valamennyi felirata nyelvenként külön állományban foglal helyet. A magyar és az angol címkéket készítettem el, de további nyelvekre könnyen lefordítható a program.

[51] `http://www.research.att.com/sw/tools/graphviz/`

[52] Grappa, Java **Grap**h **Pa**ckage, `http://www.research.att.com/~john/Grappa/`

6.4. RACER kezelése

A betöltött ontológiák aktiválásukkal kerülnek a következtetőgépbe, a 4.4.2. fejezetben bemutatott RACER-be. A *Racer Server*-t az egyszerűbb és gyorsabb DIG interfészen keresztül vezérli a komponens a *JRacer* Java nyelvű könyvtár segítségével. A *JRacer* csupán egy egyszerű TCP socket-kezelésből és a RACER parancsainak Java függvényekbe csomagolt változataiból áll. Szerencsére szabad forráskódú és könnyen áttekinthető, mivel hiányosságait és hibáit is kellett javítanom alkalmazása közben – egyes RACER parancsok hiányoztak, mások szintaktikai hibával kerültek a könyvtárba.

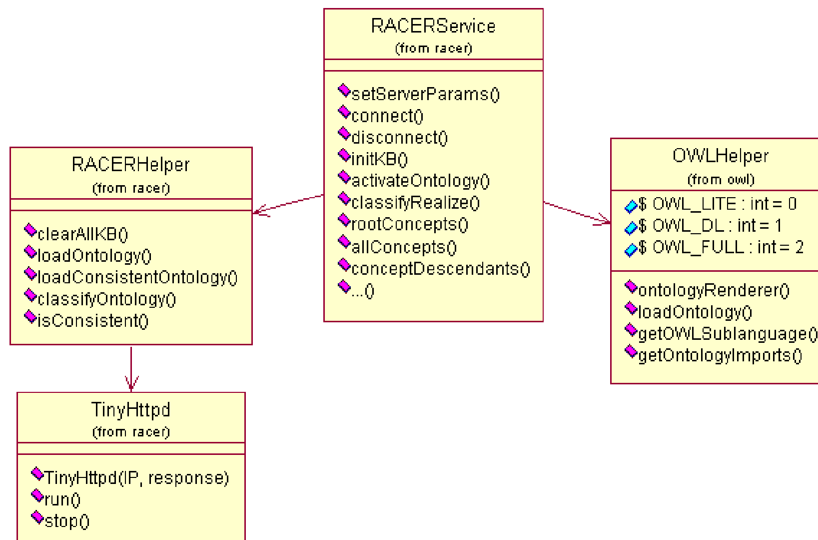
Ontológia aktiválása. A tudásbázis inicializálása, a klasszifikáció elvégzése és a lekérdezések mind-mind egyszerű JRacer függvényhívásokon keresztül valósulnak meg, melynek paraméterei a tudásbázist vagy az entitásokat azonosító karakterfüzérek. Az OWL ontológia aktiválására, azaz következtetőgépbe töltésére szintén tartalmaz a RACER beépített parancsokat, azonban a dokumentumot mérete miatt nem lehet paraméteren keresztül átadni [RACER referencia]. Az ontológia betölthető lokális file-ból (*owl-read-file*) és URL-lel megadva, HTTP protokollon keresztül (*owl-read-document*).

Az ontológiát lokális file-ként ebben az esetben nem lehet átadni, mivel a *Racer Server* akár távoli gépen is futhat. A betöltést ezért egy mini HTTP szerver implementálásával oldottam meg, mely az ontológiát szolgáltatja. Az egyszerűség kedvéért mindig csak az éppen betöltendő dokumentum érhető el egy fix útvonalon: a HTTP szervert működtető szál a RACER kiszolgálása előtt foglal le egy portot és az ontológia betöltése után meg is szűnik. A RACER-nek küldött parancs formátuma például:

```
(owl-read-document |http://152.66.253.32:2145/ontology|)
```

ahol `152.66.253.32` az ontológia komponenst futtató gép címe, `2145` a lefoglalt port száma. A jogosulatlan hozzáférések elkerülése végett csak a RACER-t futtató géptől fogad el kéréseket, de a kapcsolat felépülésének feltétele, hogy a *Racer Server* elérje a hálózaton az ontológia komponenst.

Konzisztencia biztosítása. A tudásbázis konzisztenciáját minden újabb ontológia modul aktiválásakor ellenőrizni kell, mivel inkonzisztens tudásbázison nincs értelme lekérdezéseket futtatni. Ezért a komponens automatikusan klasszifikáltatja a tudásbázist, majd inkonzisztencia esetén az aktiválást visszavonja.



13. ábra. Az OWL és RACER-kezelés (*owl, racer* csomagok) UML diagramja

Az aktiválás visszavonásához a tudásbázist újra fel kellene építeni, ami több ontológia modul feltöltését majd újbóli klasszifikálást jelentheti. Helyette a komponens aktiválás előtt a *Racer Server*-t a *clone-tbox* és *clone-abox* parancsokkal a tudásbázis lemásolására utasítja, mely a már klasszifikált tudásbázis teljes adatstruktúráját lemásolja. Inkonzisztencia esetén ehhez a tudásbázishoz kell visszatérni, nincs szükség újbóli építkezésre.

Ontológia modulok következtetőgépből kivonása (deaktiválása) viszont már megköveteli a tudásbázis újraépítését, mivel új konfigurációba is kerülhet a komponens, a modulok új részhalmaza lehet aktív állapotban.

Aktiváló egység felépítése. A *Racer Server*-t működtető, ontológia aktiválását végző egység osztályait tartalmazó UML diagram látható a 13. ábrán. A *RACERService* tartalmazza az összes, RACER-rel kapcsolatos tevékenységet, így a lekérdezéseket is (pl. *allConcepts*). A *RACERHelper* osztály néhány összetett funkciót valósít meg, mint például ontológia betöltése konzisztencia ellenőrzés nélkül (*loadOntology*) és a konzisztencia biztosításával (*loadConsistentOntology*). A részletezett HTTP szervert a *TinyHttpd* osztály valósítja meg.

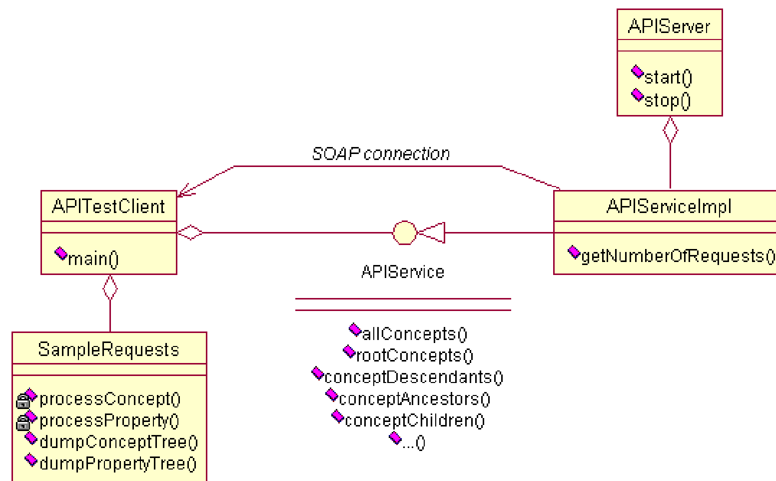
Az OWL API funkcióira épülő tevékenységeket az *OWLHelper* tartalmazza. Ide tartozik az ontológia függőségek lekérdezését végző *getOntologyImports* metódus, az ontológiát betöltő *loadOntology* függvény, a nyelvjárást megadó *getOWLSublanguage*, illetve az ontológia aktiválásakor a RACER-nek szolgáltatott ontológia forrást előállító *ontologyRenderer* függvény.

6.5. API szerver

Az ontológia lekérdezhetőségét biztosító programozói felületet a SOAP protokollon keresztül lehet elérni. A SOAP szerver megvalósítására két lehetőség kínálkozik. A Java Web szerver fejlesztői csomagjához (Java WSDP) és az *Apache SOAP*[53] vagy *Axis*[54] projektjéhez hasonló eszközök feltételezik egy Web szerver vagy alkalmazás szerver jelenlétét, amibe a SOAP szolgáltatások telepíthetőek. A megoldás a JSP-hez[55] hasonló: a Web szerverhez befutó kérés kiszolgálásakor hívódik meg a telepített alkalmazás.

A másik lehetőség olyan SOAP könyvtár használata, amely tartalmazza a HTTP kérések kiszolgálásához szükséges szervert. A jelen implementáció esetében az utóbbi megoldást részesítettem előnyben, mivel sokkal egyszerűbbé teszi az alkalmazás telepítését. Ráadásul az alkalmazás szerver tengernyi további szolgáltatását nem használnánk, fölöslegesen foglalna erőforrásokat.

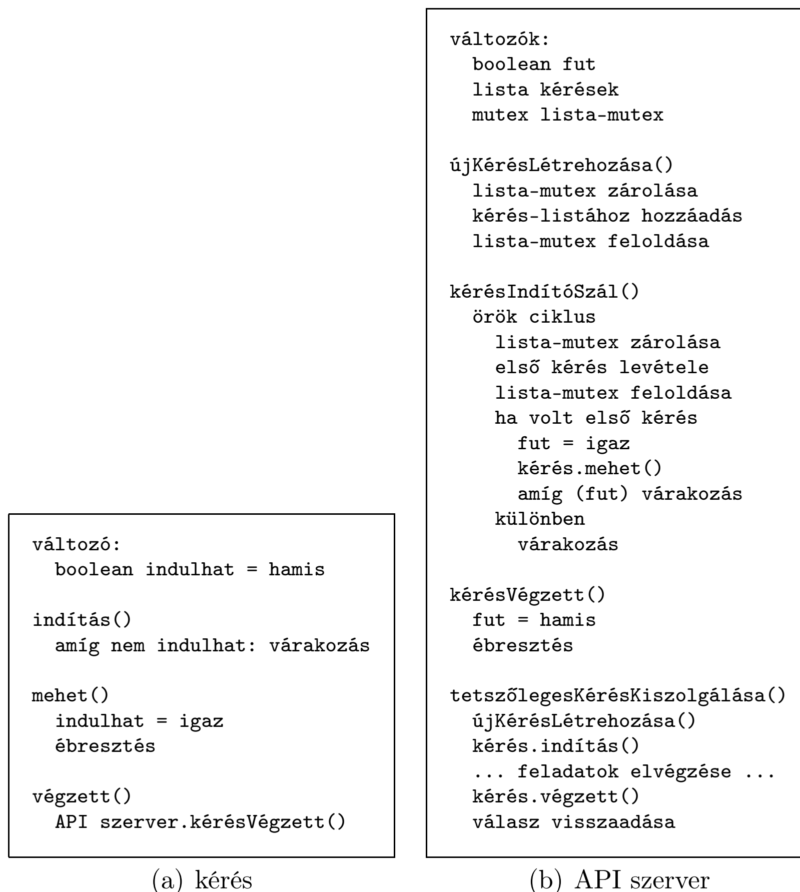
Végül az *XSOAP Toolkit*[56] nevű eszközt választottam, mely egy kisebb SOAP könyvtár. Biztosan képes együttműködni a legelterjedtebb SOAP implementációkkal, mint a .NET és az Apache SOAP, tartalmaz Web szervert, és Java és C++ platformok között is támogatja a kivételkezelést.



14. ábra. Az ontológia szerver (*api* csomag) UML diagramja

Az XSOAP szerver példányosításához nem kell csonkokat és csontvázakat generálnunk, mint például a CORBA esetében, hanem csak a szolgáltatott interfészt kell definiálnunk, ami aztán egy XSOAP szerverhez köthető. Az API szervert megvalósító egység belső felépítését mutatja a 14. ábra. Az *APIService* interfész írja le

[53] `http://ws.apache.org/soap/`
[54] `http://ws.apache.org/axis/`
[55] JavaServer Pages Technology, `http://java.sun.com/products/jsp/`
[56] `http://www.extreme.indiana.edu/xgws/xsoap/`



(a) kérés (b) API szerver

15. ábra. Konkurens kérések kiszolgálásának pszeudo-kódja

a szolgáltatásokat, a SOAP-on keresztül elérhető függvényhívásokat. Az interfész implementációja az *APIServiceImpl* osztály, mely a RACER-en keresztül a lekérdezéseket végző egység segítségével szolgálja ki a kéréseket. A *getNumberOfRequests* metódus a befutott kérések számát adja meg, így ábrázolható a szerver terhelése.

A demonstrációs és tesztelési célokat szolgáló *APITestClient* az ontológia modult használó egyszerű kliens. A csatlakozáskor az *XSOAP toolkit* egy *APIService* interfészt nyújtó egyedet bocsát rendelkezésére, ahonnan egyszerűen függvényhívásokon keresztül érhető el a szolgáltatás.

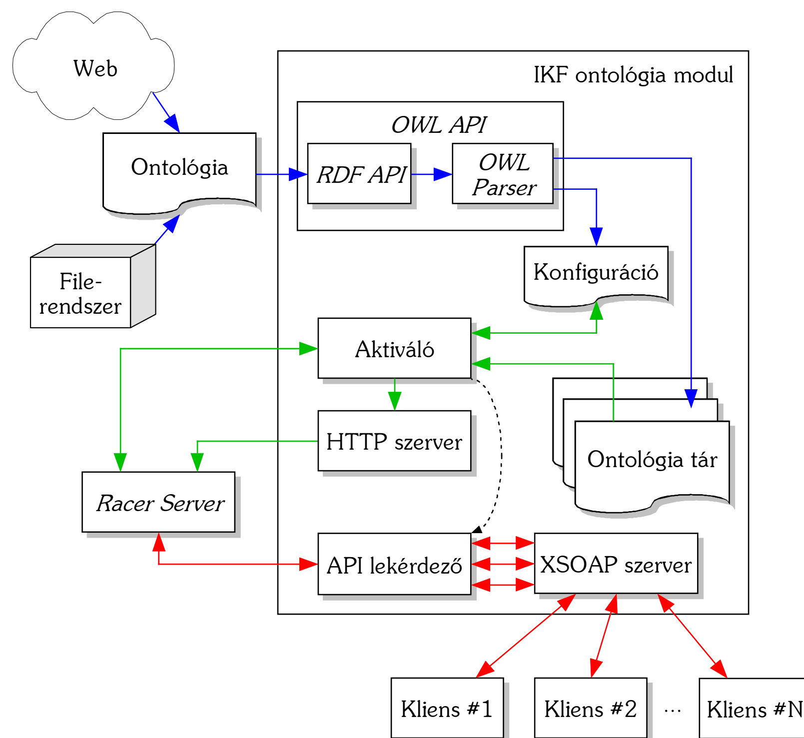
Konkurens kérések kiszolgálása. A párhuzamosan befutó kérések kezelését a tervezés fejezetben részletezett módon oldottam meg. A konkurens kéréseket kezelő függvények pszeudo-kódja a 15. ábrán látható.

A külön szálakon futó lekérdezések egy kérés-lista végére kerülnek, majd várakoznak. A lista elejéről az API szerver indításakor létrejövő szál szedi le és engedi elindulni a lekérdezések kiszolgálását (*kérésIndítóSzál*). Minden szolgáltatást végző, kliensek által hívható függvényt a *tetszőlegesKérésKiszolgálása* függvény szerint kell felépíteni. Először létre kell hozni egy *kérés* osztályt, ami felkerül a lista végére. A

kérés *indítás* függvénye megvárja a sorra kerülését. Végül a *végzett* metódus hívásával egy újabb kérés kerülhet kiszolgálásra. (A pszeudo-kódban bármely szál *várakozás* utasítása blokkolja a futást az objektumon belül bekövetkező *ébresztés* hívásig.)

A kérések az ismertetett pszeudo-kód szerinti kiszolgálása biztosítja a kérés-kiszolgálások közötti kölcsönös kizárást, amire a következtetőgép soros elérése miatt van szükség, valamint segítségével elkerülhető a kérések kiéheztetése is.

6.6. Végleges architektúra



16. ábra. Az IKF ontológia modul részletes architektúrája

A kék nyilak az ontológia betöltését, a zöld nyilak az aktiválást (következtetőbe betöltést), míg a pirosak a kérés-kiszolgálást jelzik.

Az ontológia-kezelő modul részletes felépítése a 16. ábra szerint alakul. A tervezéskor elkészített, az 5.2. fejezetben bemutatott architektúra kiegészült az implementáció során alkalmazott technológiákkal, illetve a kiszolgálásukhoz szükséges részekkel.

A beolvasott ontológia feldolgozását az OWL API segítségével végezzük, melyben az RDF API felel az RDF hármasok kinyeréséért, az OWL Parser pedig az ontológia értelmezéséért. Az API segítségével lekérdezhetők az ontológia függőségek, melyeket a konfiguráció tárol.

A következtetőgép a *Racer Server*, melybe az ontológia aktiváló egy belső *HTTP*

szerver segítségével tölti az ontológiát. A kliensektől érkező lekérdezéseket az *XSOAP szerver* fogadja, melyekre az *API lekérdező* válaszol. A konkurens, párhuzamosan több szálon érkező kéréseket a lekérdező sorosítja, egyetlen TCP kapcsolatot tartva a *Racer Serverrel.*

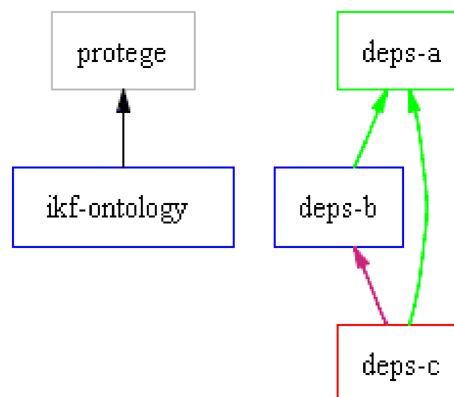
6.7. Demonstráció

Az ontológia modul teszteléséhez és képességeinek bemutatásához kísérleti ontológiákat és egy demonstrációs célú kliens programot készítettem. Segítségükkel igazolom, hogy az ontológia kezelő modul képes az alábbi feladatok elvégzésére:

- moduláris ontológia betöltésére a függőségek kezelésével;
- az ontológia következtetőgépbe töltésére, klasszifikálására;
- konkurens lekérdezések kiszolgálására.

A demonstrációs ontológiák teljes RDF/XML szintaxisú forráskódja megtalálható a C. függelékben.

Ontológia betöltése. Az ontológiák betöltése után a B. függelék 22. ábráján látható felületen tekinthetők át az ontológia modulok legfontosabb paraméterei. Itt lehet elvégezni a modulokkal kapcsolatos műveleteket (*betöltés, eldobás, aktiválás, frissítés*) is.



17. ábra. A modulok függőségi gráfja

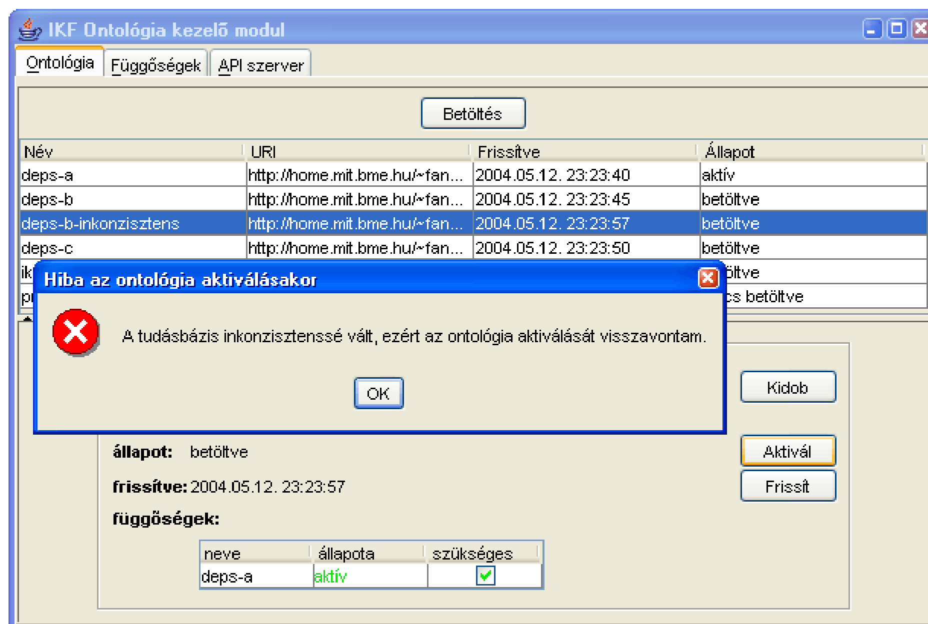
A modulok függőségeinek szemléltetéséhez készítettem a *deps-a*, *deps-b* és *deps-c* ontológiákat. A függőségi gráfon ezek viszonya látható (17. ábra, az egész panel a 23. ábrán), valamint az IKF keretrendszer jelenlegi kísérleti ontológiája (*ikf-ontology*) és az általa használt *protege* ontológia.

A függőségi viszonyokat mutató gráfon a színek az állapotokra utalnak. A zöld az aktivált állapotot jelzi, a *deps-a*-ba mutató élek zöld színe a függőség teljesülésére

utal. A kék téglalap aktiválható modult jelöl (*deps-b*) míg a piros olyan modult, melynek valamelyik függősége még nem teljesült. A *protege* ontológia még betöltve sincsen (szürke szín), de az IKF ontológia aktiválható, mivel a köztük lévő függőség ki van kapcsolva (fekete nyíl).

Klasszifikálás. Az ontológiák aktiválásukkal töltődnek be a következtetőgépbe. Az aktiválást végző gomb csak akkor érhető el, ha teljesülnek a modul függőségei. Az ontológia klasszifikálása minden modul következtetőgépbe töltése után megtörténik.

Ha inkonzisztencia lép fel, arról figyelmeztetést kapunk, ilyenkor az aktiválásnak nincs hatása, azaz visszavonódik. Az inkonzisztencia-ellenőrzés teszteléséhez hoztam létre a *deps-b-inkonzisztens* ontológiát.



18. ábra. Figyelmeztetés inkonzisztens ontológia esetén

Tartalmát tekintve „*barack*” és „*nembarack*” diszjunkt osztályokat tartalmaz, melyek metszeteként definiáltam az „*inkonzisztens_barack*” osztályt. Utóbbi természetesen nem kielégíthető (biztosan nincs példánya), így az ontológia inkonzisztens. Aktiválására a komponens megfelelően reagál, amint az a 18. ábrán látható. (Az inkonzisztens ontológia forrása megtalálható a C. függelékben.)

A klasszifikálás hatásának szemléltetéséhez betöltöttem a *deps-a* és *deps-b* modulokat a következtetőgépbe, majd megjelenítettem a tudásbázis fogalmi hierarchiáját a teszteléshez készített kliens programmal.

Az *APITestClient* az ontológia kezelő komponens lekérdezésével bejárja a fogalmak hierarchiáját. Először lekérdezi a legáltalánosabb fogalmakat (*getRootConcepts*), majd rekurzívan minden fogalom közvetlen leszármazottait (*getConceptChild-*

```
Fogalmak:
deps-a:gyumolcs
  deps-b:sargas_gyumolcs
    deps-b:sargabarack
      egyede: deps-b:kajszi
      egyede: deps-b:rozsabarack
  deps-b:barack
    deps-b:sargabarack
      egyede: deps-b:kajszi
      egyede: deps-b:rozsabarack
    deps-b:oszibarack
      egyede: deps-b:nektarin
  deps-b:korte
    egyede: deps-b:arabitka
    egyede: deps-b:vilmos
[...]
```

```
[...]
    egyede: deps-b:conference
  deps-a:alma
    egyede: deps-b:golden
    egyede: deps-a:gala
    egyede: deps-a:jonathan
    egyede: deps-a:gloster
    egyede: deps-a:mutsu
    egyede: deps-a:jonagold
    egyede: deps-a:idared
    egyede: deps-a:starking
deps-b:szin
  deps-b:sargas_szin
    egyede: deps-b:sarga
deps-a:ontologia_deps-a
deps-b:ontologia_deps-b
```

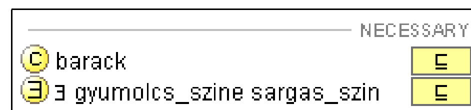
19. ábra. Klasszifikált ontológia a teszt kliens kimenetén

A valódi kimeneten az entitások teljes URI-ja olvasható. Az áttekinthetőség kedvéért azonban az ontológia modulokat *deps-a*-val és *deps-b*-vel jelöltem, és az XML névterek használatából eredő jelölést alkalmaztam.

ren). A több szülővel rendelkező fogalmak tehát több helyen is szerepelnek. A fogalmak azon egyedeit is felsorolja, melyeket nem tartalmazza egy szűkebb fogalom. Más szóhasználattal a felsorolt egyedeknek a *típusa* az aktuális fogalom.

A kliens kimenetét a 19. ábra mutatja. Ha összevetjük az ontológiák forrásával (C. függelék), feltűnik, hogy a következtetőgép levezette: a sárgabarack egy sárgás gyümölcs. Ehhez a következő információkra volt szüksége:

– a barack gyümölcs;
– a sárgabarack barack és sárgás színű;
– pontosan azok a sárgás gyümölcsök, melyek sárgás színűek és gyümölcsök.



(a) sárgabarack

(b) sárgás gyümölcs

20. ábra. A sárgabarack és a sárgás gyümölcs a Protégé OWL Plugin jelöléseivel

Az utóbbi két állítás a *Protégé* ontológia-szerkesztő jelöléseivel látható a 20. ábrán. A klasszifikáció által tehát az ontológia implicit tartalma is lekérdezhetővé vált.

6.8. Teljesítményelemzés

A lekérdezések kiszolgálásának sebessége lényeges szempont a komponens alkalmazhatóságának megítélésében. A modul teljesítményét ezért a tesztelő kliens és a komponens API szerver terhelését megjelenítő panele segítségével végeztem el.

A tesztelés során a kliens alkalmazást folyamatosan, több példányban távoli gépekről futtattam annak érdekében, hogy az ontológia modul maximálisan kihasználhassa a szerver gép erőforrásait. A lekérdezéseket párhuzamosan végző kliensek számának nem volt számottevő befolyásoló hatása, az eredmények a szerver processzorának maximális terhelésére vonatkoznak.

Az ontológia modult futtató számítógép 2,4 Ghz-es Pentium IV processzorral, 512MB memóriával és Windows XP Professional operációs rendszerrel rendelkezett. A szervert az 1.4.2_03-as verziójú Java SDK futtatta. A memóriából a *Racer Server* 35MB-ot, az ontológia modul 25MB-ot foglalt le.

Ontológia	*fogalmak*	*tulajdonságok*	*példányok*
ikf-ontology.owl	150	16	5
Geography.owl	432	212	654

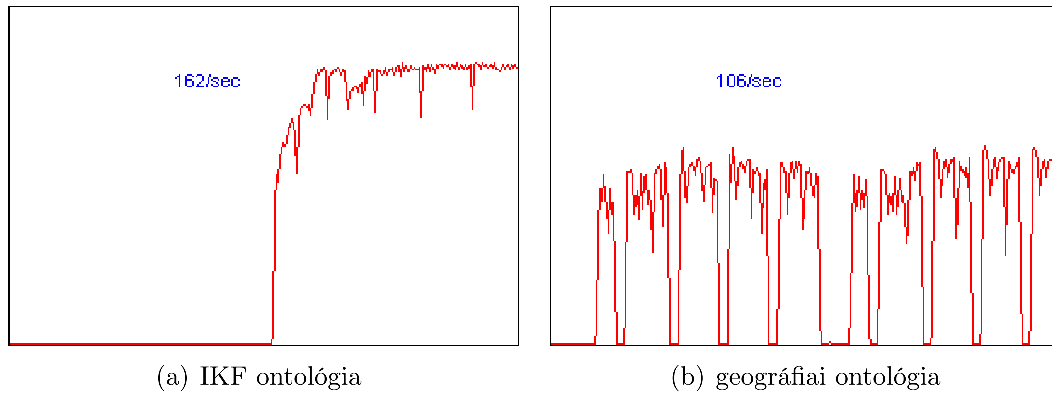
3. táblázat. A teljesítményelemzéshez használt ontológiák mérete

A méréseket két nagy méretű ontológiával végeztem el, egyik az IKF projekt keretében készülő kísérleti ontológia[57] (*ikf-ontology.owl*), másik a DAML Ontology Library-ból[58] származó földrajzi ontológia[59] (*Geography.owl*). Utóbbiban egy apró szintaktikai hibát ki kellett javítanom, hogy szabványos OWL dokumentummá váljon. Az ontológiák méretét a 3. táblázat mutatja.

A mérési eredmények alapján az ontológia modul sebessége 100–200 kérés/másodperc közé tehető (21. ábra). A sebesség természetesen függ a lekérdezések eredményének nagyságától. Az IKF ontológia esetén a fogalmi hierarchia elágazási tényezője kiegyenlített, példányok alig vannak, ezért a sebesség viszonylag állandó. A földrajzi ontológia már sokkal változatosabb ilyen tekintetben. Megfigyelhető például, hogy a kliens minden ciklusában az egyik lekérdezés 3–4 másodpercig is eltart, közben teljesen blokkolva a többi lekérdezést.

A sebesség grafikonokról leolvasható további jellegzetesség, hogy a lekérdezések első ciklusában a modul még lassabban teljesít, majd utána gyorsabb, egyenletes szintre áll be. Mivel az éledés mindig az első ciklus végéig tart, a jelenség oka a *Racer*

[57] `http://aisrv.mit.bme.hu/~pvarga/ikf-ontology/kezdeti/ikf-ontology.owl`
[58] `http://www.daml.org/ontologies/`
[59] Geography ontology, `http://reliant.teknowledge.com/DAML/Geography.owl`



(a) IKF ontológia (b) geográfiai ontológia

21. ábra. Teljesítmény mérése

Server működésében keresendő. A tudásbázisban a klasszifikáció közben nem épül fel az összes struktúra, melyeket a lekérdezések közben ki kell számolni. Később, a másodszori lekérdezéseknél erre már nem kell időt fordítania, így a kiszolgálás gyorsabb.

7. Továbbfejlesztési lehetőségek

Az alkalmazást számos különböző aspektusa mentén lehet továbbfejleszteni, ezek közül említeném meg néhány kínálkozó lehetőséget.

Hibakezelés javítása. A modul kialakításánál a kívánt funkcionalitás megvalósításán illetve a megfelelő ergonómia elérésén volt a hangsúly. A továbbfejlesztés lényegi pontja a használhatóságot nagy mértékben növelő részletes hibakezelés és hibatűrés megvalósítása. A hibákat a jelenlegi implementáció is kezeli, azonban néhány esetben a figyelmeztető ablakok feldobása helyett az alkalmazás megpróbálhatná elhárítani őket.

A hibatűrés kapcsán elsősorban a *Racer Serverrel* fenntartott kapcsolatra gondolhatunk, ott alkalmazhatók hibafelismerő és kezelő technikák. Például a kapcsolat megszakadása esetén azt automatikusan újra felépítjük, és megvizsgáljuk, rendelkezésre áll-e továbbra is a tudásbázis. Ha nem, újra generáljuk. Ha a szerver sem elérhető, vagy a klasszifikáció várhatóan sokáig fog tartani, csak akkor jelzünk hibát.

Verziókövetés. Nagyobb, főleg több fejlesztőt is foglalkoztató projektek esetén elengedhetetlen valamilyen verziókövető rendszer használata. Segít összehangolni a résztvevők munkáját, és a komponensek időbeli fejlődése is nyomon követhetővé válik.

Moduláris ontológiák fejlesztésekor az ontológia részeinek kompatibilitása önmagában felveti a verziókövetés szükségességét. Az OWL ontológia nyelv segítséget nyújt a verziók és tulajdonságaik jelöléséhez, a következő állításokat vezették be [OWL referencia]:

owl:priorVersion az ontológia előző verziójára hivatkozik;

owl:backwardCompatibleWith az új ontológia kompatibilis a hivatkozott régivel, tehát a régi ontológia minden entitásának azonos jelentést tulajdonít;

owl:incompatibleWith az előzővel ellentétben nem szabad áttérni az új verzióra, mivel elképzelhető, hogy a változtatásokhoz alkalmazkodni kell.

Az OWL referenciára tehát lehet támaszkodni az ontológiát kezelő komponens verziókövetésének elkészítésénél. A különböző verziójú modulok viszonyának leírására, például annak ábrázolására, hogy egy függőség a beillesztendő ontológia milyen verzióival működik együtt, bevezethetünk az IKF-en belül használandó jelöléseket.

A fentiek után természetesen adódik annak támogatása, hogy az ontológiákat közvetlenül a verziókövető rendszerből (pl. *CVS repository*-ból) tudjuk beolvasni.

Részletes statisztikák a kérés-kiszolgálásról, válaszidő kézben tartása. A kérések sorba állításával könnyen mérhetővé válik számos hasznos statisztikai adat, például a kérések minimális, átlagos, maximális kiszolgálási ideje. Amennyiben a szervert használó kliensek ezt lehetővé teszik, a túl hosszú kérések kezelésére bevezethetőek stratégiák, melyek egyes lekérdezések visszadobásával a válaszidő kézben tartását szolgálják. A visszautasított kérés kiválasztásának szempontjai lehetnek:

- egy kérés megválaszolására fordított maximális idő,
- kérés maximális kiszolgálási ideje,
- a kérések listájának maximális hossza.

A komponenst használó kliensek a legtöbb esetben számítanak a lekérdezés eredményére, ezért egy ilyen stratégia általános alkalmazása nem kivitelezhető. Ha azonban a kliensek jelezhetik, hogy feltétlenül szükségük van-e a válaszra, vagy csak a szerver szabad kapacitására tartanak igényt, a szerver ezt figyelembe veheti, így jobban ki tudja használni erőforrásait, skálázhatóbbá válik.

8. Összegzés

A dolgozat elején bemutattam a Szemantikus Web kezdeményezést és az IKF rendszerprototípus megalkotását célzó projektet. Megmutattam, hogy a két alapvetően eltérő megközelítés között valójában szoros párhuzam húzható: mindkettő a cél környezet fogalmaira építkezve igyekszik felépíteni a dokumentumok szemantikus leírását.

Az IKF keretrendszer működéséhez – csakúgy, mint a Szemantikus Web esetében – elengedhetetlen a tárgyterület fogalmainak egzakt leírása, amihez a megfelelő eszköz az ontológia. Részletesen bemutattam az ontológiához kapcsolódó, jelenleg elterjedt technológiákat. Először a fogalmi rendszer ábrázolására használt, a leírás kifejezőerejét meghatározó ontológia nyelveket mutattam be, majd kiválasztottam a projekt számára legmegfelelőbb reprezentációt. Javaslatot tettem a projekt méreteinek és céljainak megfelelő ontológia szerkesztő megválasztására. Végül az ontológiák használatát megkönnyítő programozói felületek közül választottam ki a komponens megvalósítására legalkalmasabb API-t.

A diplomatervezés során megvalósítandó komponens, az IKF belső ontológiáját beágyazó modul feladatai azonban nem csak a moduláris ontológia kezelésére és lekérdezhetőségére terjednek ki, hanem azt a lekérdezések kiszolgálása előtt automatikusan klasszifikálni kell. A klasszifikáció az ontológia állításai által implicit megfogalmazott információ kinyerését jelenti, melyet hatékonyan következtetőgép használatával lehet elérni.

Bemutattam az ontológiai következtetés fajtáit, valamint annak célját, majd rátértem a leíró logikák ismertetésére. Utóbbi a választott ontológia nyelven, az OWL-en való következtetés alapjául szolgál. A szakirodalom segítségével áttekintést adtam a leíró logikák fajtáiról, és megvizsgáltam, melyek feleltethetőek meg legjobban a számunkra érdekes ontológia nyelveknek. Végül a logikák ismeretében már könnyen kiválaszthattam az ontológia modul feladatainak legmegfelelőbb következtetőgépet.

A komponens tervezését a feladatok elemzése után a részletes feladat-specifikáció elkészítésével kezdtem. Az architektúra tervezésekor sikeresen elválasztottam egymástól az ontológiához kapcsolódó három fő feladatcsoportot – betöltést, aktiválást, kérés-kiszolgálást – ellátó egységeket, és tisztáztam azok viszonyát. Megfogalmaztam a több modulból felépülő ontológiák függőségeinek kezelésével kapcsolatos teendőket, a modulok és a függőségi viszonyok lehetséges állapotait. Végül a kérés-kiszolgálás kapcsán megterveztem a nyújtott lekérdező API-t. A konkurens kérések kezelésére FIFO rendszerű kiszolgálót használtam.

Az implementáció során először az ontológiához és következtetéshez kapcsolódó

technológiákon túl programozási nyelvet (Java) és elosztott környezetben is jól használható távoli eljáráshívási módot (SOAP) választottam. A grafikus felület elkészítésekor a függőségi gráf áttekinthető megjelenítéséhez a GraphViz könyvtárat hívtam segítségül. Az ontológia függőségek teljes kézbentarthatósága végett módosítanom kellett OWL API-t, hogy az automatikus importálást elkerüljem. Az egyes ontológia modulok aktiválását egy HTTP szerver implementálásával oldottam meg: a következtetőgép a komponenst futtató gépre mutató Web címet kap, annak segítségével tölti le az ontológiát. Természetesen minden aktiváláskor a teljes ontológia konzisztenciáját is ellenőrzi a komponens.

A komponens helyes működésének bemutatásához demonstrációs célú ontológia modulokat és egy tesztelő kliens komponenst készítettem. Segítségükkel megmutattam, hogy a modul inkonzisztencia fellépésekor automatikusan figyelmeztet, illetve a lekérdező felületen már a klasszifikált, implicit információkat is tartalmazó ontológia kérdezhető le. Annak megállapítására, hogy a komponens képes megfelelő teljesítménnyel kiszolgálni a kéréseket, két nagy méretű ontológiával végeztem méréseket. Az eredmények szerint 100–200 kérés/másodperces kiszolgálási sebességre képes a modul, mely elegendő az IKF rendszerprototípus igényeinek kielégítésére. Végül ismertettem néhány, általam legfontosabbnak tartott továbbfejlesztési lehetőséget, melyek mentén a komponens képességei tovább fokozhatóak.

Függelék

A. Szolgáltatott primitívek leírása

Az ontológia kezelő komponens szolgáltatásait, tehát az API szervernek küldhető kéréseket táblázatok formájában, tömören ismertetem. A szolgáltatásokat a lekért entitás típusa (*fogalom*, *tulajdonság* vagy *példány*) és a szolgáltatás típusa (*kiértékelés* vagy *lekérdezés*) szerint csoportosítottam.

Az első oszlopba került a függvényhívás neve, a paraméterek megadásánál a C fogalmat (*concept*), a P tulajdonságot (*property*), az I példányt (*instance*) jelöl. Utána megadtam a RACER következtetőgép TCP interfészén kiadott parancs nevét, mely a szolgáltatást megválaszolja. Végül röviden magyaráztam a hívás szemantikáját.

Fogalmak lekérdezése

Függvény	*RACER megfelelő*	*Magyarázat*
isConcept(C)	concept-p	C egy fogalom?
isConceptParent(C_p, C_c)	concept-subsumes-p	C_p szülője-e C_c-nek?
isConceptsEquivalent(C_1, C_2)	concept-equivalent-p	C_1 és C_2 ekvivalens?
isConceptsDisjoint(C_1, C_2)	concept-disjoint-p	C_1 és C_2 diszjunkt (kizáró)?

4. táblázat. Fogalmak lekérdezése, igaz-hamis értékű függvények

Függvény	*RACER megfelelő*	*Magyarázat*
getAllConcepts()	all-atomic-concepts	minden fogalom
getRootConcepts()	concept-children(TOP)	nem általánosítható fogalmak
getConceptDescendants(C)	concept-descendants	C leszármazottai
getConceptAncestors(C)	concept-ancestors	C ősei
getConceptChildren(C)	concept-children	C gyerekei (közvetlen leszármazottai)
getConceptParents(C)	concept-parents	C szülei (közvetlen ősei)
getConceptSynonyms(C)	concept-synonym	C szinonímái (ekvivalens fogalmak)
getPropertyDomain(P)	role-domain	P tulajdonság értelmezési tartománya
getPropertyRange(P)	role-range	P tulajdonság értékkészlete
getIndividualConcepts(I)	individual-types	I-t tartalmazó fogalmak
getIndividualTypes(I)	individual-direct-types	tovább nem szűkíthető, I-t tartalmazó fogalmak

5. táblázat. Fogalmak listájával visszatérő lekérdezések

Tulajdonságok lekérdezése

Függvény	*RACER megfelelő*	*Magyarázat*
isProperty(P)	role-p	P egy tulajdonság?
isPropertyParent(P_p, P_c)	role-subsumes-p	P_p szülője-e P_c-nek?
isTransitive(P)	transitive-p	P tranzitív tulajdonság?
isSymmetric(P)	symmetric-p	P szimmetrikus tulajdonság?
isReflective(P)	reflective-p	P reflexív tulajdonság?

6. táblázat. Tulajdonságok lekérdezése, igaz-hamis értékű függvények

Függvény	*RACER megfelelő*	*Magyarázat*
getAllProperties()	all-roles	minden tulajdonság
getTransitiveProperties()	all-transitive-roles	minden tranzitív tulajdonság
getPropertyDescendants(P)	role-descendants	P leszármazottai
getPropertyAncestors(P)	role-ancestors	P ősei
getPropertyChildren(P)	role-children	P gyerekei (közvetlen leszármazottai)
getPropertyParents(P)	role-parents	P szülei (közvetlen ősei)
getPropertySynonyms(P)	role-synonym	P szinonímái (ekvivalens tulajdonságok)
getPropertyInverse(P)	role-inverse	P inverze
getIndividual-PairAttributes(I_1, I_2)	retrieve-individual-filled-roles	az I_1 és I_2 közötti összes tulajdonság

7. táblázat. Tulajdonságok listájával visszatérő lekérdezések

Példányok lekérdezése

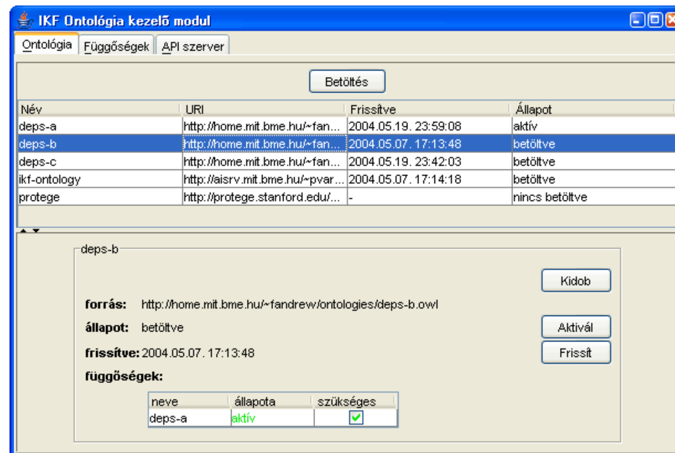
Függvény	*RACER megfelelő*	*Magyarázat*
isIndividual(I)	individual-p	I példány?
isIndividualOf(I, C)	individual-instance-p	I a C példánya?
areIndividualsRelated(I_1, I_2, P)	individuals-related-p	I_1 és I_2 össze van kapcsolva P-vel?

8. táblázat. Példányok lekérdezése, igaz-hamis értékű függvények

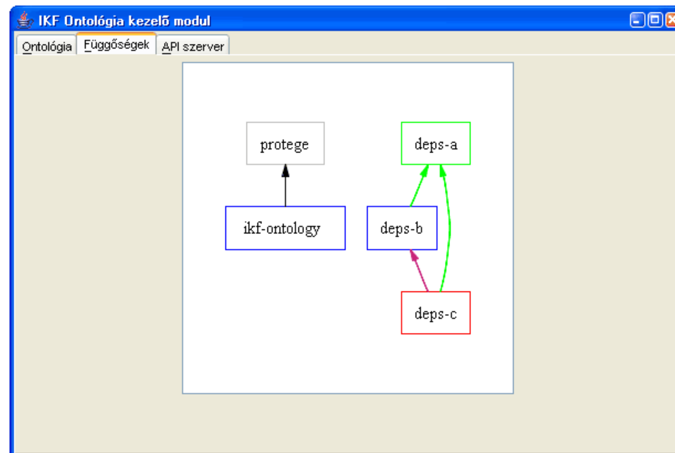
Függvény	*RACER megfelelő*	*Magyarázat*
getAllIndividuals()	all-individuals	minden példány
getConceptInstances(C)	concept-instances	C összes példánya
getAttribute(I, P)	individual-fillers	I egyed P típusú attribútumai
getAttributePairs(P)	related-individuals	P által összekapcsolt példány-párok
getReverseAttribute(I, P)	retrieve-direct-predecessors	I-be honnan mutat P?

9. táblázat. Példányok listájával visszatérő lekérdezések

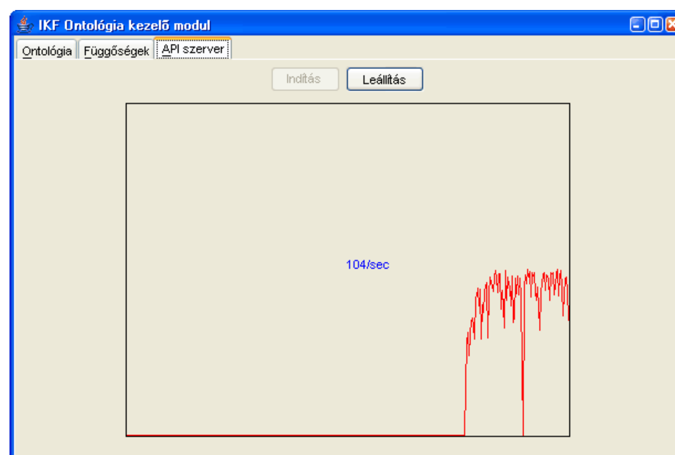
B. Grafikus felhasználói felület



22. ábra. Az ontológiák kezelését végző felhasználói panel



23. ábra. Az ontológia modulok függőségeit ábrázoló felhasználói panel



24. ábra. Az API szerver működését, terhelését jelző felhasználói panel

C. Demonstrációs ontológiák

`deps-a` ontológia

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns="http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="gyumolcs"/>
  <owl:Class rdf:ID="alma">
   <rdfs:subClassOf rdf:resource="#gyumolcs"/>
  </owl:Class>
  <alma rdf:ID="jonathan"/>
  <alma rdf:ID="jonagold"/>
  <alma rdf:ID="gloster"/>
  <alma rdf:ID="starking"/>
  <alma rdf:ID="mutsu"/>
  <alma rdf:ID="gala"/>
  <alma rdf:ID="idared"/>
  <owl:Class rdf:ID="ontologia_deps-a"/>
</rdf:RDF>
```

deps-b ontológia

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY deps-a "http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl#">
  <!ENTITY owl    "http://www.w3.org/2002/07/owl#">
  <!ENTITY xsd    "http://www.w3.org/2001/XMLSchema#">
]>

<rdf:RDF
    xmlns="http://home.mit.bme.hu/~fandrew/ontologies/deps-b.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:ontoa="http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl#"
    xml:base="http://home.mit.bme.hu/~fandrew/ontologies/deps-b.owl">
  <owl:Ontology rdf:about="">
   <owl:imports
     rdf:resource="http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl"/>
  </owl:Ontology>
  <!-- a sárga egy sárgás szín -->
  <owl:Class rdf:ID="szin" />
  <owl:Class rdf:ID="sargas_szin">
   <rdfs:subClassOf rdf:resource="#szin" />
  </owl:Class>
  <sargas_szin rdf:ID="sarga" />
  <!-- gyümölcsöknek lehet színük -->
  <owl:ObjectProperty rdf:ID="gyumolcs_szine">
    <rdfs:domain rdf:resource="&deps-a;gyumolcs" />
    <rdfs:range rdf:resource="#szin" />
  </owl:ObjectProperty>
  <!-- pontosan azok a sárgás gyümölcsök, melyek színe sárgás -->
  <owl:Class rdf:ID="sargas_gyumolcs">
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#gyumolcs_szine" />
        <owl:someValuesFrom rdf:resource="#sargas_szin" />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

```
  <!-- a barack gyümölcs és nem alma -->
  <owl:Class rdf:ID="barack">
   <rdfs:subClassOf rdf:resource="&deps-a;gyumolcs" />
    <owl:disjointWith rdf:resource="&deps-a;alma" />
  </owl:Class>
  <!-- a sárgabarack barack, mellesleg sárgás színű -->
  <owl:Class rdf:ID="sargabarack">
   <rdfs:subClassOf rdf:resource="#barack" />
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#gyumolcs_szine" />
        <owl:someValuesFrom rdf:resource="#sargas_szin" />
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <!-- az őszibarack is barack, de nem sárgabarack -->
  <owl:Class rdf:ID="oszibarack">
   <rdfs:subClassOf rdf:resource="#barack" />
    <owl:disjointWith rdf:resource="#sargabarack" />
  </owl:Class>
  <!-- a körte az almától és baracktól különböző gyümölcs -->
  <owl:Class rdf:ID="korte">
   <rdfs:subClassOf rdf:resource="&deps-a;gyumolcs" />
    <owl:disjointWith>
      <owl:Class rdf:about="&deps-a;alma" />
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#barack" />
    </owl:disjointWith>
  </owl:Class>
  <!-- példák (példányok) gyümölcsökre -->
  <ontoa:alma rdf:ID="golden" />
  <korte rdf:ID="vilmos" />
  <korte rdf:ID="arabitka" />
  <korte rdf:ID="conference" />
  <sargabarack rdf:ID="rozsabarack" />
  <sargabarack rdf:ID="kajszi" />
  <oszibarack rdf:ID="nektarin" />
  <owl:Class rdf:ID="ontologia_deps-b" />
</rdf:RDF>
```

deps-b-inkonzisztens **ontológia**

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY deps-a "http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl#">
  <!ENTITY owl    "http://www.w3.org/2002/07/owl#">
  <!ENTITY xsd    "http://www.w3.org/2001/XMLSchema#">
]>

<rdf:RDF
    xmlns="http://home.mit.bme.hu/~fandrew/ontologies/deps-b.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:ontoa="http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl#"
    xml:base=
     "http://home.mit.bme.hu/~fandrew/ontologies/deps-b-inkonzisztens.owl">
  <owl:Ontology rdf:about="">
   <owl:imports
     rdf:resource="http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl"/>
  </owl:Ontology>
  <owl:Class rdf:ID="nembarack"/>
  <!-- a barack gyümölcs, de nem alma, nem is ,,nembarack'' -->
  <owl:Class rdf:ID="barack">
   <rdfs:subClassOf rdf:resource="&deps-a;gyumolcs" />
    <owl:disjointWith>
      <owl:Class rdf:ID="nembarack"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="&deps-a;alma"/>
    </owl:disjointWith>
  </owl:Class>
  <barack rdf:ID="inkonzisztens_barack"/>
  <nembarack rdf:ID="inkonzisztens_barack"/>
  <owl:Class rdf:ID="ontologia_deps-b-inkonzisztens"/>
</rdf:RDF>
```

`deps-c` ontológia

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

<!DOCTYPE rdf:RDF [
  <!ENTITY deps-a "http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl#">
  <!ENTITY deps-b "http://home.mit.bme.hu/~fandrew/ontologies/deps-b.owl#">
  <!ENTITY owl    "http://www.w3.org/2002/07/owl#">
  <!ENTITY xsd    "http://www.w3.org/2001/XMLSchema#">
]>
<rdf:RDF
    xmlns="http://home.mit.bme.hu/~fandrew/ontologies/deps-c.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:deps-a="http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl#"
    xmlns:deps-b="http://home.mit.bme.hu/~fandrew/ontologies/deps-b.owl#"
   xml:base="http://home.mit.bme.hu/~fandrew/ontologies/deps-c.owl">
  <owl:Ontology rdf:about="">
   <owl:imports
     rdf:resource="http://home.mit.bme.hu/~fandrew/ontologies/deps-a.owl"/>
   <owl:imports
     rdf:resource="http://home.mit.bme.hu/~fandrew/ontologies/deps-b.owl"/>
  </owl:Ontology>
  <owl:Class rdf:ID="cseresznye">
    <owl:disjointWith>
      <owl:Class rdf:about="&deps-a;alma"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="&deps-b;barack"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="&deps-b;korte"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="ontologia_deps-c"/>
</rdf:RDF>
```

Irodalomjegyzék

Az irodalomjegyzékben és a dolgozatban található Internetes hivatkozásokat a dolgozat beadásakor ellenőriztem.

[Baader02] Franz Baader, Werner Nutt, „Basic Description Logics”, *In the Description Logic Handbook*, edited by F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider, Cambridge University Press, 2002., pp. 47–100.

[Baader03] Franz Baader, Ian Horrocks, Ulrike Sattler, „Description Logics as Ontology Languages for the Semantic Web”, In Dieter Hutter and Werner Stephan, Eds., Festschrift in honor of Jörg Siekmann, *Lecture Notes in Artificial Intelligence*, Springer, 2003., megjelenés alatt

[Bechhofer01a] Sean Bechhofer, Carole Goble, „Towards Annotation using DAML+OIL”, *K-CAP 2001 workshop on Knowledge Markup and Semantic Annotation*, Victoria B.C, 2001. október

[Bechhofer01b] Sean Bechhofer, Carole Goble, Ian Horrocks, „DAML+OIL is not Enough”, SWWS-1, Semantic Web working symposium, Stanford (CA), 2001. július 29–augusztus 1.

[Bechhofer03] Sean Bechhofer, Phillip Lord, Raphael Volz, „Cooking the Semantic Web with the OWL API”, In International Semantic Web Conference, ISWC 2003, *Lecture Notes in Computer Science*, vol. 2870, pp. 659–675, 2003. október

[Berners-Lee01] Tim Berners-Lee, James Hendler, Ora Lassila, „The Semantic Web”, *Scientific American*, vol. 284, no. 5, pp. 35–43, 2001. május

[Clements02] Tom Clements, „Overview of SOAP”, 2002. január, `http://java.sun.com/developer/technicalArticles/xml/webservices/`

[Denny02] Michael Denny, „Ontology Building: A Survey of Editing Tools”, 2002. november, `http://www.xml.com/pub/a/2002/11/06/ontologies.html`

[Domenico02] Pisanelli D. M., Gangemi A., Steve G., „Ontologies and Information Systems: the Marriage of the Century?”, *Proceedings of Lyee Workshop*, Paris

[Fensel02] Dieter Fensel, Frank van Harmelen, Ian Horrocks, „OIL: A Standard Proposal for the Semantic Web”, Deliverable 0 in the European IST project

[Gomez-Perez02] Asunción Gómez-Pérez, Oscar Corcho, „Ontology Specification Languages for the Semantic Web”, *IEEE Intelligent Systems and their Applications*, vol. 17, no. 1, pp. 54–60, 2002. február

[Guarino95] Nicola Guarino, Pierdaniele Giaretta, „Ontologies and Knowledge Bases: Towards a Terminological Clarification”, *Mars N.J.I. (ed.): Towards Very Large Knowledge Bases*, pp. 25–32, IOS Press, Amsterdam, 1995.

[Haarslev03a] Volker Haarslev, Ralf Möller, „Racer: An OWL Reasoning Agent for the Semantic Web”, *Proceedings of the International Workshop on Applications*, Products and Services of Web-based Support Systems, in conjunction with the 2003 IEEE/WIC International Conference on Web Intelligence, Halifax, Canada, 2003. október 13., pp. 91–95.

[Haarslev03b] Volker Haarslev, Ralf Möller, „Racer: A Core Inference Engine for the Semantic Web”, *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003)*, located at the 2nd International Semantic Web Conference ISWC 2003, Sanibel Island, Florida, USA, 2003. október 20., pp. 27-36.

[Horrocks00] I. Horrocks, U. Sattler, S. Tobies, „Practical Reasoning for Very Expressive Description Logics”, *Logic Journal of the IGPL*, vol. 8, no. 3, pp. 239–263, 2000.

[Horrocks02] I. Horrocks, „A Denotational Semantics for Standard OIL and Instance OIL”, `http://www.ontoknowledge.org/oil/downl/semantics.pdf`, 2000. november

[Horrocks03] Ian Horrocks, Peter F. Patel-Schneider, Frank van Harmelen, „From SHIQ and RDF to OWL: The Making of a Web Ontology Language”, *Journal of Web Semantics*, vol. 1, no. 1, pp. 7–26, 2003.

[Kankaanpää99] Kankaanpää Tomi, *Design and Implementation of a Conceptual Network And Ontology Editor*, diplomaterv, Helsinki University of Technology, Espoo, 1999.

[Kazakov03a] Kazakov M., Abdulrab H., „DL-workbench: a meta-modeling approach to ontology manipulation”, Evaluation of Ontology-based Tools (EON 2003), Sanibel Island, Florida, USA, 2003. október 20–23., *CEUR-WS Electronic Workshop proceedings*, vol. 87., ISSN 1613-0073

[Kazakov03b] Mikhail Kazakov levele a DL-workbench bejelentéséről „DL-workbench: a meta model based ontological platform” címmel a DL (Description Logics) levelezési listán
`https://dl.kr.org/pipermail/dl/2003/000188.html`

[Maedche03a] Alexander Maedche, Boris Motik, Ljiljana Stojanovic, Rudi Studer, Raphael Volz, „Ontologies for Enterprise Knowledge Management”, *IEEE Intelligent Systems*, vol. 18, no. 2, pp. 26–33, 2003.

[Maedche03b] Alexander Maedche, Steffen Staab, „KAON: The Karlsruhe Ontology and Semantic Web Meta Project”, *Künstliche Intelligenz*, *Special Issue on Semantic Web*, vol. 2003, no. 3, pp. 27–30.

[McBride01] Brian McBride, „Jena: Implementing the RDF Model and Syntax Specification”, *Proceedings of the Second International Workshop on the Semantic Web (SemWeb 2001)*, Hongkong, 2001. május 1.

[McGuinnes02] Deborah L. McGuinness, Richard Fikes, James Hendler, Lynn Andrea Stein, „DAML+OIL: An Ontology Language for the Semantic Web”, *IEEE Intelligent Systems*, vol. 17, no. 5, pp. 72–80, 2002.

[McIlraith01] Sheila A. McIlraith, Tran Cao Son, Honglei Zeng, „Semantic Web Services”, *IEEE Intelligent Systems and their Applications*, vol. 16, no. 2, pp. 46–53, 2001. április

[Mészáros01] Tamás Mészáros, Zsolt Barczikay, Ferenc Bodon, Tadeusz P. Dobrowiecki, György Strausz, „Building an Information and Knowledge Fusion System”, IEA/AIE-2001 The Fourteenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, 2001. június 4–7., Budapest

[Motik02] Boris Motik, Alexander Maedche, Raphael Volz, „A Conceptual Modeling Approach for Semantics-Driven Enterprise Applications”, *Proceedings of the First International Conference on Ontologies, Databases and Application of Semantics (ODBASE-2002)*, Springer, LNAI, California, USA, 2002.

[Natalya01] Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W. Fergerson, Mark A. Musen, „Creating Semantic Web Contents with Protege-2000”, *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 60–71, 2001.

[Oberle04] Daniel Oberle, Raphael Volz, Boris Motik, Steffen Staab, „An extensible ontology software environment”, *In Handbook on Ontologies*, chapter III, pp. 311–333. Steffen Staab and Rudi Studer, Eds., Springer, 2004.

[OWL referencia] Sean Bechhofer et al., „OWL Web Ontology Language Reference”, W3C Recommendation 2004. február 10., Wide Web Consortium, `http://www.w3.org/TR/owl-ref/`

[Pan01] Jeff Pan, Ian Horrocks, „Metamodeling architecture of web ontology languages”, *Proceedings of the Semantic Web Working Symposium (SWWS 2001)*, pp. 131–149, 2001. július

[RACER referencia] Volker Haarslev, Ralf Möller, „RACER User's Guide and Reference Manual”, Version 1.7.7, `http://www.sts.tu-harburg.de/~r.f.moeller/racer/download.html`

[Randall93] Randall David, Howard Shrobe, Peter Szolovits, „What is a Knowledge Representation?”, *AI Magazine*, vol. 14, no. 1, pp. 17–33, 1993.

[Russel00] Stuart J. Russel, Peter Norvig, *Mesterséges intelligencia – modern megközelítésben*, Panem – Prentice Hall, Budapest, 2000.

[Sintek02] Michael Sintek, Stefan Decker, „TRIPLE – A Query, Inference, and Transformation Language for the Semantic Web”, *International Semantic Web Conference (ISWC)*, Sardinia, 2002. június

[Sowa01] John F. Sowa, *Building, Sharing, and Merging Ontologies*, `http://www.jfsowa.com/ontology/ontoshar.htm`, 2001.

[Stevens03] Robert Stevens, Chris Wroe, Sean Bechhofer, Philip Lord, Alan Rector, Carole Goble, „Building Ontologies in DAML+OIL”, *Comparative and Functional Genomics*, vol. 4, no. 1, pp. 133–141, 2003.

[Tsarkov03] Dmitry Tsarkov, Horrocks, „DL Reasoner vs. First-Order Prover”, *Proceedings of the 2003 Description Logic Workshop (DL 2003)*, vol. 81, pp. 152–159, 2003.

[Varga03] Péter Varga, Tamás Mészáros, Csaba Dezsényi, Tadeusz P. Dobrowiecki, „An ontology-based information retrieval system”, P. W. H. CHung, C. J. Hinde, and M. Ali, editors, Developments in Applied Artificial Intelligence, 16th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expet Systems (IEA/AIE 2003), *Lecture Notes in Artificial Intelligence*, pp. 359–368, Springer Verlag, 2003.