Informatikai rendszerekkel szembeni követelmények érzékenységanalízise

TDK dolgozat

Hőnich Ágnes Kata

BSc. Mérnök Informatikus szak

Konzulens:

Dr. Pataricza András, egyetemi tanár

Budapesti Műszaki Egyetem

Méréstechnika és Információs Rendszerek Tanszék

2014

1. Absztrakt

A TDK dolgozat célja az informatikai rendszerek és az általuk vezérelt fizikai vagy üzleti folyamatok követelményrendszerei helyességének formális módszerekkel való bizonyítása.

A követelményspecifikáció, az informatikai rendszerek tervezési életciklusának első lépése döntő fontosságú a fejlesztés sikerességében. A hagyományos szöveges követelményspecifikáció nem alkalmas bonyolult rendszerek megvalósításának megalapozására, különösen, ha a rendszerrel szemben a megkívánt funkcionalitásán túlmenő extra-funkcionális (biztonsági, teljesítménybeli stb.) követelmények is vannak. Emiatt rohamosan terjed a követelményrendszer (fél-) formális modellezése és annak formális eszközökkel történő strukturálása és vizsgálata.

A korszerű modellalapú követelménytervezésben már a szöveges specifikáció strukturálását is matematikai eszközök támogatják, a modell teljességének és ellentmondás-mentességének vizsgálatára az akadémiai világ formális (pl. ontológia alapú) megoldásokat dolgozott ki.

Azonban mindmáig hiányzik az az eszköztár, amellyel fel lehetne mérni a funkcionális követelményekben vagy a környezetben bekövetkező változások hatását. Az általános mérnöki gyakorlat az ilyen problémák széles körére érzékenységanalízist használ, de az informatikában – néhány speciális terület kivételével - ez még nem elterjedt.

A dolgozat az érzékenységanalízist több kérdéskörre használja. A rendszerkövetelmények között léteznek a külvilágot korlátozó feltételezések és teljesítendő elvárások. A környezet változása miatt egyes, a tervezést megkönnyítő feltételezések érvényességüket veszthetik (pl. a rendszert a kidolgozáskor még nem ismert, új típusú támadások érhetik). Hasonlóan a rendszerrel szembeni elvárások is bővülhetnek, illetve módosulhatnak. Az ilyen változások hatásanalízise a lehetőségek nagy száma és a változások tovagyűrűző hatása miatt algoritmikus támogatást igényel.

Módszeremben az érzékenységanalízis során a követelmények szisztematikus módosításával a rendszer különböző mutációi jönnek létre, amelyek a lehetséges változások hatását reprezentálják. Ezeket vizsgálva behatárolhatóak a kritikus változások és követelmények, amelyek ismerete megalapozza a rendszer további finomításának módját és a megerősítendő pontokat.

A követelménymodell sokaspektusú vizsgálata különböző matematikai technikák kombinációit igényli. A formális módszereket megvalósító alapeszközök problémára szabásával az analízis automatikusan elvégezhető.

Az érzékenységanalízisre az Alloy modell ellenőrző köré épített keretrendszert alakítottam ki a dolgozatomban, ami megvalósítja az ipari gyakorlatban is a követelményrendszer tárolására elterjedten használt ontológiák és az Alloy fölé épített érzékenységvizsgáló alkalmazás közötti konverziót is, így lehetővé téve a rendszerkövetelményeinek mély elemzését.

Tartalomjegyzék

2. Bevezető

A rendszerkövetelmények szabatos specifikálása és a jó minőségű követelményeket garantáló folyamatok kialakítása az egész informatikai iparon keresztbenyúló probléma.

A jó minőségű követelményrendszer kialakítása ugyanis megszabja az implementációs folyamat hatékonyságát és minőségét. Ugyanakkor a nagybonyolultságú alkalmazások létrehozásának is előfeltétele, hiszen azokat csak nagyobb csapatok tudják kooperatív módon elkészíteni egy egymáshoz jól illeszkedő csapatokra bontott rész-követelményrendszer alapján.

Ezért a nagy nemzetközi szervezetek, mint IEEE-CS („*Institute of Electrical and Electronics Engineers Computer Society*”) [28] és az INCOSE („*International Council on Systems Engineering*”) ajánlásokat dolgoztak ki a terminológia és a folyamat egységesítésére megteremtve ezzel az általános célú támogató eszközpark kialakításának lehetőségét.

2.1. Rendszerkövetelmények definíciója

Az előzőekben említettek szerint a rendszerkövetelmények hatékony kialakítását és kezelését az informatika szinte minden részterülete maga elé tűzte. Az egyes részterületek kezdetben külön-külön definiálták folyamataikat, ami fogalmi-és terminológiai divergenciához vezetett az alapvetően közös probléma kezelése során. Mindemellett ezek az indokolatlan eltérések gátolták az egységes felfogást, metodika és eszközrendszer létrejöttét, így megindult az egységesítési folyamat.

Az egységes fogalomkészlet és terminológia jelenleg legérthetőbb formában a SEBoK („*Guide to the Systems Engineering Body of Knowledge*”) [30] útmutatóban szerepel, amely számos szabványt illetve ajánlást foglal magában.

Az egységesítés kiterjed a rendszertervezés olyan fő aspektusaira, mint a modellezés, a szoftverfejlesztés vagy a projektmenedzsment. Ennek előnye, hogy tömör és karbantartható összefoglalót nyújt a rendszertervezés bármely témakörében végzett tevékenységek számára. A Wikipédia szerű felépítés ugyanakkor kidolgozottság szintjében közel áll egy hierarchikus terminológiához és lehetővé teszi az „*Rendszerkövetelmények formalizálása*” és „*Analízis feladatok és eszközök*” fejezetekben részletesen ismertetendő módon a tudás formális ábrázolását például ontológia segítségével.

Az egységesített definíció szerint a rendszerkövetelmények definíciója a következő:

„A rendszerkövetelmények az összes rendszerszinten megjelenő követelményt reprezentálják. Megadják a rendszer egésze által teljesítendő funkciókat, melyek a felhasználók igényeinek és követelményeinek kielégítéséhez szükségesek. Kifejezésük a megfelelő szöveges állítások, egyes aspektusok és nem-funkcionális követelmények kombinációja, ahol a nem-funkcionális követelmények alatt az elvárt biztonsági-, megbízhatósági-, rendelkezésre állási szint megadását értjük.” (SEBoK, v1.3 290.o)

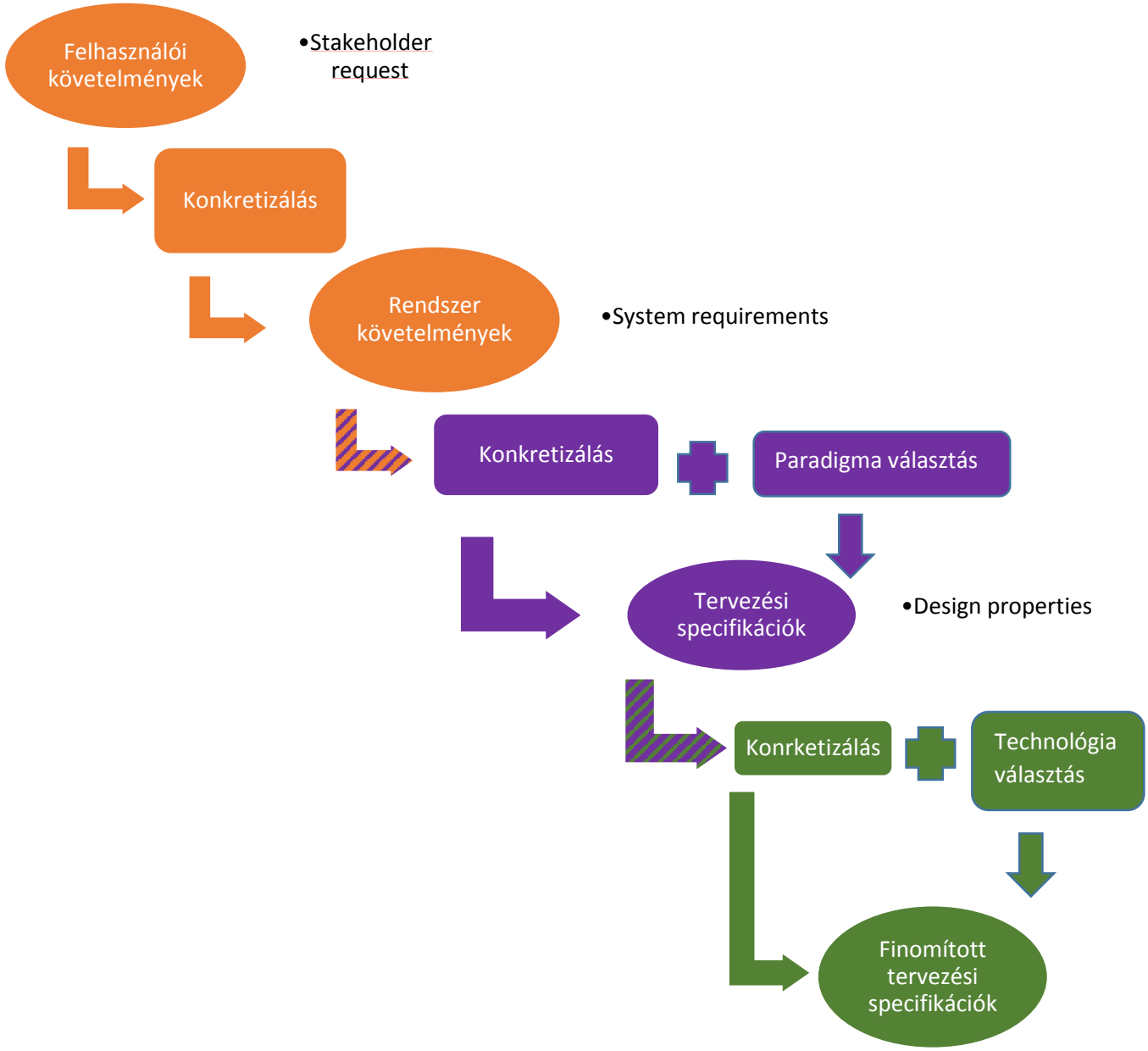
Tehát a követelmények azon képességeket és feltételeket specifikálják, melyeket a rendszernek kötelező kielégítenie: ezek vagy egy működési funkcióra vagy nem-funkcionális feltételekre vonatkoznak.

Egy rendszer akkor illeszkedik egy követelményspecifikációhoz, ha

- a működése során a követelményspecifikációban szereplő összes követelmény az összes állapotában mindig teljesül
- az előre megadott, a külvilágra vonatkozó korlátozások által definiált környezeti megkötések mellett.

2.1.1. A rendszerkövetelményektől a specifikációig

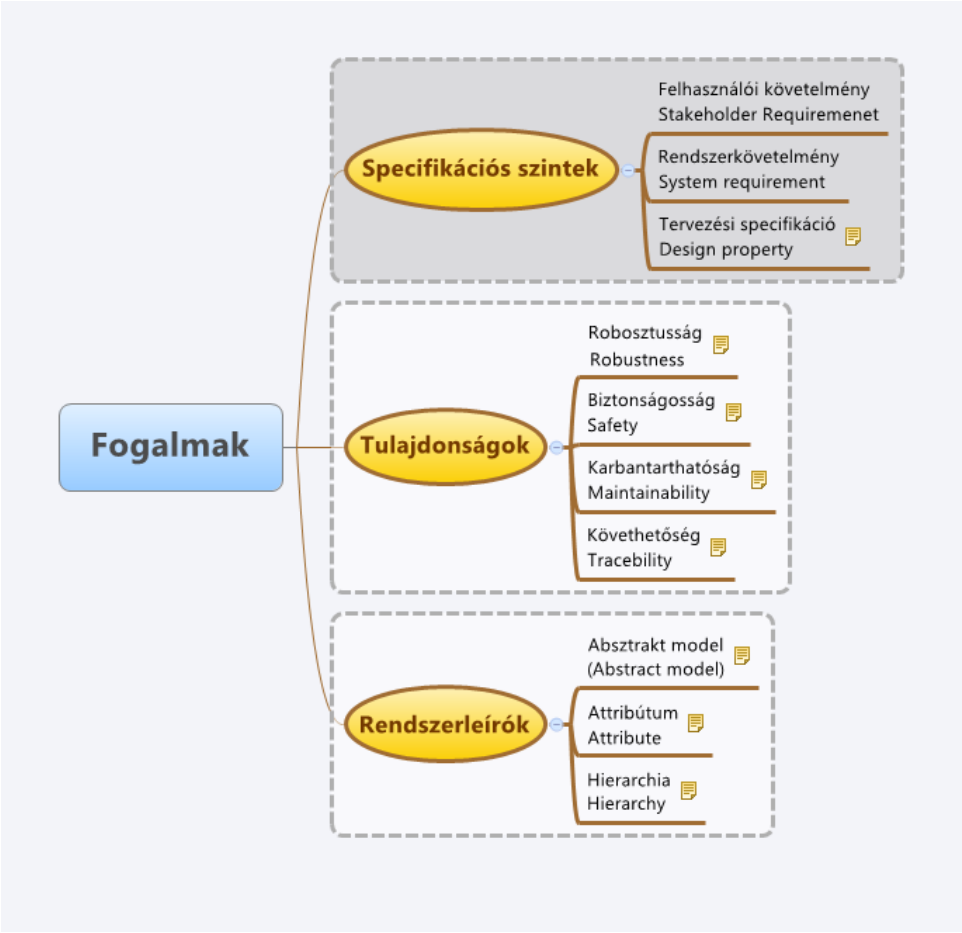
A fenti ajánlások mindegyikében megjelenik a rendszerkövetelményekből származtatott specifikáció hierarchikus strukturálása. A specifikációs folyamat során a követelmények folyamatos finomításával jut el az általánostól az egyedi követelményekig. (1. ábra)



1. ábra Specifikációs folyamat

2.1.2. A tervezési folyamat fogalomkészlete

A rendszertervezéshez kapcsolódó fő fogalmakat (2. ábra) a továbbiakban a SEBoK [31] által megadott értelemben használom.



2. ábra Fogalmak

Tulajdonságok: azok a főbb rendszerre jellemző fogalmak, amelyeket egy jó minőségű követelményrendszernek teljesíteni kell. (A gyakorlatban ennél több létezik, jelen felsorolásban a dolgozat szempontjából relevánsak szerepelnek.)

Rendszerleírók*:* A specifikációs finomítási folyamat során a rendszer specifikációjának különböző mélységű, formalizáltságú és finomságú változatai készülnek el. Leíróként az egyes szinteken elkészített főbb specifikációs elemek főbb aspektusait és komponenseit értik a szabványok.

A fogalmi struktúrát lényegesen tisztábban kezelő fél-formális modellt a *„Követelménymodellezés és eszközei”* fejezet fog ismertetni.

Specifikációs szintek: A rendszerkövetelmények finomítása során azok a pontok, amelyek önálló specifikációrendszert is eredményeznek.

A további tárgyalás szempontjából az utóbbi kategória a legfontosabb, ezért az ide tartozó fogalmak egységesített definíciója itt külön megadásra kerül, míg a többi definíció fordítása az *„A függelékben”* található.

2.1.3. Specifikációs szintek

Felhasználói követelmény („*Stakeholder requirement*”) – A felhasználóktól származó, a rendszer által teljesítendő és a külvilággal való kölcsönhatást megszabó követelmények együttese.

Rendszerkövetelmény („*System requirement*”) – A felhasználói követelmények finomítása és konkretizálása során létrejövő olyan műszaki paraméterekkel történő leírása a megkívánt szolgáltatásoknak, amelyeknek teljesülése objektív módon eldönthető.

Tervezési specifikáció („*Design property*”) – A rendszerkövetelmények kielégítésére többféle megoldás létezhet. *Tervezési specifikációknak* alatt a rendszerkövetelményeknek a kiválasztott megközelítéssel konzisztens változata értendő.

A rendszerkövetelmények finomítása során ugyanis a műszaki követelményrendszer kielégítése érdekében a tervező első lépésként valamely megvalósítási paradigmát választ. Ennek az első alapvető döntésnek a következményei alapvetően befolyásolják azt, hogy az egyes műszaki követelmények milyen környezetben és peremfeltételek mellett valósulnak meg. A tervezési specifikáció így a műszaki követelménynek egy olyan specifikus reprezentációja, amelyben már szervesen integrálódik a választott paradigmának a tervezési térre gyakorolt hatása. Hangsúlyozandó, hogy ez a lépes még nem feltétlenül tartalmazza az implementációs technika közvetlen befolyását (sőt célszerűen független a megvalósítási platformtól), azonban már nem is tisztán független a megvalósítási elvtől.

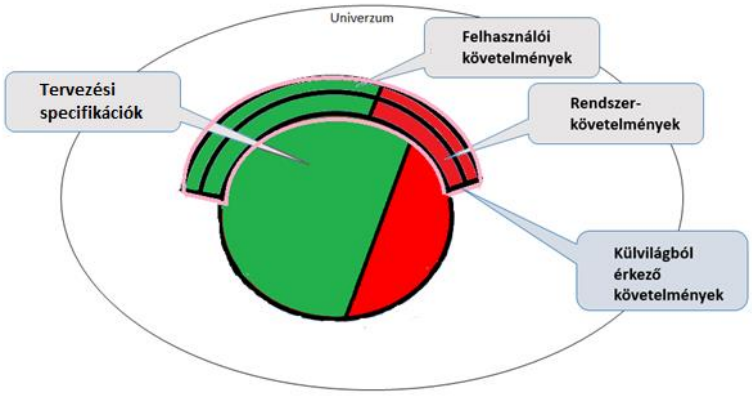
Például a követelményeknek részspecifikációra való bontása implicite magában hordozza azokat az elveket, amelyeket a jóval későbbi tervezési fázisban a kialakítandó rendszer architektúrájának finomítása során követ majd a tervező, hiszen ez teszi lehetővé azt, hogy a majdani implementáció kellő hatékonyságú legyen.

A fenti definíciókból is látszik, ami a bevezetésben már említésre került, hogy a rendszerkövetelmények kialakítása egy hierarchikus folyamat. Első lépésként a **felhasználói követelmények** azonosítása a feladat. Ilyen felhasználói követelmény például, hogy a rendszer legyen biztonságos.

Ezeket követik a hierarchiában a **rendszerkövetelmények**, melyek valamilyen műszaki specifikációja az előző szintnek. Az fenti példát folytatva ilyen rendszerkövetelmény lehet, hogy feleljen meg a SIL 4-es szintnek[1]. Ezek is megadhatnak valamilyen kötelező elvárást vagy tiltást. Ezt a két fenti szintet közösen is lehet kezelni, mint a **külvilágból érkező követelmények** halmazát. Ezt a halmazt egy másik szempont szerint is ketté lehet bontani aszerint, hogy mi történhet meg a külvilágban és mi nem. Ez alapján létezik a **lehetséges** és a **kizárt** környezeti követelmények halmaza.

A rendszert axióma szinten leíró halmazba tartoznak a **tervezési specifikációk**. (3. ábra) Ezek állhatnak elvárt működést **megadó** (zöld) illetve kizárt működést **tiltó** (piros) állításokból.

[1] SIL – Safety Integrity Level: egy olyan metrika, amely egy adott biztonságkritikus biztonságintegritási szintjét reprezentálja.



3. ábra Felhasználói követelmények, rendszerkövetelmények és tervezési specifikációk közti kapcsolat

A fenti ábrában az *Univerzum* a megvalósítandó rendszer és működési környezetét befoglaló teljes külvilágot jelenti. Az ábra kívülről befelé haladva fejezi ki azt, hogy az egyes követelmények a *felhasználói követelmény – rendszerkövetelmény – tervezési specifikációk* befelé haladva hogyan válik egyre konkrétabbá. A felhasználói- és a rendszerkövetelményeket együttesen *külvilágból érkező követelményekként* lehet értelmezni.

- A külvilágból érkező követelményeken belül a
  - *lehetséges környezeti viselkedések* halmazát a zölddel jelölt szegmens,
  - *kizárt környezeti viselkedések* halmazát pedig a piros szegmens reprezentálja.
- Hasonló módon a tervezési specifikációk halmazán belül a
  - *kötelező rendszer működések* halmazát a zölddel jelölt részhalmaz,
  - *a tilos rendszer működések* halmazát pedig a zöld részhalmaz jeleníti meg.

Az ábrán megfigyelhető, hogy a külvilágból érkező tiltott működés specifikáció eltér a tervezési specifikációkban szereplő tiltott működésektől. Ennek oka, hogy bizonyos megkötések implementálása során a rendszerben szereplő axióma nem tiltásként, hanem kötelezően teljesítendő funkcióként jelenik meg. (Pl. annak a rendszerkövetelménynek az implementálása, hogy a rendszer ne legyen érzékeny egy fegyveres támadás ellen szerepelhet olyan axiómaként, hogy létezzen védelem fegyveres támadás ellen.)

2.2. Követelmény minőség és tervezési hatékonyság

2.2.1. Gazdasági háttér

A követelmények **rossz meghatározása** kritikus hatással bír a szoftver minőségére és előállításának költségeire. [32] Amennyiben nem egyértelmű, elégtelen, inkonzisztens vagy nem teljes követelményrendszer kerül kidolgozásra, akkor e hibák detektálása és javítása is nagy erőfeszítést igényel. A fejlesztésnek minél későbbi fázisában fedezzük fel a hibát - azaz a követelmény specifikálás időpontjától időben minél távolabb - a javítási költségek annál magasabbak lesznek.

A rossz követelmények nem csak a költséges hibajavításhoz, hanem akár az egész projekt bukásához is vezethetnek. Egy 2001-es felmérés szerint a szoftverfejlesztési projektek sikertelenségének okául a projektmenedzserek az esetek több mint 50%-ában a helytelenül megadott követelményeket jelölték meg. [32]

Nem csak az befolyásolja a projekt kimenetelét, hogy rossz követelmények helyett jók irányítsák a későbbi projekt fázisokat, hanem a jó követelmények **megfogalmazási módja** is. [33] Ennek vizsgálatára a COCOMO („*Constructive Cost Model*”) algoritmikus szoftver előállítási költségek megbecslésére felállított modell használható.

A COCOMO család specializációkra vonatkozó ágának egyik eleme a COSYSMO („*Constructive Systems Engineering Cost Model*”), amely a rendszertervezéssel kapcsolatos becslések módját foglalja magában. A COSYSMO által megfogalmazott rendszertervezés költségeit befolyásoló két faktor a következő:

- a rendszerben szereplő elemek számára, azaz a rendszer mértére vonatkozó faktor („*Size Driver*”), ami a „mit kell megoldani?” kérdésre válaszol,
- valamint az kialakítás erőfeszítésére („*Effort Driver*”) vonatkozó faktor, ami a „hogyan kell megoldani” kérdésre válaszol.

A méretre vonatkozó faktort kialakító elemek között a *követelmények száma* az egyik legfontosabb. Ugyanakkor az erőfeszítésre vonatkozó faktor alakulásában is helyet kapnak követelmények *érthetőségével* és a *komplexitásukkal* kapcsolatos aspektusok is.

# of System Requirements	Easy	Nom.	Diff.
# New	0,5	1,0	5,0
# Design For Reuse	0,7	1,4	6,9
# Modified	0,3	0,7	3,3
# Deleted	0,3	0,5	2,6
# Adopted	0,2	0,4	2,2
# Managed	0,1	0,2	0,8

4. ábra Különböző nehézségű követelményekkel kapcsolatos tevékenységek költségei

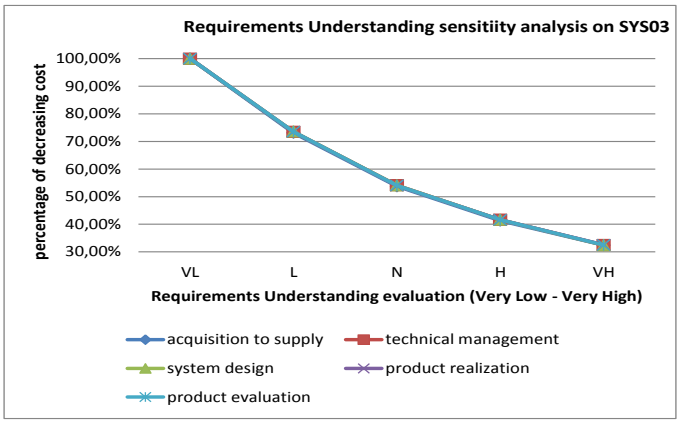
Egy újonnan felvett, átlagos nehézségű követelmény költsége vehető egységül. Ehhez képest, ha a tervezés közben az újrahasznosíthatóság szempontja is figyelembe van véve, akkor egy követelmény felvételének költsége 40%-al nő. Az újrafelhasználható követelmény módosításának költsége kb. 70%.

Ebből következik, hogy ha például három hasonló termékeket szeretnénk létrehozni és a hozzájuk tartozó követelmények létrehozása közben energiát fektetünk az újrahasznosíthatóságra is, akkor a költségeket kb. 20%-al is csökkenthetjük, hiszen:

$$100 + 100 + 100 > 140 + 70 + 70.$$

Minél többször hasznosítható újra egy követelmény valamilyen módosítással, annál költséghatékonyabb a kivitelezés. A változtatások nyilvántartására az iparban széles körben elterjedt metrika a követelmény stabilitási index (RSI – „*Requirement Stability Index*”). [34] Ennek segítségével a fenti módosítások követhetőségét, irányítását és szervezését lehet könnyíteni.

Az *újrafelhasználhatóság* mellett a másik fontos szempont az *érthetőség*. Ahogy az 5. ábra mutatja, amennyiben a követelmények érthetősége igen magas, akkor ez a fejlesztés összköltségét akár 30%-ra is redukálhatja.



5. ábra Követelmények érthetőségének hatása a költségekre nézve

2.2.2. A formalizálás előnyei

A követelményrendszerrel kapcsolatos fenti definíciók és gazdasági megfontolások jól illusztrálják mindazokat a nehézségeket, amelyekkel a hagyományos szöveg-alapú követelményspecifikációs folyamatok az egyes alkalmazásoknál küzdenek. A szabatos és helyes specifikáció a fejlesztési folyamat kulcseleme, azonban a verbális megfogalmazások annak ellenére, hogy jobban érthetőek, nem kellően precízek, így félreérthetőek, és ellenőrzésükre csak az ember által végzendő szakértői lektorálás alkalmas.

Természetes igény, hogy a követelményrendszer fogalmait lehetőség szerint matematikai szabatossággal lehessen definiálni és az ilyen, immár precíz keretet meta-modellként használva az egyes alkalmazások modelljeit is egyértelmű és ellenőrizhető módon lehessen megalkotni.

Természetesen az alkalmazói rendszereket nem absztrakt modellezés-elmélettel foglalkozó szakemberek végzik, hanem az adott szakterületet ismerő specialisták. A formális leírásokat olyan módon kell megválasztani, hogy azok elterjedt, a műszaki gyakorlatban jártas szakemberek számára közérthető és elterjedten használt modellezési paradigmákhoz illeszkedjenek.

2.3. Célkitűzés és motiváció

2.3.1. Tipikus elvárások

A fenti fejezetek konklúziója, hogy a követelményspecifikálás során gazdasági megfontolásokból hangsúlyt kell fektetni a megfogalmazás egyértelműségére és érthetőségére, valamint az egyes követelmények újrahasznosíthatóságára.

A követelményrendszerekkel szemben támasztott további fontos elvárás, hogy logikailag helyes legyen, azaz

- ne szerepeljenek benne egymással ellentmondó követelmények (inkonzisztencia) és
- legyen teljes, azaz minden releváns követelmény szerepeljen benne.

A harmadik fontos elvárás a szolgáltatásbiztonságra vonatkozik:

- legyen robusztus, azaz a környezet változásaival és a felhasználóktól érkező követelményváltoztatásokkal szemben ne legyen érzékeny. Amennyiben nem lenne robusztus, hanem érzékeny lenne a fenti változásokra, akkor képtelen lenne rájuk helyesen reagálni, azaz bekövetkezésük esetén helytelenül működne tovább.

Ahhoz, hogy a fenti tulajdonságokat vizsgálni lehessen, a rendszerkövetelmények valamilyen formális modelljét kell felállítani „*A formalizálás előnyei*” fejezetben kifejtett okok miatt.

A legtöbb modellező eszköz képes a rendszer konzisztenciájáról dönteni és a teljesség eléréséhez is léteznek olyan algoritmusok, amelyek irányított módon segítenek a szakembereknek felmérni a lehetséges hiányosságokat.

2.3.2. Érzékenységanalízis és SARI

Azonban mindmáig hiányzik az az eszköztár, amellyel fel lehetne mérni a funkcionális követelményekben vagy a környezetben bekövetkező változások hatását. Az általános mérnöki gyakorlat az ilyen problémák széles körére érzékenységanalízist használ, de az informatikában – néhány speciális terület kivételével - ez még nem elterjedt.

Az érzékenységanalízis tehát egy olyan módszer, melynek segítségével a rendszer robusztusságát lehet vizsgálni. Minél robusztusabb, annál kevésbé érzékeny a változásokra, tehát több változás típus esetén képes továbbra is helyen működni.

Dolgozatom legfontosabb eredménye ennek a rendszerkövetelményekre vonatkoztatott megvalósíthatóságára egy lehetőség bemutatása.

A módszer lényege, hogy a követelmények érvényességének kikapcsolásával a rendszer különböző mutációi hozhatóak létre. Ezek vizsgálatával megállapítható, hogy az éppen kikapcsolt követelmény mennyire kritikus a rendszerben, és a kapott eredmények alapján szükséges-e a rendszert további axiómákkal kiegészíteni vagy esetleg el lehet közülük valamelyiket hagyni.

Mivel a rendszerkövetelmények specifikációja hierarchikus, többféle tudást igénylő folyamat, ezért a folyamat különböző pontjain különböző eszközök és különböző szaktudást igénylő emberi munkaerőre van szükség. Az első szakaszban a leggyakrabban használt technológiák közé tartozik az ontológiákat leíró OWL 2 nyelv alkalmas arra, hogy az adott szakterületről specifikus tudással rendelkező szakember tudását elkezdje összegyűjteni és bizonyos követelmények is megfogalmazhatóak benne.

Azonban nem alkalmas a felállított rendszer konzisztenciáján túlmutató kérdések megválaszolására. Például nem lehet rákérdezni egy adott összetett követelmény teljesülésére egy adott mutációban.

A második technológia, az Alloy viszont képes erre. Azonban az Alloy-ban történő modellépítéshez már modellező szaktudásra van szükség, ez már a következő fázishoz tartozik.

A rendszerkövetelmények specifikációjának folyamatához az említett két technológia közti konverzió a megvalósításával hatékony támogatást nyújthat, hiszen a szakterület specialistája által létrehozott tudásbázist automatikusan konvertálná a modellező által használt modellező eszköz formátumára, amit a modellező ki tud egészíteni, és amin különböző analíziseket tud elvégezni.

Ennek megfelelően egy olyan alkalmazás implementálása a cél, melynek kettős funkciója van:

- OWL 2 és Alloy közötti konverzió és
- érzékenységanalízis megvalósítása.

Az alkalmazás jelenlegi neve „Sensitivity Analysis of Requirements Integrity”, azaz SARI.

Az alábbiakban ismertetem az érzékenységanalízis és a SARI helyét a rendszerkövetelményeket kialakító folyamatban. (6. ábra)

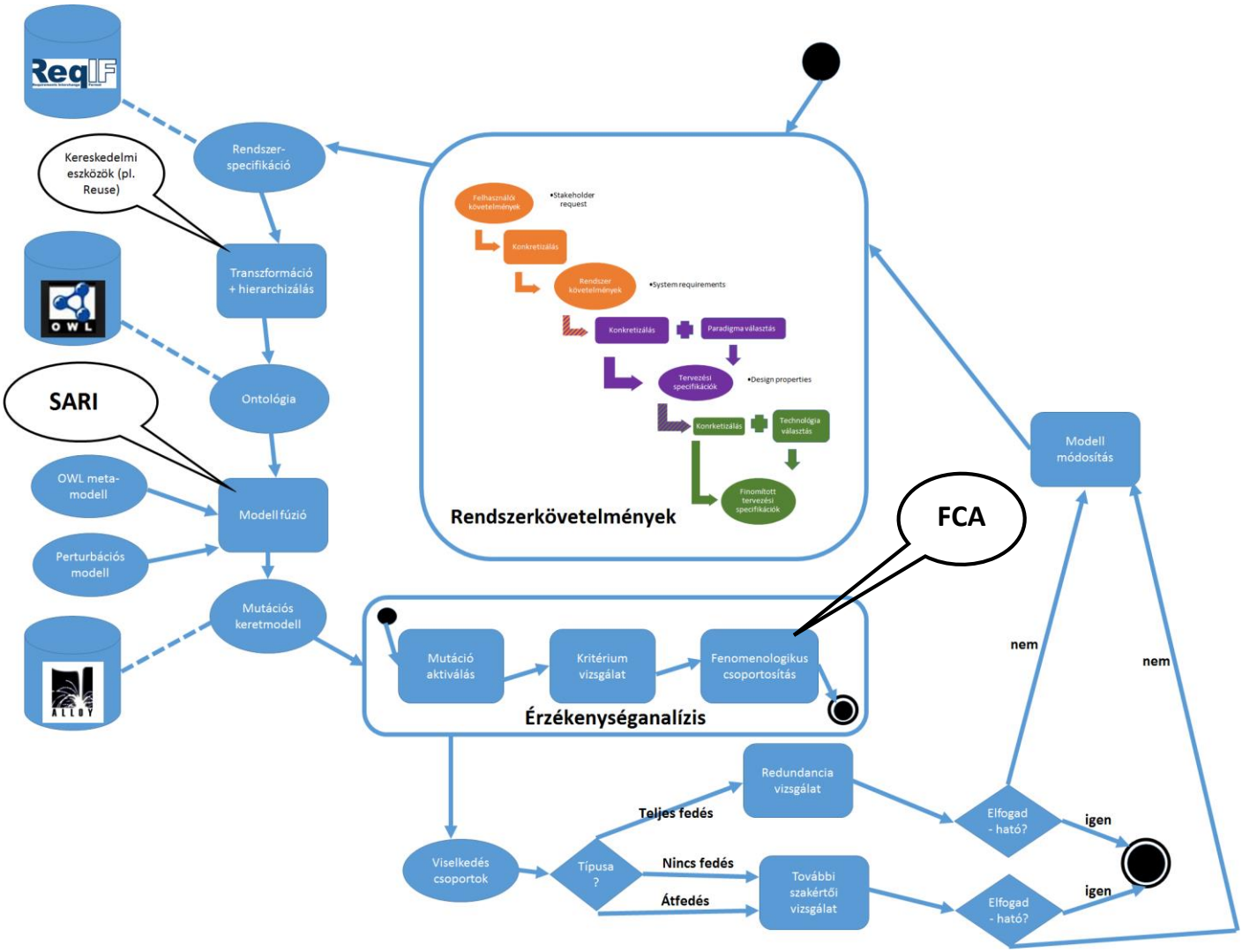
A hierarchikus rendszerkövetelmények meghatározás folyamatát követően létrejön a rendszerspecifikáció, például ReqIF formátumban. A különböző kereskedelmi eszközök, mint például

a „Reuse” segítségével ezek hierarchiába szervezhetőek és áttranszformálhatóak ontológiává, melyet például az OWL 2 nyelv segítségével lehet leírni.

A SARI alkalmazás bementként megkapja a létrejött ontológiát, az OWL meta-modellt és valamilyen perturbációs modellt (azaz hogy mi alapján végezze a mutáció generálást) és ezekből létrehoz egy mutációs keretmodellt, melynek leírásához az Alloy modellező eszközt használja. A mutációs keretmodell már tartalmazza a kapcsolókat, melyek segítségével a különböző modell entitások érvényessége kikapcsolható.

Az érzékenységanalízis folyamata a mutációk aktiválásával kezdődik. Az Alloy az adott mutációt megvizsgálja a különböző követelményekből származtatott kritériumok alapján, (pl. továbbra is biztonságos a rendszer?). Ezt követően a későbbiekben bemutatandó formális fogalmi analízis segítségével a kimeneteik alapján fenomenologikus csoportokba lehet a mutációkat sorolni.

A kimenetek alapján kiderül, hogy az adott mutáció kimutat-e megerősítendő pontot, vagy redundanciát. Ezt követően már lehet tudni, hogy hol kell a rendszerben további szakértői vizsgálatokat végezni vagy a redundancia létjogosultságát validálni. Ezen vizsgálatok eredményeként nagy eséllyel a kiinduló modell módosítására van szükség.



6. ábra Összefoglaló

2.4. A dolgozat felépítése

Az alábbiakban ismertetem a dolgozat fejezeteinek összefoglalóját.

A rendszerkövetelmények definiálása és a követelményspecifikációs folyamat gazdasági jelentőségéről, valamint ebből fakadóan a bemutatandó érzékenységanalízis jelentőségéről volt eddig szó. Ebből következnek az implementálandó alkalmazás, a SARI céljai: OWL 2 és Alloy közötti konverzió, valamint az érzékenységanalízis megvalósítása.

A követelmények modellezéséhez szükséges eszközök valamint a követelmények csoportosításának módja és a specifikációval kapcsolatos nehézségeket a harmadik fejezet mutatja be.

A követelmények csoportosításáról és a specifikációval kapcsolatos nehézségekről a „*Rendszerkövetelmények formalizálása*" fejezetben esik szó. Ezt követően a

- modellezés és a követelmények kapcsolata,
- a modellezési megközelítések,
- az ezeket implementáló technikák,
- a hozzájuk tartozó, rendelkezésre álló eszköztámogatás,
- a követelmények ábrázolásának módja a különböző modellekben,
- és a különböző modellek helye a tervezési folyamatban a dolgozat tárgya.

Az érzékenységanalízis megvalósítását és eredményeinek interpretációját a negyedik fejezet mutatja be. A bemenetei a rendszer különböző mutációi, kimenetként pedig három lehetséges eset létezik:

- Teljes fedés
- Nincs fedés
- Átfedés

Mind a három eset további szakértői vizsgálatot, vagy redundancia vizsgálatot von maga után. A lényeg eredmény azonban az, hogy az érzékenységanalízis kimenetei rámutatnak a rendszer azon pontjaira, amelyek irányában érdemes a további vizsgálatokat végezni.

Az érzékenységanalízis során a nyílt- és zártvilág modellezési filozófia közti különbség hatalmas szerepet játszik, hiszen általában a zárt modellek egy ponton történő megnyitásával lehetséges a megerősítendő pontok kiszűrése.

A két szemantika közti különbség megadása után a megvalósíthatóságra vonatkozó új ötlet, a kapcsolók bevezetésének módját mutatom be. Ezek segítségével a modellben szereplő axiómák érvényessége relaxálható (például zártvilág szemléletből lehet nyíltvilág szemléletre váltani.) A fejezet vége az érzékenységanalízis korlátait és a további kidolgozandó kérdéseket gyűjti össze.

A következő fejezet a formális fogalmi analízis (FCA) matematikai fogalomalkotó eljárásról és az érzékenységanalízis során a lehetséges felhasználási módjairól szól.

Az „*Analízis eszközök és feladatok*" fejezet bemutatja a tipikus követelményanalízis feladatokat és kapcsolatukat az érzékenységanalízissel, valamint az ebből származó elvárásokat az érzékenységanalízis megvalósításával szemben. Bizonyítja, hogy az ontológia kezelő eszközök erre nem alkalmasak, a folyamat következő fázisában más modellezőre van szükség.

A fenti igények kielégítésére az Alloy nyelv viszont alkalmas. Ennek bizonyítását követi az ontológiákat leíró OWL 2 és Alloy közti egyirányú konverzió megvalósíthatóságának bizonyítása, majd néhány OWL 2 axióma Alloy-beli reprezentációja olvasható.

A hetedik fejezetben a SARI funkcióiról és előrelátható hiányosságairól esik szó. Az architektúrális felépítése és az implementálása közben használt technológiák bemutatása után a jelenlegi működés és a felhasználó esetek („*use-case*”) demonstrálása következik a szállítmányozási példa segítségével.

A „*Példa*” fejezetben a fent említett szállítmányozási példa részletes leírása és rajta elvégzett érzékenységanalízis eredményeinek értékelése szerepel.

Az utolsó fejezet az eddigi munkát értékeli és foglalja össze. Tisztázza mind a megvalósítással kapcsolatos, mind az alkalmazás jelenlegi állapotával kapcsolatos hiányosságokat és az ezekből következő a bővítési lehetőségeket.

3. Rendszerkövetelmények formalizálása

3.1. Modellezés célja

Az informatikában a modellezés célja, hogy az implementálandó rendszer egy egyszerűsített képét hozzuk létre. Ez egy olyan véges matematikai konstrukció, amely megkönnyíti a tervezést és segítségével ellenőrizhetőek a rendszer bizonyos tulajdonságai. Előnye, hogy áttekinthetőbb és a rajta végzett szimulációk általában sokkal olcsóbbak, mint a valóságos rendszeren.

A modellben definiálni kell, hogy a rendszert milyen aspektusból írja le, illetve hogy milyen szempontokat nem vesz figyelembe (azaz hogy mennyire egyszerűsítünk a valósághoz képest). A rendszer működésére tett megállapítások csak a megszabott világon belül lesznek érvényesek, ezért nagyon fontos a külvilágra vonatkozó követelmények részletes specifikációja is.

A modellezés során két fontos aspektust kell figyelembe venni. Az egyik a validáció kérdése, azaz hogy a definiált rendszer jó-e, a másik pedig a verifikáció, azaz hogy a rendszer megfelel-e a modellnek. Az érzékenységanalízis a validáció témakörbe tartozik.

3.2. Követelmények és rendszerspecifikáció kapcsolata

Az informatikai rendszer által implementálandó célok meghatározása során a rendszerrel szemben támasztott követelményeket tisztázzák. Csak ezt követően alakítják ki a szoftver specifikációját a követelmények átalakításával.

3.2.1. A követelményrendszer felállításának kihívásai

Fontos tehát megvizsgálni, hogy mi okozza a fő nehézségeket a követelmények meghatározása során. Az egyik legfőbb probléma, hogy a hatókörük igen kiterjedt. Ez abból ered, hogy nem csak az elkészítendő szoftvert kell definiálniuk, hanem azt a környezetet is, amiben ez a szoftver működni fog, ami gyakran igen komplex leírást von maga után. Szükség lehet például fizikai törvények megfogalmazására vagy vállalti környezetben az összes vállalati folyamat figyelembevételére stb. Sokszor már önmagában az is gondot okoz, hogy a környezetet és a szoftvert külön kell kezelni. (Pl. egyes modellező eszközökben a kettő leírásának módjában nincs különbség, és csak a szemantikai értelmezés segít a szétválasztásban.)

További nehézséget okozhat, hogy a rendszerrel szemben állított extra-funkcionális követelmények egymással ellentmondásosak lehetnek. Előfordul, hogy a működést több résztvevő befolyásolja, akiknek érdekeik nem azonosak. Például egy autó alkatrészeket vezérlő mikrokontroller egyik komponensének célja, hogy az autó minél gyorsabb haladásra legyen képes, míg egy másik komponens a lehető legnagyobb biztonság elérésére törekszik. Az ilyen esetekben szükséges priorizálni vagy valamilyen optimumokat keresni.

Alex van Lamsweerde tanulmányában [36] leírtak szerint a rendszertervezést hátráltatják a fentieken kívül az alábbi tényezők is:

1. Bizonyos alapfogalmak (konzisztencia, teljesség) gyakori félreértése
2. Konstruktív eljárások helyett (azaz már kidolgozz fogalmak és technikák kombinálása) újak kialakítására való természetes törekvés
3. Állandó konfliktus a formális leírás precizitása és az informális leírás érthetősége között

A fenti kihívásokból fakadóan gyakran előfordul, hogy a követelményrendszer felépítésében az ütközéseken és az inkonzisztencián kívül más hibák is jelentkeznek. Ezek között vannak olyanok, amik súlyosak, elkövetésük az egész informatikai rendszer specifikálásának helytelenségét vonják maguk után. Erre a legfontosabb három példa:

- nem teljesség (incompleteness)
- elégtelenség (inadequacies)
- Többértelműség (ambiguities)

Léteznek olyan hibák is, amik csak hátráltatják a fejlesztést, de nem feltétlenül okozzák a rendszer helytelenségét:

- zavarosság (noises)
- túlspecifikálás (overspcefication)
- körbehivatkozások (forward references)
- wishful thinking (vágyálom)

Ezek elkerüléséhez szükségesek olyan eszközök, amik támogatják a helyes követelményrendszer felépítését. Ezeket a „*Modellezés és követelmények kapcsolata*” fejezet mutatja be.

3.2.2. Követelményfajták

A követelményeket több szempontból lehet osztályozni. A besorolás megkönnyíti a kezelésüket és a leírásukat. Néhány modellező nyelvben különböző típusú követelményeket különböző szintaxissal kell ábrázolni, ezzel támogatva a rendszer átláthatóságát. A követelménytípusokat szintén az Alex van Lamsweerde [36] cikk gondolatmenete alapján ismertetem.

Az egyik fontos aspektus, hogy milyen típusú kérdésre adnak választ:

1. **MIÉRT?** típusú követelmények: A célok definiálására szolgáló rendszerfunkcióik. Ezek leírásához az adott szakterület mély ismerete szükséges, hiszen a szakterület-specifikus problémák és lehetséges szituációk analizálásával nyerhetőek ki.
2. **MIT?** Típusú követelmények: Céljuk a minőségbiztosítás, megkötéseket és extra-funkcionális elvárásokat fogalmazna meg.
3. **KI?** típusú követelmények: az előző két típusú követelményekhez szerződéses formában felelőst kell rendelni. Ilyen felelős lehet a szoftver egy komponense vagy a külvilág ez eleme.

Megkülönböztethetőek a környezet szempontjából is. Azok feltételek, amik a környezet írják rá (pl. a már említett fizikai törvények vagy vállalti folyamatok működése) nem a rendszerrel szemben támasztott követelmények, hanem úgynevezett „*domain property*”-k, azaz környezeti változók.

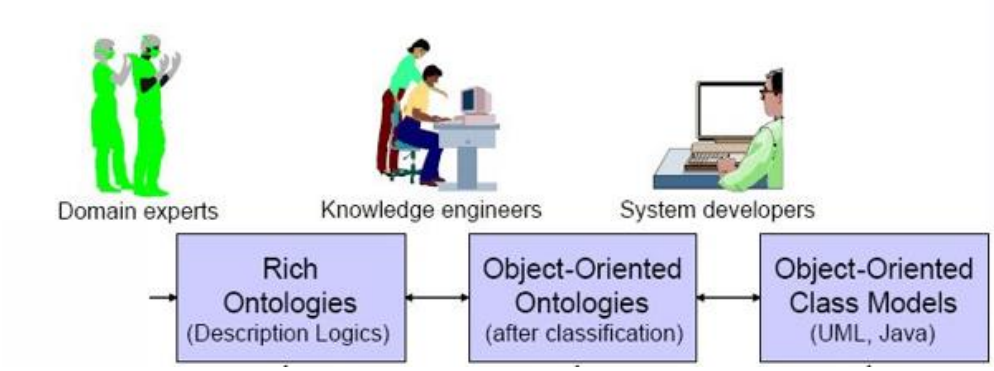
Különbség van aszerint is, hogy kiknek a nyelvezetén történik a megfogalmazás. A *követelmények* maguk a felhasználók számára érthető nyelvezettel íródnak le, hiszen ők fogalmazzák meg az általános elvárásokat. Ezen a szinten az elvárt relációk definiálása a környezetben létező és a szoftver által irányítandó objektumok között történik. A *szoftverspecifikáció* szintjén ezek az objektumok a programozó számára érthető nyelven, mint szoftverbeli entitások (pl. változók, adatbázis rekordok stb.) kerülnek leírásra. A fizikai és a virtuális világ közti összefüggések megadására a *hitelesítési pontok* szolgálnak.

Felelősség szempontjából elkülöníthetőek a *követelmények*, melyek teljesülését az implementálandó szoftvernek kell kikényszeríteni, illetve az *előfeltételek,* melyekért a szoftver környezetében lévő szereplők felelősek.

3.3. Tervezési fázisok

A kiinduló filozófia alapjait, mely szerint előbb definiáljuk, hogy **mit** kell mondani, és csak azt követően azt, hogy **hogyan** kell mondani H. Knublauch egy 2003-as cikkében fektette le. [35] Ez a tervezést három részre bontja. A három különböző szakaszban három különböző típusú emberi erőforrásra van szükség. Ahogy haladunk az időben, az adott szakterület szakértője egyre kisebb mértékben vesz részt a tervezésben, a követelmények rendszerezése, analizálása már nem az ő felelőssége. Ez összhangban van a fejezet elején kifejtett követelmények között fennálló hierarchikus viszonnyal.

Ugyanakkor a modell finomításának támogatására csak fél-automatikus eszközök léteznek, ezek használatához mindenképp szükség van a szakértő bevonására.



7. ábra Knublauch-féle tervezési fázisok emberi erőforrásai

A követelményspecifikáció folyamata során az első fázisban a *szakértők* elmondása alapján rögzítjük a magas szintű elvárásokat. Először a követelmények és a taxonómia is csak strukturálatlan formájú. A cél ebben a fázisban, hogy a szakértő számára a tudásátadás minél egyszerűbb módon kivitelezhető legyen.

Ezt követően a megadott követelményeket valamilyen struktúrába kell rendezni és ellenőrizni kell az általuk felállított rendszer konzisztenciáját. Ezen vizsgálatot már nem az adott szakterület szakembere végzi, hanem az úgynevezett *tudás elemzők*. Ebben a fázisban a legfontosabb cél a példányok helyes felvétele.

A harmadik fázisban a programozók által létrehozandó szoftver komponensekkel szembeni követelményeket határozzák meg: a tudás elemzők által rendszerezett tudáson alapuló program implementálása a feladat.

3.4. Modellezési megközelítések

A modellezés során három megközelítés létezik: informális, fél-formális és formális. Ezen módszerek rendre egyre precízebb leírást biztosítanak, azonban az érthetőség és a modellező eszközök használata ezzel párhuzamosan egyre nehezebbé válik.

A tervezés során az informális módszerektől haladunk a formális módszerek felé, így biztosítva a szakterület szakértője számára - aki főleg az első fázisban vesz részt – a könnyebben kezelhető eszközök használatát.

3.4.1. Informális

Az informális módszerek során általában a természetes nyelveket (pl. angol) használják a leíráshoz. Útmutatóul sokszor használnak valamilyen szabvány által előre definiált módszert a követelmények összegyűjtésre.

Az előnye, hogy bárki számára érthető leírást ad egy adott rendszerről és informális leírást viszonylag gyorsan lehet készíteni. A rendszertervezés első fázisában gyakran alkalmazzák ezt a módszert: a követelmények összegyűjtésére szóbeli és írásbeli informális interjúkat készítenek a leendő felhasználókkal.

Fontos tisztázni, hogy nem csak a felhasználóktól érkezhetnek követelmények, hanem más, úgynevezett „stakeholder”-ektől is származhatnak (nem feltétlenül funkcionalitásra vonatkozó) igények.

Az informális leírás azonban magában hordozza egy fogalom többértelműségének veszélyét. Mivel nem precíz, elérőre definiált szemantikájú komponensekre épít, ezért az ily módon leírt modelleket nehéz kommunikálni, ellenőrzésük csak emberi munkával lehetséges.

3.4.2. Fél-formális

A fél-formális modellezési eljárások során a modell valamely aspektusának (általában a fogalmak és azok közti relációk) formális leírása mellett megjelennek még olyan részek, melyek megjelenítése még nem formális úton történik (pl. a fogalmakhoz kapcsolódó tulajdonságok). Ennek egyik előnye, hogy a modell még mindig könnyen értelmezhető, a fél-formális módszereknek általában van valamilyen grafikus megjelenítési formájuk, a rendszer különböző szempontokból írható le és sok olyan szoftver létezik, ami támogatja az ilyen típusú modellek elkészítését. Azonban ezek a modellek általában csak a MIÉRT (lásd Rendszerkövetelmények fejezet) kérdésre adnak választ, nehéz rajtuk bármilyen analizálást elvégezni és a grafikus ábráknak akár több interpretációja is létezhet. Az UML szabványban szereplő diagramok közül sokan követik ezt a filozófiát. Az alábbiakban ismertetem a három legelterjedtebb fél-formális grafikus diagramot.

Az *entitás-reláció diagramot* (ER diagram) általában adatbázis tervezés során használják. Az alapvető komponensei az entitások - objektumok osztályainak kifejezésére - valamint a köztük lévő relációk. Az osztályok rendelkezhetnek közös attribútumokkal, a relációk irányítottsága és számossága is megadható és a hierarchikus strukturálásra is lehetőség van a specializáció reláció segítségével.

Az *adatáramlás („dataflow”) diagram* az UML 2 szabvány részeként jelent meg. Segítségével egy informatikai rendszerben történő adatáramlás folyamatát lehet grafikusan ábrázolni. Alapvető komponensei a funkciók, az adatbázisok és az input/output-ok, melyek közti nyilak az adatok áramlását jelzik. Kifejtésük történhet több szinten, egy funkció finomítása lehet egy másik dataflow diagram.

A harmadik az *állapotátmenet diagram*, ami az egyik legrégebben használt tervezési eszköz az objektum orientált szoftverfejlesztésben. [38] Az alapvető elemei a rendszer állapotai, melyeket a külső világból érkező események befolyásolnak. A nyilak a köztük lévő átmeneteket és azok trigger feltételeit reprezentálják.

3.4.3. Formális

A formális modellezés során a fogalmak és a hozzájuk tartozó tulajdonságok specifikálása is valamilyen matematika alapú formális nyelven történik. Ennek lényege, hogy a szintakszis és a szemantika is előre definiált, valamint a modell vizsgálata során feltehető kérdéseket is egyaránt egzakt módon fogalmazzák meg. Ennek köszönhetően a modellen különböző analízisek és vizsgálatok végezhetőek

el, valamint a kielégíthetőségére példa vagy a kielégíthetetlenségére ellenpélda generálható. A másik előnye, hogy a precíz leíró szabályoknak köszönhetően minden információ egyértelműen megadható.

A precizitás miatt viszont az ilyen modelleket nehéz írni és olvasni, ezért a hibázási lehetőség is nagyobb. Elkészítésükhöz nem csak a szakterület ismeretére van szükség, hanem a formális leírás technikájához is érteni kell. Az alábbiakban bemutatok néhány elterjedt formális modellező szemléletet és kitérek arra, hogy az érzékenységanalízis megvalósításához melyiket választottam és miért.

- A *történet-alapú specifikációban* („history-based specification”) összegyűjtik a rendszer időbeni összes elfogadható viselkedését. Ezek leírása a temporális logikai állításokkal történik, melyekben a múltra, a jelenre és a jövőre vonatkozó logikai operátorok szerepelnek. Fontos mérnöki döntés az ilyen modellek esetén, hogy az idő kezelése folytonos vagy diszkrét módon történjen.
- Az *állapot-alapú specifikáció* („state based specification”) során a rendszer összes lehetséges állapotait határozzák meg úgynevezett pillanatfelvételek („snapshot”) készítésével. Definiálni kell, hogy egyik állapotból a másikba milyen elő-és utófeltételekkel lehet átjutni. Állapotalapú specifikációra nyújt lehetőséget például a halmaz-elméleten alapuló Z nyelv.
- Az *átmenet-alapú specifikáció* („transaction-based specification”) az állapotosztályokból való átmenetek leírására koncentrál: a bemeneti állapotok és triggerek függvényében ábrázolja a kimeneti állapotokat. Az átmenetek számára megadhatók perkondíciók, melyek szükséges, de nem elégséges feltételek, illetve triggerek, melyek hatására az átmenet azonnal bekövetkezik.
- A *funkcionális specifikáció* („funcional specification”) során a rendszert matematikai függvények strukturált halmazaként kell megadni. A strukturálásnak két lehetséges módja van: történhet algebrai vagy magasabb rendű függvények alapján. Az előbbi esetben a függvényeket az értelmezési tartományukban szereplő objektumok típusa alapján kell csoportosítani, az utóbbiban a logikai állítás típusa szerint (pl. típus definíció, változó deklaráció, implikáció stb.).
- A *működtetési-alapú specifikáció* („operational specification ”) valamilyen absztrakt gép által végrehajtható konkurens folyamtok segítségével definiálja a rendszert. Ez a szemlélet áll a legközelebb az imperatív programozás filozófiájához.

3.5. Követelménymodellezés és eszközei

3.5.1. Informális

Az egyik legelterjedtebb informális modellezést támogató szabvány az IEEE által dedikált 830-as szabvány, amely egy könnyen követhető folyamat mellett azt is definiálja, hogy milyen kritériumoknak kell megfelelnie egy jó szoftverkövetelménynek. Az alábbiakban a szabvány leírását követve emelem ki a jó szoftverkövetelmény specifikációkkal (SRS – Software Requirement Spceification) kapcsolatos fontosabb aspektusokat és tulajdonságaikat.

A követelmények megadása során öt pontot kell végiggondolni. Az első a *funkcionalitás*, azaz hogy mit kell a szoftvernek csinálnia, mik a fő célok, melyeket teljesíteni kell. Ezek specifikálása során szokás az alapvető felhasználói esetek (use-case) definiálása. A második a külvilággal való érintkezés módjának, azaz a külső *interfészeknek* leírása. Ezek szabják meg, hogy hogyan lép kapcsolatba az emberi felhasználókkal, a rendszeren belüli és kívüli hardverekkel illetve egyéb szoftverekkel. Ezt követi a *teljesítményre* (gyorsaság, elérhetőség, válaszidő, visszaállítási idő stb.) és a rendszer *attribútumokra* (hordozhatóság, fenntarthatóság, helyesség, biztonság) vonatkozó *extra-funkcionális* követelmények meghatározása. Végül ellenőrizni kell, hogy vannak-e a szoftverre vonatkozóan külső *tervezési*

kényszerek, (pl. szabványoknak, valamilyen operációs rendszeren való futtathatóság, implementációs nyelv, adatbázis integrációs szabályok) melyeket ki kell elégítenie.

A szabványban nyolc tulajdonságot rögzít, melyekkel a jó követelményeknek rendelkezniük kell. Az első a *helyesség*: minden követelményt, amely a szoftverkövetelmény specifikációban szerepel, a szoftvernek ki kell elégíteni. Ezek származhatnak a külső környezetből (pl. amennyiben a SW egy magasabb szintű rendszer része, akkor ennek a specifikációjában leírtaknak meg kell felelnie). Ennek ellenőrzése igen nehéz, mert nem létezik olyan eszköz vagy folyamat, amely használata ezt a fajta helyességet garantálná.

A többértelműség elkerülése végett a követelményekkel szemben támasztott második fontos elvárás az *egyértelműség*. Informális modellekben ennek megvalósítása azonban igen nehéz. Minden követelménynek pontosan egy interpretálási lehetőséggel kell rendelkeznie. Amennyiben ez nem teljesül, akkor a specifikációhoz tartozó szótárban egyértelműsíteni kell azon fogalmakat, amelyek a követelmények feldolgozására több lehetőséget is adnak.

A harmadik tulajdonság a *teljesség*: a fent említett öt aspektussal kapcsolatban az összes követelménynek szerepelnie kell a rendszerben. Ezen felül definiálni kell minden összefüggést a rendszer komponensei között valamint minden lehetséges input típusra adandó válaszát (érvényes és érvénytelen bemenetre is).

A helyességhez hasonló elvárás a *konzisztencia*, ami a belső helyességre vonatkozik: nem létezhet a követelményeknek olyan részhalmaza, amelyek egymással összeférhetetlenek. Ennek három típusa van. A való-világ objektumaira vonatkozó ellentmondások (pl. egy osztály egy változójának két különböző típusú deklarációja van), a logikai ellentmondások (pl. egy folyamatban A és B állapot között egyszerre kellene fennállnia párhuzamosságnak és egymásutániságnak is) és a többértelműségből származó ellentmondások (pl. az egyik követelményben a fájlrendszer elemeire mappaként („*folder*”) egy másikban pedig könyvtárként („*directoy*”) hivatkozunk).

Általában a követelmények nem azonos jelentőségűek. Vannak olyanok, melyek nem teljesülése a biztonság vagy stabilitás szempontjából kritikus következményeket von maga után. Ezen pontokra különös figyelmet kell fordítani, és külön meg kell őket jelölni. Tehát szükséges a *fontosság szempontjából kategóriákba sorolni* őket. Eddig kevés olyan módszer létezett, amely segítségével ez a besorolás automatikusan vagy valamilyen szoftveres támogatással elvégezhető lett volna. Az érzékenység vizsgálat segítségével pontosan az ilyen kritikus pontok megállapítására van lehetőség.

A követelményeknek *verifikálhatóaknak* kell lenniük. Nem csak a bennük szereplő fogalmaknak kell egyértelműnek lenniük, hanem a rájuk vonatkozó kritériumoknak is. A céloknak mérhetőnek kell lenniük, azaz a rendszerrel szemben támasztott elvárások ellenőrzésére léteznie kell valamilyen véges idejű, költséghatékony módszernek.

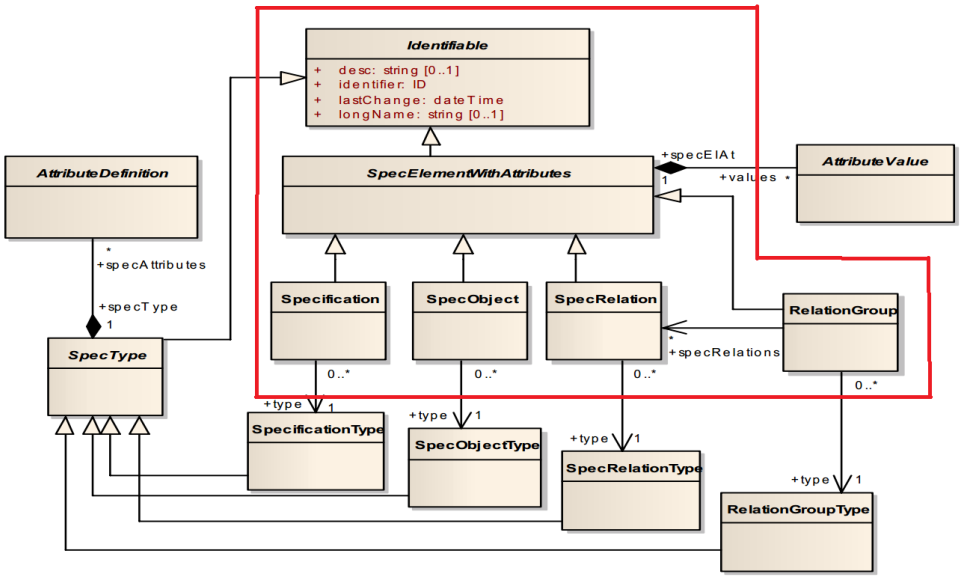
Az utolsó két tulajdonság a *változtathatóság* és a *követhetőség*. A szoftverkövetelmény specifikációjának struktúráját és a stílusát is úgy kell kialakítani, hogy bármelyik elemben történő változás könnyen és követhetően véghez vihető legyen. Ebből következik, hogy redundáns követelmények nem szerepelhetnek benne (hiszen ezek módosítása könnyű hibázási lehetőséget von maga után).

3.5.2. Fél-formális

Az ipari gyakorlatban létezik egy az OMG („*Object Managemenet Group*”) [48] által kifejlesztett ReqIF („*Requirement Interchange Format*”) [27] szabvány, melynek célja a követelmények egyértelmű leírására alkalmas XML alapú fájlformátum definiálása. Segítségével a követelmények és a rájuk

vonatkozó meta-adatok könnyen kommunikálhatóvá válnak különböző szervezetek között akkor is, ha azok nem ugyanazt a követelménykezelő eszközt vagy folyamatokat használnak.

Az általa biztosított nyelvi apparátus segítségével a fent említett hierarchiában bármilyen helyet elfoglaló követelmény leírása megvalósítható, tehát a SEBoK által megadott bármelyik szintű követelmény értelmezhető a ReqIF által definiált SpecObject-ként.

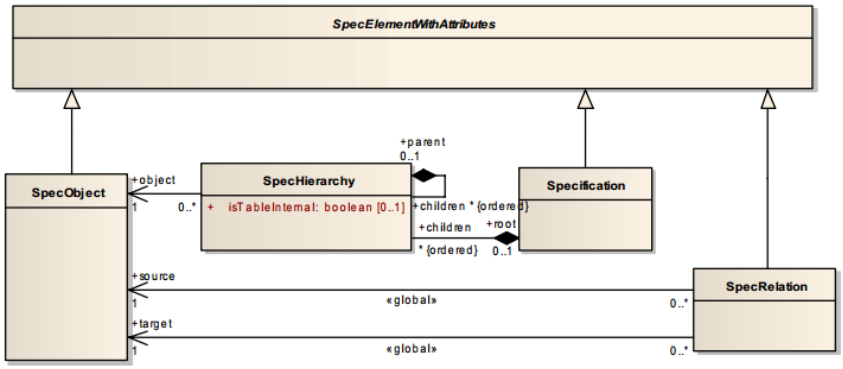


8. ábra ReqIF meta-modell

Az 8. ábra mutatja a ReqIF által definiált modellek meta-modelljét. Dolgozatomb a követelmények közti szabadon definiálható relációk és hierarchia viszonyok ábrázolásánál mélyebb leírásokat (például adattípusok kezelése) nem bont ki. (8. ábra – piros vonalon kívüli elemek), mert az általam használt követelmény definíció nem igényli ezeket és kezelésük kidolgozása aránytalanul nagy többletmunkát igényel.

A követelmények között fennálló kapcsolatok reprezentálására SpecRelation osztály szolgál, melynek segítségével elméletben unáris és bináris relációknál magasabb fokúak megadására is lehetőség van, de ezek szétbonthatóak ilyenekre. [40] Dolgozatomban azonban csak a fenti két típusú kapcsolatokat tartalmazó modellek vizsgálata a cél.

A hierarchikus viszonyok reprezentálására a SpecHierarchy osztály szolgál, segítségével egymásba ágyazott struktúra alakítható ki. (9. ábra) Ezzel a követelmények finomítási szintjeit lehet kifejezni. Ennek jelentősége abban rejlik, hogy amennyiben valamilyen vizsgálatot szeretnénk a rendszeren végezni, el lehet dönteni, hogy azt milyen követelményszinten történjen.



9. ábra Hierarchikus viszonyok megadása ReqIF-ben

Fontos része a szabványnak, hogy a bekövetkező változások követhetőségének biztosítása érdekében nagy hangsúlyt fektet a státuszukra vonatkozó meta-adatok átadására is.

Megadja a lehetséges átadási forgatókönyveket, a ReqIF információs modell absztrakt architektúráját, az átadásra alkalmas XML dokumentum struktúráját és annak részleteit, valamint a ReqIF XML séma előállításra vonatkozó általános szabályokat.

A fentebb említett grafikus fél-formális modellezési megközelítéseket különböző szoftverek támogatják. (Pl. Enterprise Architect). Ezeken kívül azonban meg kell említeni azokat a követelmény-kezelő szoftvereket is, amelyek a rendszer leírását nem objektum szinten, hanem a teljesítendő elvárások megadásával valósítják meg és támogatják a ReqIF modellek kezelését. A gyakorlatban a két legelterjedtebb eszköz az IBM DOORS és az Eclipse Requirement Modeling Framework (RMF).

IBM Rational DOORS („Dynamic Object Oriented Requirements System”) [41]

Az IBM DOORS (10. ábra) egy kliens-szerver architektúrájú követelménykezelő eszköz. A legfontosabb tulajdonságai a következőek:

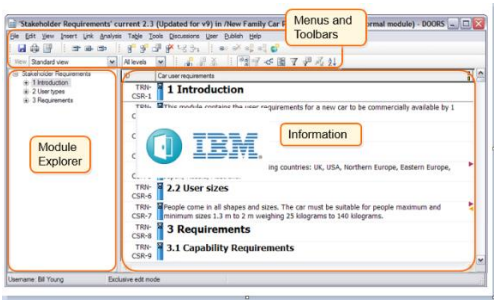
- központosított dokumentumkezelés
- követhetőség
- hatás analízis
- külső formátumok támogatása

A külső adatbázison tárolt adatokat különböző modulokba osztva kezeli. A fejlesztés során a felhasználóknak projekteket kell létrehozniuk, de a projektek közti összefüggések kezelése is támogatott. A projektekben objektumok szerepelnek, melyek valamilyen hierarchiába vannak szervezve. Ezek vagy „*heading*”-ek, vagy szöveges követelmények („*text*”).

A felhasználók menedzselése is magas szinten megoldott, pl. csoportokba szervezhetőek, ami alapján hozzáférési jogaik a különböző objektumokra külön-külön is beállíthatóak. A követhetőség érdekében a DOORS az objektumokkal kapcsolatos minden tevékenységet logol.

Az objektumokhoz különböző attribútumok rendelhetőek, melyek valamilyen rávonatkozó kiegészítő információt hordoznak magukban. Ilyen például az objektum prioritása, állapota és a hozzátartozó felelős, de saját típusú objektumok megadására is van lehetőség.

Szintén a követhetőséget segíti, hogy az objektumok között vagy valamilyen külső elemmel (pl. Word dokumentum) igen könnyen (akár drag&drop technikával) vehetőek fel különböző típusú linkek. Ezen linkek mentén jól követhetőek a változtatások és az IBM DOORS képes különböző dinamikusan frissülő riportokat létrehozni.



10. ábra IBM DOORS SnapShot

Eclipse Requirement Modeling Framework (RMF) [42]

Az Eclipse RMF (11. ábra) egy olyan opensource kiterjesztése az Eclipse-nek, amely segítségével ReqIF meta-modellt kielégítő, szöveg alapú követelményrendszerek létrehozása és kezelése valósítható meg.

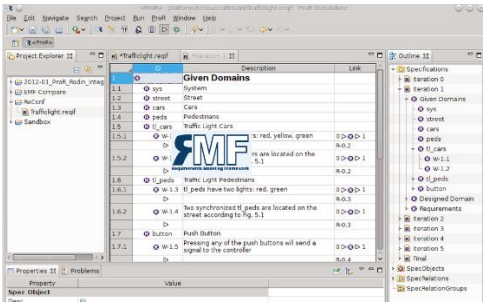
Az RMF maga két részből áll: a meta-modell kezelő motorból, illetve a ProR nevű grafikus felhasználói felületből.

Mivel a ReqIF-ben kidolgozott modell köré épül, ezért benne a követelmények „SpecObject”-ként definiáltak. Ezekhez tartozik egy „SpecType”, ami meghatározza, hogy milyen attribútumok tartoznak hozzá. A szabványban szereplő harmadik fontos elem, a „SpecRelation”-el pedig a követelmények között lévő linkek adhatóak meg, akárcsak az IBM DOORS-ban, itt is drag&drop technikával. Ezen kapcsolatok segítségével válnak a követelményekben bekövetkező változások követhetővé.

Az „Specification” elemek a követelmények között felépülő hierarchikus viszonyokat egy fa-struktúraként tárolják.

A fejlesztői környezetben alapvetően új projekt létrehozásakor egy új ReqIf modell jön létre, ami egy darab Specificationt tartalmaz, benne egy SpecObjecttel.

Az RMF egyik legnagyobb előnye, hogy követelményekkel kapcsolatos változáskövetést megvalósítja. Az ezzel kapcsolatos bizonyításokhoz az Event-B formális modellező módszert használja. A másik fontos előnye, hogy támogatja más modellekkel való integrációt.



11. ábra Eclipse RMF SnapShot

3.5.3. Formális

A követelmény fogalmának „Bevezető” fejezetben megadott definíciója alapján *„egy rendszer akkor illeszkedik egy követelményspecifikációhoz, ha*

- *a működése során a követelményspecifikációban szereplő összes követelmény az összes állapotában mindig teljesül*
- *az előre megadott, a külvilágra vonatkozó korlátozások által definiált környezeti megkötések mellett*."

Vizsgálataink tárgya a felhasználói rendszer, az abból származtatott rendszerkövetelmények és annak finomításával létrehozott tervezési specifikációk. A későbbiekben részletezendő módon az érzékenységanalízis ezekből származtatott rendszer-mutációkat vizsgál.

A definíció első pontja ugyanakkor előírja azt, hogy a rendszer összes állapotára teljesülnie kell a kritériumoknak. Ennek megfelelően a matematikai modell választásakor két feltételnek kell teljesülnie:

1. A matematikai paradigmának képesnek kell lennie a „*Fel-formális*" fejezetben megfogalmazott ReqIF meta-modell és az azzal kompatibilis alkalmazás modellek ekvivalens szemantikájú leírására. Minimális követelmény a tervezett felhasználási kör fenti fejezetben szereplő korlátozását figyelembe vevő modellek fedése, de miután a későbbiekben tervezett a módszerek kibővítése a részletes tervezési specifikációkra is, célszerű a teljes ReqIF modellezési spektrum kezelésére alkalmas matematikai háttér apparátus választása.
2. A modellek formális megfogalmazása mellett az ellenőrzéseket végzendő apparátusnak képesnek kell lennie az *„összes követelmény az összes állapotában mindig teljesül*" kritérium ellenőrzésére. Ez azt jelenti, hogy az ellenőrző apparátusnak az összes szóba jövő példány összes lehetséges megvalósítása illetve összes állapota (állapottal rendelkező rendszerek esetében) felett kimerítő vizsgálatot kell végezni. Ez igényli az univerzális kvantorok bevezetését a kérdéshez kapcsolódva és a kimerítő keresést az ellenőrző eszközzel.

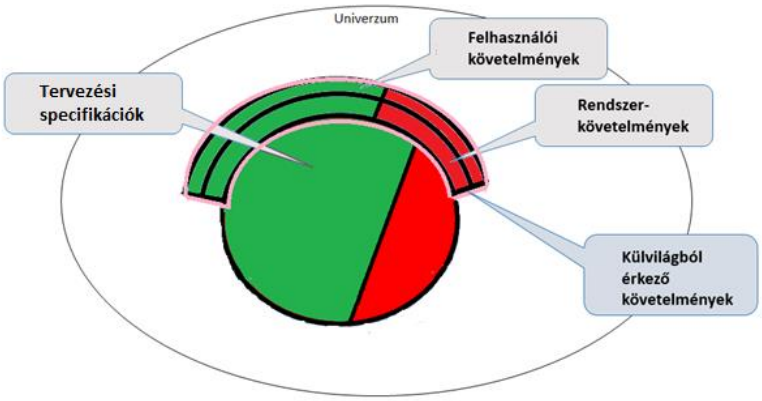
A továbbiakban elsőként a követelménymodellezés matematikai háttérapparátusát vizsgálom, majd a következő pontban röviden megvizsgálom azt, hogy e modell felett a lekérdező-ellenőrző apparátusnak mi a matematikai igénye.

1.3.5.3 A modellező nyelv matematikai alapjai

A követelmények magas szintű formális leírásához a leíró logikák („DL – Description Language") közé tartozó ***ALC*** („Attributive Concept Language with Complements") nyelvet választottam, a következő előnyös tulajdonságai miatt:

- kifejező ereje (azaz az általa használt fogalom-kifejezések - atomi negálás, metszet, értékkorlátozás, egyszerű létezési korlátozás, teljes negálás), amely megegyezik az ALCUE nyelv kifejezőerejével [43] (ALC az unióval, a teljes létezési korlátozással és a teljes negációval kiegészítve) elegendő a fenti „Bevezető" fejezetben informálisan definiált tervezési szintek formális megadásához
- eldönthető
- a nyílt-világ elméleten alapszik (részletesebben lásd „Érzékenységanalízis" fejezetben)
- egyszerű, a halmazelméleten alapszik
- fogalmak („*concept*"), szerepek („*role*"), és példányok („*individuals*") leírására alkalmas
- különválaszthatóak a modell struktúrájára vonatkozó álltások (T-doboz) és a példányokra vonatkozó állítások (A-doboz)

A bevezetőben már ismertetett követelmény értelmezés (12. ábra) formális leírása:



12. ábra Felhasználói követelmények, rendszerkövetelmények és tervezési specifikációk közti kapcsolat

Δ^I= {teljes külvilág}
Univerzum = Δ^I
Külvilágból_érkező_követelmények ⊑ Univerzum
Tervezési_specifikációk ⊑ Univerzum
Felhasználói_követelmények ⊔ Rendszer_követelmények ≡ Külvilágból_érkező_követelmények
Felhasználói _követelmények ⊓ Rendszer_követelmények ≡ ∅
Külvilágból_érkező_követelmények ⊓ Tervezési_specifikációk ≡ ∅
Tervezési_specifikációk ≢ ∅
Külvilágból_érkező_követelmények ≢ ∅
Lehetséges_környezeti_viselkedések ⊔ Kizárt_környezeti_viselkedések ≡
Külvilágból_érkező_követelmények
Lehetséges_környezeti_viselkedések ⊓ Kizárt_környezeti_viselkedések ≡ ∅
Kötelező_rendszer_működések ⊔ Tilos_rendszer_működések ≡ Tervezési_ specifikációk
Kötelező_rendszer_működések ⊓ Tilos_rendszer_működések ≡ ∅

2.3.5.3 Ontológia

Az ontológia egy hierarchikus, tudás reprezentáló adatmodell, amely objektumok és azok között fennálló relációk segítségével írja le az adott tárgykört. Az ontológiával leírt rendszer konzisztenciája különböző bizonyító motorok segítségével (pl. FaCT ++, Pellett reasoner) ellenőrizhető.

A formális modellezés szemléletei egy-egy modellben gyakran keverten jelennek meg. Az ontológiák alkalmasak arra, hogy kevert szemléletű modellt lehessen bennük felvenni. Segítségükkel úgy építhető fel a rendszer egy formális leírása, hogy ehhez nem feltétlenül szükséges a szakterülettel kapcsolatos ismereteken felül adat-analízis szaktudás. Ez alapján célszerű az érzékenységanalízis megvalósítására alkalmas alkalmazás bemeneteként egy ilyen modellt kell megadni.

A leíró logika háttértámogatásának köszönhetően az ontológiával történő fogalomalkotás során más előnyök is léteznek: [13]

- A rendszerrel kapcsolatosan *lekérdezések* fogalmazhatóak meg
- Lehetőség van a hierarchia rendszer *kiterjesztésére* – bizonyos leíró logika alapú módszerek segítségével az egyszerű fogalmakból komplex fogalmak generálhatóak, melyek valósi létét a szakembernek kell validálni

- Az ontológiába való szerevezéskor a szakemberrel szemben támasztott követelmény, hogy a fogalmakat *explicit* módon határozza meg, ezzel segítve az emberek és a gépek számára is a fogalmak pontos interpretálását

1. Az ontológia axiómái

Az ontológia építést sok szoftver támogatja. Ezek közül az egyik a Protegé, amely telepítése és használata is igen egyszerű. A Protegé-ben az építkezés során először felvesszük a fogalmakat és azok hierarchiáját, majd a köztük lévő relációkat definiáljuk, ezt követően pedig egyéb megkötéseket adhatunk meg a fogalmakra és a relációkra vonatkozóan. Ezek képezik az axiómákat. Néhány példa ezen megkötésekre:

1. Fogalmakra:
- *Kizáró fogalmak* – Ha A és B egymást kizáró fogalmak, akkor nem létezhet olyan példányuk, ami egyszerre elégíti mind a kettőt

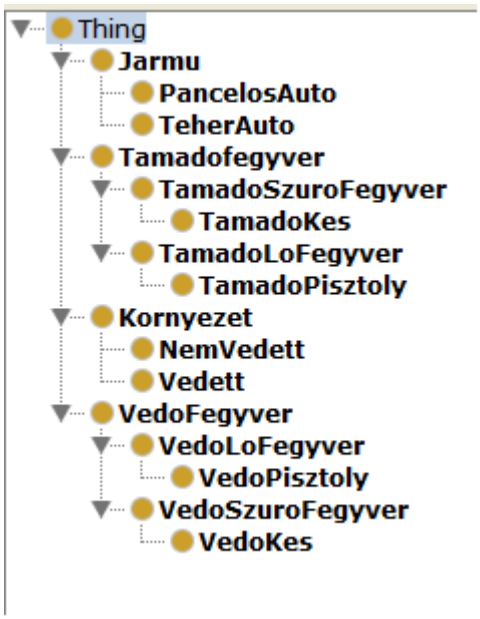
F: fogalmak halmaza és A, B ⊆ F. A és B kizáró fogalmak akkor és csak akkor, ha A ∩ B = { ∅ }.

- *Teljes kizáró lefedés* – Ha A és B teljesen lefedik C-t és A és B kizáró fogalmak, akkor nem létezhet olyan D, amire igaz, hogy C részhalmaza, de nem részhalmaza se A-nak se B-nek
  A, B, C, D ⊆ F és A, B kizáróak. A és B teljesen lefedik C-t akkor és csak akkor, ha ∄ D, amire D ⊆ F és ¬(D ⊆ A) és ¬(D ⊆ B)

2. Relációkra
- *Inverz relációk* – F1 és F2 relációk inverzek, ha az egyik teljesülése maga után vonja a másik teljesülését az ellenkező irányba
  A, B ⊆ F és F1, F2 reláció. F1 és F2 inverz akkor és csak akkor, ha A F1 B <==> B F2 A
- *Funkcionális relációk* – Ha egy fogalom (A) funkcionális relációban (F1) van egy másik fogalommal (B), akkor nem létezhet olyan fogalom (C), amire igaz, hogy A funkcionális relációban (F1) áll C-vel és C != B.
  A, B, C ⊆ F, F1 reláció és A F1 B. F1 funkcionális akkor és csak akkor, ha ∄ C, amire C ⊆ F és A F1 C és C != B.
- *Inverz funkcionális relációk* – Olyan funkcionális relációk, melyek inverzek
- *Tranzitív relációk* – Egy F1 reláció tranzitív, ha A és B valamint B és C fogalmak közötti fennállása maga után vonja A és C közötti fennállását is
  A, B, C ⊆ F és F1 reláció. F1 tranzitív akkor és csak akkor, ha A F1 B és B F1 C <==> A F1 C
- *Szimmetrikus relációk* – Egy F1 reláció szimmetrikus, ha A és B fogalmak közötti fennállása maga után vonja B és A közötti fennállását is
  A, B ⊆ F és F1 reláció. F1 szimmetrikus akkor és csak akkor, ha A F1 B <==> B F1 A
- *Aszimmetrikus reláció* - Egy F1 reláció aszimmetrikus, ha A és B fogalmak közötti fennállása kizárja B és A közötti fennállását
  A, B ⊆ F és F1 reláció. F1 szimmetrikus akkor és csak akkor, ha A F1 B <==> ¬(B F1 A)
- *Reflexív reláció* - Egy F1 reláció reflexív, ha minden A fogalomra igaz, hogy A F1 A
  A ⊆ F és F1 reláció. F1 reflexív akkor és csak akkor, ha A F1 A
- *Irreflexív relációk* - Egy F1 reláció irreflexív, ha egy A fogalomra se igaz, hogy A F1 A
  A ⊆ F és F1 reláció. F1 irreflexív akkor és csak akkor, ha ¬(A F1 A)

2. Ontológia leíró formátum

Az ontológiák leírására számos szabványos nyelv létezik. Az egyik legelterjedtebb az XML alapú OWL („*Web Ontology Language*”), melyet a W3C [13] fejlesztett ki. Ennek a nyelvnek a létrehozása során az ontológiában megadható axiómák hierarchikusan rendszerezték. Az OWL 2 a leíró logika kifejező erejével bír, eldönthető és nyílt- világ-szemléletet használ. (Erről bővebben az „Érzékenységanalízis” fejezetben esik szó). Az alábbiakban arra mutatok egy példát, hogy egy Protegé-ben felépített ontológia axiómái hogyan jelennek meg OWL-ben.



13. ábra Hierarchikus viszonyok

A **13**. ábra mutatja a szállítmányozással kapcsolatos példa (részletesebben lásd „*Példa*” című fejezet) fogalmai között fennálló hierarchiát. A hierarchikus viszony leírására az OWL-ben a „subClassOf” axióma szolgál. Például a „Fegyver” fogalom egyik részfogalma a „Pisztoly”. (14. ábra)

```
<SubClassOf>
   <Class IRI="#Pisztoly"/>
   <Class IRI="#Fegyver"/>
</SubClassOf>
```

14. ábra OWL - subClassOf axióma

Az ontológiák másik fontos előnye, hogy alkalmassá lehet tenni állapot-alapú megközelítés implementálására is. A követelményspecifikáció során használt legalapvetőbb axióma típusok és azok OWL 2-ben történő reprezentációi az alábbi táblázatban láthatóak.

Követelmény Típus	DL	OWL 2 Reprezentáció
implikáció (A ==> B)	$A \sqsubseteq B$	<SubClassOf> <Class IRI="#A"/> <Class IRI="#B"/> </SubClassOf>

A ÉS B = C	$A \sqcap B \equiv C$	<EquivalentClasses> <Class IRI="#C "/> <ObjectIntersectionOf> <Class IRI="#A"/> <Class IRI="#B"/> </ObjectIntersectionOf> </EquivalentClasses>
A VAGY B = C	$A \sqcup B \equiv C$	<EquivalentClasses> <Class IRI="#C"/> <ObjectUnionOf> <Class IRI="#A"/> <Class IRI="#B"/> </ObjectUnionOf> </EquivalentClasses>
Nem A = B	$\neg A \equiv B$	<EquivalentClasses> <Class IRI="#A"/> <ObjectComplementOf> <Class IRI="#B"/> </ObjectComplementOf> </EquivalentClasses>

15. ábra Követelménytípusok reprezentálása OWL 2 axiómákként

4. Érzékenységanalízis

4.1. Cél és definíció

A rendszerkövetelmények specifikálása közben az egyik legfontosabb eljárás a *Lehetséges Hibamód- és hatáselemzés (FMEA – Failure Mode and Effect Analysis)*, amely során a rendszerben lévő lehetséges hibák és okaik azonosítása, valamint hatásuk felmérése a feladat a rendszer megbízhatóságának biztosítása érdekében. A hibaanalízis jelentősége a „Bevezető" fejezetben már bemutatott gazdasági okokra vezethető vissza, ráadásul a követelményspecifikációban szereplő hibák az implementált informatikai rendszerben is jelen lesznek, javítási költségük az idő előrehaladtával egyre nő.

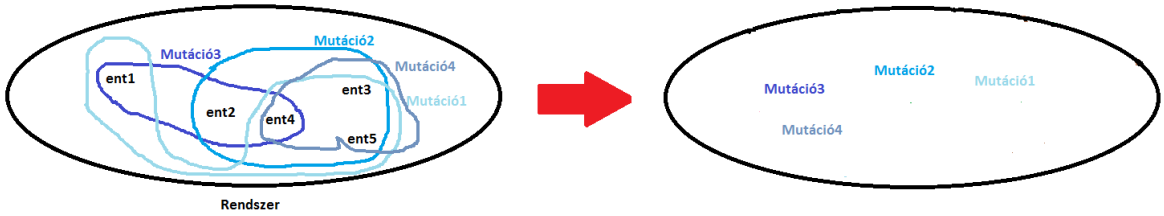
A rendszerben lévő hibák (pl. rendszerkomponensek véletlen vagy szisztematikus hibái) és a követelmény specifikációban lévő hibák (pl. hiányosság, alulspecifikáltság) feltárásának egyik módja az érzékenységanalízis, amely során a rendszer robusztusságát vizsgáljuk, azaz azt a tulajdonságát, hogy mennyire érzékeny a módosításokra és a hibákra. A rendszerspecifikációnak azon szakaszában kell végezni, amikor a követelmények egymással való konzisztenciáját már belátták. A legtöbb eszköz, ami képes valamilyen analízist végezni, semmilyen problémát nem mutat olyan esetben ki, amikor a követelmények egymással nem állnak ellentmondásban. Például egy konzisztensen felvett ontológián egy bizonyító motor (reasoner) a futtatása során semmilyen rendellenességet nem talál. Az ilyen típusú vizsgálatok a külvilág működését ideálisnak feltételezik, a rendszert véges számú bemenet típusra készítik fel.

Azonban léteznek más típusú hibák is. Ezek egyik oka még mindig a tervezésből fakad, tehát magából a rendszerből ered, de nem a konzisztenciára vonatkozik: például (1) szerepelnek benne felesleges megkötések. A hibák létrejöhetnek valamilyen (2) belső módosítás következtében is. A harmadik ok, ha a (3) rendszert a külvilágból olyan hatás éri, amire nincs felkészítve. Tehát három szempontból kell

ellenőrizni: hogyan reagál és mennyire érzékeny a külvilágból érkező ingerekre és belső módosításokra, illetve van-e benne felesleges követelmény.

Az érzékenységanalízis során a végzett vizsgálatok lényege, hogy a rendszerre és a külvilágra vonatkozó követelmények érvényességét valamilyen sorrendben relaxálni kell, így létrehozva a rendszerkülönböző mutációit, melyek segítségével meg lehet állapítani, hogy melyek a megerősítendő pontok, illetve hogy hol szerepelnek túl erős megkötések vagy redundancia.

Pontosításra szorul a *mutáció* fogalma. A modell különböző entitásokból (osztályok, köztük fennálló relációk stb.) és logikai állításokból épül fel, ezeket együttesen nevezzük most entitásoknak. Egy-egy mutáció alatt a rendszer egy olyan változatát értjük, amelyben a fenti entitások közül néhány nem szerepel, de a modell továbbra is értelmes. Egy-egy követelmény kielégítéséhez több entitásra is szükség lehet. (16. ábra)



16. ábra Mutáció-generálás

A megállapított megerősítendő pontok esetében általában a rendszert további megkötésekkel kell kiegészíteni, míg redundancia esetében lehetséges, hogy egy axióma elhagyásával csökkenteni lehet az implementáció költségét anélkül, hogy a követelmének sérülnének. Az elhagyás előtt azonban további manuális vizsgálatokat kell végezni, mert előfordulhat, hogy a redundancia direkt került a rendszerbe a kritikus pontok megerősítése érdekében.

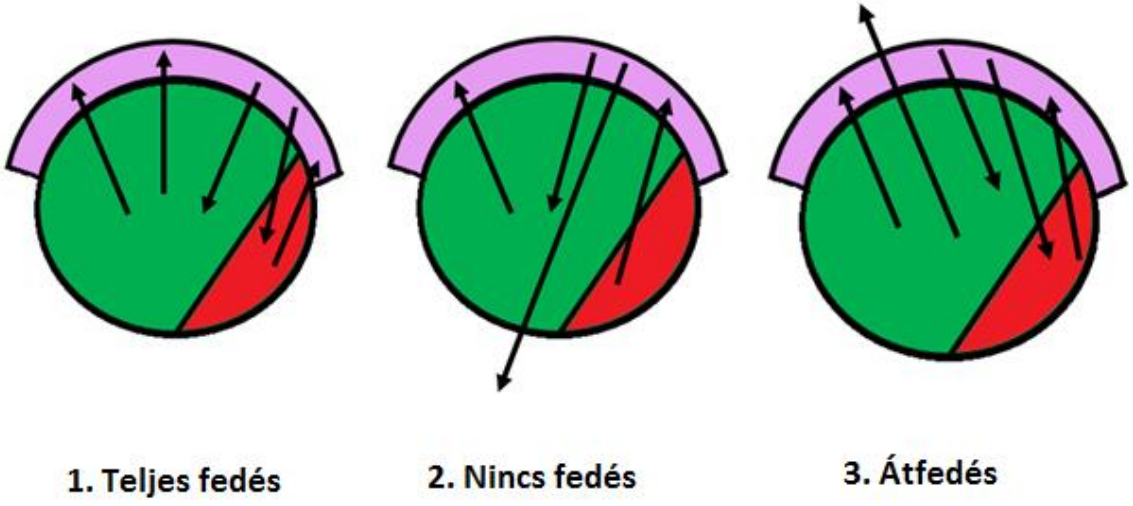
A módszer során nem kapunk egyértelmű választ az olyan jellegű kérdésekre, hogy melyek a redundáns axiómák és hogy a megerősítendő pontokat pontosan hogyan kell megerősíteni. Az érzékenységanalízis azonban hatékonyan mutatja meg, hogy melyek a rendszernek azon kritikus részei, melyekben további vizsgálata esetén lehetséges új típusú hibákat fedezhetünk fel. Egy egyszerű példa erre, amikor az érzékenységanalízis egy zártsági axiómáról kimutatja, hogy kritikus, mert relaxálása után a rendszer többé nem felel meg a felhasználói követelményeknek (pl. biztonságosság). A további vizsgálódás ez esetben arra vonatkozik, hogy e zártsági feltétel milyen valószínűséggel és hogyan sérülhet. Ezen analízisekhez azonban már szükség van a szakterület specifikus ismeretekkel rendelkező szakember bevonására is.

4.2. Lehetséges kimenetek és értékelésük

A „*Rendszerkövetelmények formalizálása*” fejezetben már ismertetett követelmény definíció alapján három lehetséges kimenet jöhet létre. Az analízis eredményének értékelése tehát a rendszer strukturális szintjén történik.

A három alap-esetnek létezhetnek különböző kombinációi is, dolgozatomban azonban ezek kifejtésével nem foglalkozom. Az alábbi ábrákon ezek bemutatása látható.

Az érzékenység analízis lehetséges kimenetei az egyes mutációkra nézve az alábbiak lehetnek (17. ábra):



17. ábra Az érzékenységanalízis lehetséges kimenetei

A lehetséges kimenetekre a „*Példa*” című fejezetben bemutatandó szállítmányozással kapcsolatos példából mutatok részleteket. A fő cél a biztonságos szállítás. A példában a tervezési sajátosságok és a külvilágból érkező követelményekre vonatkozó axiómák különválasztása manuálisan történik.

1. Teljes fedés - Az összes tervezési tulajdonság találkozik a megfelelő külvilágból érkező megkötéssel.

Az első esetben a mutációt követően helyes marad a rendszer. Ebből következően az elhagyott axióma redundánsan szerepelt eddig a rendszerben.

A második és a harmadik esetek elérésére két-két lehetőség van (illetve ezek kombinációja, amennyiben az analízis több axióma azonos idejű relaxációjára kiterjed.):

2. Nincs fedés – A tervezési sajátosságok nem fedik le az összes külvilágból érkező követelményt.

- <u>Külvilágból érkező követelmények bővítése:</u>
  - ami a lehetséges környezeti viselkedések bővítésével:
    Lehetséges_környezeti_viselkedés1 $\equiv$ {„Védett környezet.”}
    Lehetséges_környezeti_viselkedés2 $\equiv$ {„Védett környezet.”, „Nem védett környezet.”}
  - vagy a kizárt környezeti viselkedések szűkítésével állítható elő:
    Kizárt_környezeti_viselkedés1 $\equiv$ {„Nem védett környezet”}
    Kizárt_környezeti_viselkedés2 $\equiv \emptyset$

A $\equiv$ jel használatából következik, hogy a Lehetséges_környezeti_viselkedés1-ben nem szerepelhet más, csak a „Védett környezet.”, azaz hogy a rendszer ebből a szempontból zárt. Az érzékenységanalízis ezen a zártságon változtat azzal, hogy kibővíti az adott osztályokat. Ebben az esetben előfordulhat, hogy a szállítás nem lesz biztonságos, hiszen létezik olyan eset, amikor a rendszer „Védett környezet” esetén biztonságos, „Nem védett környezet” esetén viszont nem az. Amennyiben az érzékenységanalízis során a fenti esetekre a rendszer nem működik megfelelően (például az analizáló eszköz ki tudja mutatni, hogy valamelyik felhasználói követelmény nem teljesül), akkor előfordulhat, hogy további axiómákkal kell kibővíteni.

- <u>A rendszer működésének csökkentése:</u>
  - ami a rendszer kötelező működésének szűkítésével:
    Kötelező_működés1 $\equiv$ {„Teherautó.”, „Páncélos autó.”}

Kötelező_működés2 ≡ {„Teherautó.”}

- vagy a rendszer tiltott működéseknek növelésével állítható elő:
  Tilos_működés1 ≡ ∅
  Tilos_működés2 ≡ {„Tilos nem biztonságos parkolóban parkolni”}

3. Átfedés – A tervezési sajátosságok túllógnak a külvilágból érkező követelményeken**.** (Azaz a rendszer nem használja ki a tervezési könnyítéseket, vagy többet tud az elvártnál.)

- Külvilágból érkező követelmények csökkentése:
  - ami a lehetséges környezeti viselkedés szűkítésével:
    Lehetséges_környezeti_viselkedés1 ≡ {„Védett környezet.”, „Nem védett környezet.”}
    Lehetséges_környezeti_viselkedés2 ≡ {„Védett környezet.”}
  - vagy a kizárt környezeti viselkedések bővítésével állítható elő:
    Kizárt_környezeti_viselkedés1 ≡ ∅
    Kizárt_környezeti_viselkedés2 ≡ {„Nem védett környezet.”}
- A rendszer működésének bővítése:
  - ami a kötelező rendszer működések bővítésével:
    Kötelező_működés1 ≡ {„Páncélos autó.”}
    Kötelező_működés2 ≡ {„Teherautó.”, „Páncélos autó.”}
  - vagy a tilos rendszer működések csökkentésével állítható elő:
    Tilos_működés1 ≡ {„Tilos nem biztonságos parkolóban parkolni”}
    Tilos_működés2 ≡ ∅

A 18. ábra táblázata a fenti négy hibás kimenet értékelésének módját foglalja össze:

		Rendszer helyesen működik tovább	**Rendszer nem működik helyesen tovább**
Nincs fedés	Külvilágból érkező követelmények bővítése	Redundancia esete állhat fenn, hiszen több külvilági követelménynek felel meg, mint az elvárt minimum.	Meg kell vizsgálni, hogy létrejöhet-e ilyen típusú bővülés a valóságban, mert ha igen, az problémát okozhat.
	A rendszer működésének csökkentése	Redundancia esete állhat fenn, hiszen csökkentett működéssel is ki tudja elégíteni a rendszer az összes külvilágból érkező követelményt.	Meg kell vizsgálni, hogy fel tud-e lépni olyan hiba, ami miatt az adott rendszerműködés nem teljesül, mert ha igen, az problémát okozhat.
Átfedés	Külvilágból érkező követelmények csökkentése	Meg kell vizsgálni, hogy fel kell-e egyeltalán készülni a relaxált külvilágból érkező követelményre, hiszen a külön felkészülés nélkül is helyes a működés. (Redundancia esete állhat fenn)	Nem fordulhat elő, mert a lehetséges külvilágból érkező követelmények továbbra is ki lesznek elégítve.
	A rendszer működésének bővítése	Nincs információ tartalma.	

18. ábra Az ÉA lehetséges kimeneteinek értékelése

4.3. Nyíltvilág- és zártvilág-szemantika

Az érzékenységanalízis során fontos kérdés, hogy a rendszer leírása nyílt- vagy zártvilág-szemantikában történt-e, hiszen ettől függ, hogy a különböző mutációkat hogyan kell előállítani.

A zártvilág-szemantika (*CWA – „Closed-World Assumption*”) szerint egy állításról akkor és csak akkor feltételezhetem, hogy igaz, ha be tudom bizonyítani. Ennek következtében minden más állítás hamis.

Általában az olyan adatbázisok esetén használnak CWA-t, amikor teljes tudás áll rendelkezésre. Az alábbi példaadatbázisban a „Véd-e a golyóállómellény minden fegyver ellen?” kérdésre a válasz nemleges lenne.

Fegyver	Védelem
Kés	Golyóállómellény
Pisztoly	Golyóállómellény
Páncéltörő	

A nyíltvilág (*OWA – „Open-World Assumption*”) filozófiája szerint egy állításról egészen addig nem feltételezhetjük, hogy hamis, amíg biztosan ki nem derül, hogy nem igaz. Tehát csak a már tudottan igaz állításokból tudunk következtetéseket levonni. Ha egy állításról nem tudjuk bebizonyítani, hogy igaz, akkor ebből nem következik, hogy hamis. Tehát létezik egy köztes állapot, amit a nem kikövetkeztethető állításokhoz lehet kötni: „nem tudom”. Ennek egyik előnye, hogy monoton („*monotonic*”) logikához vezet, azaz ha a rendszert új axiómával bővítjük, akkor az eddigi összes tudásunk igazságtartalma változatlan marad.

Egy ilyen világban az állítás, hogy „A következő fegyverek léteznek: kés és pisztoly.” nem vonja maga után automatikusan azt, hogy nem létezik egy harmadik fajta fegyver (pl. Páncéltörőtörő) is. A fenti adatbázisos példakérdésre a válasz az lenne, hogy „nem tudom”.

Az leíró logika (és emiatt az OWL 2 is) az OWA szemlélten alapszik, ezért sokszor explicit módon be kell venni a rendszerbe a zártságra vonatkozó axiómákat. Ezek kritikusak a biztonság szempontjából. Amikor a követelményeket megadjuk, akkor általában zártvilágban gondolkodunk. Az érzékenységanalízis során az implementációban a legtöbbször pont az OWA miatt explicit leírt zártságra vonatkozó axiómák relaxálását kihasználom ki, mert így bővíthető a lehetséges környezeti viselkedések és a kötelező rendszer működések köre.

4.4. Új ötlet: kapcsolók

A cél az, hogy a rendszer különböző mutációit hozzuk létre, melyekben megváltozik az eredetiben szereplő axiómák érvényessége. Ennek érdekében be lehet vezetni minden axiómához egy kétállású kapcsolót, ami lehetővé teszi az érvényességük beállítását.

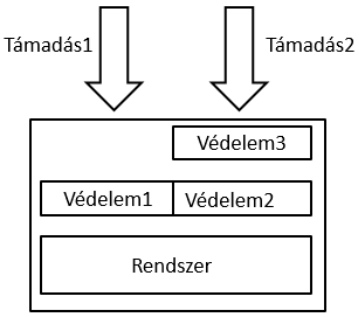
Minden axiómát úgy kell megfogalmazni, hogy teljesülésének feltételei eldönthetőek legyenek. Például egy a hőmérsékletre való megkötésben valamilyen intervallumba-esés legyen követelményként meghatározva (pl. egy rendszer helyes működése 0 és 50 C˚ között legyen garantált). Ezután minden axiómához rendelhető egy, az érvényességére vonatkozó boolean változó, mint kapcsoló.

Az analízis során az adott analizáló dekleratív eszköz a rendszerről feltett kérdésekre ad válaszokat. Ha például ismertek a biztonságosság feltételei, akkor a biztonságosságra vonatkozó kérdés eldöntése során megvizsgálja, hogy az összes feltétel teljesül-e a modellt kielégítő lehetséges példányokban. Ha a kérdések megfogalmazásában plusz feltételként megkötik, hogy a kapcsolók közül pontosan egynek hamis állásúnak kell lennie, akkor az eszköz sorra veszi az összes olyan rendszer-mutációt, amiben az

eredeti axiómák közül egy nincs benne. Az összes ilyen mutációról megpróbálja belátni, hogy teljesülnek-e rá a kérdésben megfogalmazott feltételek. A lehetséges kimeneteket és azok értelmezéseit a „*Példa és értékelés*” szakasz már kifejtette.

A fenti megoldás alapján két konklúziót lehet levonni:

(1.) A módszer olyan mutációk generálására alkalmas, amelyben egyszerre csak egy axióma érvényessége kerül kikapcsolt állapotba. Ez azért nagy hátrány, mert így nem lehet a köztük lévő összefüggéseket vizsgálni. Például előfordulhat, hogy egy rendszerben egy axióma kikapcsolása esetén csak az egy kitüntetett axióma kikapcsolása kritikus, azonban kettő kikapcsolása esetén már bármelyik axiómapár kiesése nem biztonságos állapothoz vezet.



	Védelem1	Védelem2	Védelem3	Védett?
Ideális eset, nincs kikapcsolva egy axióma sem	+	+	+	+
Egyszerre csak egy axióma van kikapcsolva: csak a Védelem1 kritikus	+	+	-	+
	+	-	+	+
	-	+	+	-
Egyszerre két axióma kikapcsolása: bármelyik két axióma közös kikapcsolása kritkus	+	-	-	-
	-	+	-	-
	-	-	+	-
	-	-	-	-

(2.) A módszer implementálásához az analizálást végző eszközzel szemben vannak bizonyos követelmények:

- Legyen deklaratív – valamit leírok és a meghívási logika el van rejtve (ilyen modell esetében mindegy a sorrend, nem számít, hogy mit kapcsolunk ki)
- Legyen képes a hibamodell implementálásakor a mutációt hasonlóan lokálisan generálni, mint ahogy az eredeti hibamodell is eredeti modell elemeket relaxál. Ezt a mechanizmust, miután a hibákat a meta-modell vagy meta-meta-modell felett definiáljuk, hasonló módon automatikusan lehessen a mutációkat a hibamodell ismeretében az eredetiből származtatni
- A hibamodellben a hibák típusán túlmenően szokás további megszorításokat is tenni, például a hibák számára vonatkozóan. Ennek hátterében az a meggondolás állhat például, hogy a hibák előfordulási valószínűsége elég kicsi ahhoz, hogy a többszörös hibák fellépte független hiba-előfordulást feltételezve elhanyagolható legyen.
- Legyen képes számosságot kezelni (Ki lehessen vele fejezni azt, hogy „pontosan egy”)

Az implementáláshoz választott eszközről, az Alloy-ról az „*Analízis feladatok és eszközök*” című fejezetben esik szó.

4.5. Mik az érzékenységanalízis korlátai?

Az érzékenységanalízist megvalósító alkalmazás természetesen bizonyos korlátokkal rendelkezik, melyek az alábbiak:

- Arra mutat rá, hogy milyen irányban érdemes elkezdeni a lehetséges hibák keresését, az eredménye sosem egy kategorikus hiba megállapítás. Segítségével egy rendszerre vonatkozó követelmény (pl. biztonságosság) vizsgálható olyan körülmények figyelembevételével, melyekre eddig nem volt felkészítve.
- Eredményének kiértékelése nem automatikus, ami megnehezíti az értelmezést.

- Minden típusú axióma (külvilágból érkező követelmény és rendszerrel szembeni elvárások) kezelése azonos módon történik. Ez szintén a kiértékelést nehezíti, hiszen a második pontban bemutatott értékelési módszerekhez ezeket külön kellene választani.
- További vizsgálatok lebonyolításához szakemberre van szükség. Nem nyújt segítséget arra nézve, hogy hogyan végezzük el további analízist, csak hogy hol végezzük.
- Jelenleg egyszerre csak egy axióma érvényességét lehet relaxálni. Így az axiómák közötti összefüggések nem vizsgálhatóak.
- A következő fejezetben bemutatandó formális fogalmi analízis segítségével előállítható komplexebb mutációk automatikus generálásával egyszerre több axiómát lehetne kikapcsolni. Ez azonban jelenleg ez nincs még kidolgozva.
- Jelenleg nem képes olyan típusú kérdések megválaszolására, hogy legfeljebb hány axióma sérülése esetén marad a rendszer továbbra is biztonságos. Ennek implementálásához is komplexebb mutációk létrehozására lenne szükség.
- Nem képes a lehetséges követelményspecifikációban fellépő hiányosságok detektálására.

4.6. Felhasznált modellező eszközök

A modell alapú analízisnél és helyességbizonyításnál kulcskérdés, hogy a modellezésre és vizsgálatokra felhasznált leíró és algoritmikus nyelvek egyértelműek legyenek, máskülönben nem biztosított a leírni szándékozott és leírt modell szemantikus azonossága, azaz eltérés esetén a kapott eredményt nemcsak irreleváns lehet a konkrét probléma szempontjából, hanem még félrevezető is.

Az általam kialakított vizsgálati módszer ezért két olyan modellező paradigmát ötvöz, amelyeknek létezik bizonyítottan helyes matematikailag is ellenőrzött és csak egyértelműen interpretálható szemantikája. Ezek közül az egyik a modellezésre használt ontológia (amelynek szemantikáját az ISO „*Common Logic*” segítségével is leírták [51]), a másik pedig kérdések feltevésére alkalmas Alloy, ugyancsak publikált és ellenőrzött szemantikával [52].

Az előző megfontolásaim szerint a korlátozott követelménymodell leírásánál nem volt szükség még az ontológiák teljes kifejező erejére sem, hiszen a strukturális követelménymodellek és érzékenységük elemzésére modellezési oldalon elégséges a halmazok feletti relációalgebra (lásd III.5.3 – „*Formális*” fejezet). Az ellenőrzendő tulajdonság megfogalmazásához és vizsgálatához a fentiek szerint elsőrendű logika az igény.

Az ontológia alapvetően a leíró logikára épül. Egyik fő előnye az, hogy van szabatos, matematikailag definiált szemantikája [53]. A dolgozatban ennek a szemantikának egy részhalmazát használom csak, ennek megfelelően a matematikai notációt e részhalmazoknak megfelelően egyszerűsítettem, anélkül, hogy ezzel az az eredeti szemantikától eltérne.

Hasonló módon a kérdések megfogalmazására is használt Alloy nyelv is rendelkezik formális szemantikával. Az innen felhasznált megoldásokat nem formalizáltam külön, hiszen az az Alloy-nak a tiszta szerkezete miatt lényegében csak egy mechanikus átírásnak felel meg.

5. Formális fogalmi analízis

A „Bevezetőben” említettek szerint a rendszerkövetelmények hierarchizálása javítja a követelményrendszer illetve specifikáció áttekinthetőségét és az egyes elemek újrafelhasználhatóságát, így jelentősen javíthatja a tervezési folyamat hatékonyságát és minőségét, csökkentve ezzel a költségeket.

A követelményrendszer kidolgozása során alapvető modell-kidolgozási technika a fokozatos finomítás. Ez azonban egy-egy követelményt dolgoz fel, így a kialakult részletes követelményhalmaz meglehetősen strukturálatlan.

Egy hatékony rendszer esetében azonban a hasonló jellegű követelményekre célszerű alapjában közös és az egyes esetekre újraalkalmazott megoldásokat keresni. Ez a „Bevezetőben" említett okokra vezethető vissza.

Az ismertetendő formális fogalmi analízis (FCA – „Formal Concept Analysis") segítségével a követelmények közti hasonlóságok alapján lehet őket azonos kategóriákba sorolni. Az FCA módszere:

- egy olyan alulról felfelé építkező követelménymodell meghatározó algoritmus, amely a pontosan ugyanazoknak a tulajdonságoknak a teljesülését megkívánó követelmények együttes kezeléséhez egy közös absztrakt fogalmat vezet be, valamint
- egy fogalmi hierarchiát alakít ki aszerint, hogy hány tulajdonság egyezését követeljük meg. Ennek segítségével természetes módon rendezhetőek el a követelmények.

5.1. Mi az FCA?

Szabatosabban a formális fogalmi analízis egy olyan eljárás, amely során objektumok és a rájuk jellemző tulajdonságok csoportosításával strukturált fogalmakat lehet képezni.

Minden egyes objektumhoz adott egy tulajdonsághalmazon belül az általa teljesített tulajdonságok halmaza. A létrehozott fogalmak egy objektumhalmazból („*E - extent*") és egy tulajdonsághalmazból („*I - intent*") épülnek fel.

Az objektumhalmazban szereplő összes objektum rendelkezik a tulajdonsághalmazban lévő összes tulajdonsággal és a tulajdonsághalmazban nem szerepel olyan tulajdonság, ami nincs az összes objektumhoz rendelve. Ezek alapján a fogalmak között hierarchia állítható fel: egy részfogalom a hierarchiában felette lévő fogalom objektumhalmazának egy részhalmazát foglalja magában.

A részfogalom reláció megfelel a parciális rendezéshez szükséges relációkkal szemben támasztott elvárásoknak [22], így lehetségessé válik egy matematikai háló felépítése („*lattice*"), amit FCA esetében fogalmi hálónak neveznek. Ez megjeleníthető grafikusan, ami lehetővé teszi az adatok könnyebb elemzését.

Az alábbi példában a támadás típusok az objektumok, és a védekezési módok a tulajdonságok. Ha egy támadás és védekezési mód metszetében szerepel „X", akkor az adott védelem elégséges a támadás elhárításához. A példa az FCA Extension for Excel eszközzel készült. [23] (19. ábra)

Context attributes header				
ObjectID	Object Name	Golyóállómellény	PáncélosSzállító	Biztonságiőr
1	Kés	x		x
2	Pisztoly	x		x
3	Gépfegyver	x		
4	Páncéltörő		x	
5	BeépítettEmber			x

19. ábra Formális kontextus

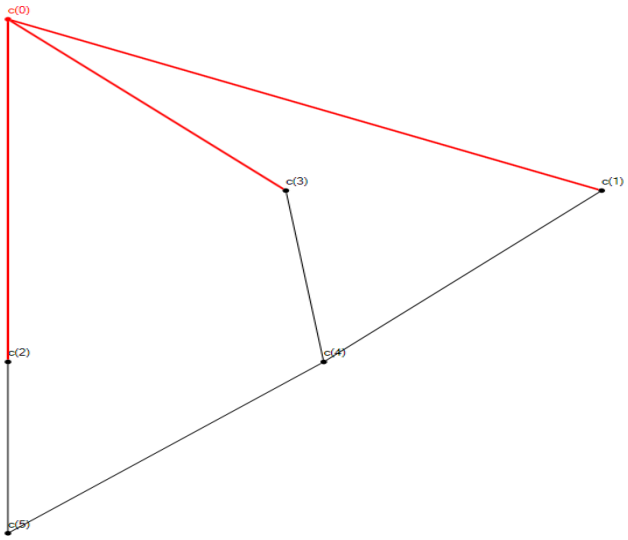
Az FCA Extension for Excel a kontextus megadását követően képes kiszámítani a lehetséges fogalmakat (20. ábra) és megrajzolni hozzájuk a matematikai hálót (21. ábra). A kijelölt fogalmat a hálóban is kijelöli:

All concepts of conceptual lattice

ConceptID	Extent	Intent	ID
c(0)	{Kés; Pisztoly; Gépfegyver; Páncéltörő; BeépítettEmber}	{}	3
c(1)	{Kés; Pisztoly; BeépítettEmber}	{Biztonságiőr}	4
c(2)	{Páncéltörő}	{PáncélosSzállító}	5
c(3)	{Kés; Pisztoly; Gépfegyver}	{Golyóállómellény}	6
c(4)	{Kés; Pisztoly}	{Golyóállómellény; Biztonságiőr}	7
c(5)	{}	{Golyóállómellény; PáncélosSzállító; Biztonságiőr}	8

20. ábra Lehetséges fogalmak

Ebből azt a következtetést lehet levonni, hogy amennyiben az a cél, hogy az összes támadás ellen szerepeljen védelem a rendszerben, akkor meg kell vizsgálni a hálóban legfelül elhelyezkedő (c(0)) fogalmat. Mivel az ehhez tartozó intent halmaz üres, ezért tovább kell haladni a hálóban a c(0)-ból kivezető vonalak mentén.



21. ábra Matematikai háló ("lattice")

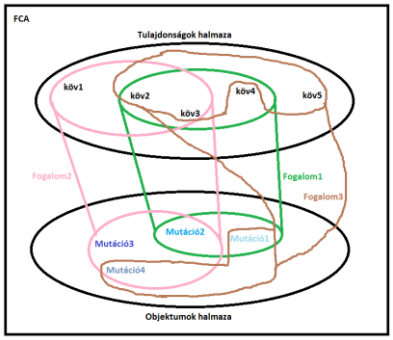
Ezek a c(1), c(2), c(3) pontokba vezetnek. Ezek extent halmazainak uniója lefedi a teljes objektumhalmazt, hiszen c(0) extentje a teljes objektumhalmaz. Mivel ezek intentjeink uniója kiadja a teljes tulajdonsághalmazt, ezért a cél elérése érdekében az összes védelemre szükség van.

5.2. Az FCA használata az érzékenységanalízisben

A rendszerkövetelmények szempontjából három lehetséges felhasználási területe létezik:

- ***Fogalmak formalizálása**:* Az FCA segítségével felállítható a rendszerben szereplő entitások között egy fogalmi hierarchia. Egy általános fogalom alá tartozó fogalmakról elmondható, hogy amennyiben a rendszer az érzékenységanalízis során az általános fogalommal szemben érzékenynek bizonyult, akkor a csoportba tartozó összes fogalommal szemben is az lesz. Ennek a módszernek a másik iránya is használható: absztrakt osztályokat lehet bevezetni, melyekre a védelem közös.
- Az érzékenységanalízis eredményének kiértékelése során az FCA-t a különböző ***mutációk csoportosítására*** is lehet használni a bennük fellelt *hibajelenségek* alapján. Ezt úgy valósíthatjuk meg, ha az FCA *tulajdonsághalmazának* a rendszer követelményhalmazát feleltetjük meg, az *objektumhalmazba* pedig a rendszer mutációi kerülnek. Minden egyes mutációhoz teljesülő tulajdonságként azokat a követelményeket rendeljük, amelyeknek a mutáció által reprezentált rendszer továbbra is megfelel (). Speciális eset a hibátlan rendszer, amelyben az összes követelmény, így az összes tulajdonság teljesül. Az FCA egy *fogalomba* vonja össze azokat a mutáció-halmazoka, amelyekre azonos követelmények (tulajdonságok) teljesülnek (**22**. ábra).

Példa a mutációk csoportosítására



22. ábra Kontextus - grafikusan

A 22. ábra-n szereplő összefüggéseket (az FCA kontextusát) a 23. ábra írja le táblázatosan. Ezt a gyakorlatban a felhasználónak kell kitöltenie: sorra kell vizsgálni az összes mutációt és figyelni, hogy az adott mutáció esetében mely követelmények teljesülnek továbbra is.

Context attributes header						
ObjectID	Object Name	köv1	köv2	köv3	köv4	köv5
1	Mutáció1		x	x	x	x
2	Mutáció2	x	x	x	x	
3	Mutáció3	x	x	x		
4	Mutáció4	x	x	x		x

23. ábra Kontextus - táblázatosan

Ezt követően az FCA segítségével ki lehet generáltatni a fogalmakat és a fogalmi hálót. ()

Tegyük fel, hogy a követelmények között nem szerepel enyhítő feltétel. Ez esetben az összes követelményt teljesítő mutáció(k) a háló legalján fog(nak) elhelyezkedni.

Az ábrán látható példában ilyen mutáció nem létezik, mert egy olyan csökkentett entitáshalmaz sincs (piros keret), amelyik továbbra is kielégítené az összes követelményt. Ezt már a kontextus kitöltése során is észre lehet venni, hiszen nincs olyan sor, amelynek az összes oszlopában szerepel X. (Abban az esetben, ha a fenti táblázat egy érzékenységi analízis kimeneti eredményét foglalná össze, ez annyit

jelentene, hogy valamennyi beinjektált hiba hatásos lenne. Így például a rendszer nem lenne redundáns.)

A háló csomópontjaiban (a fogalmak) azok a mutációk lesznek, amik azonos követelményekre reagálnak rosszul. (Követelmények szempontjából az azonos fogalomba tartozó mutációk azonosak lesznek.)

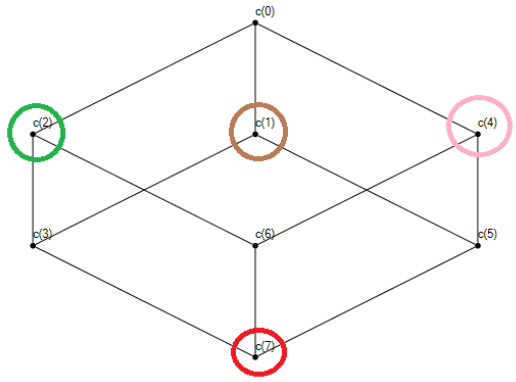
A mutáció-csoportokat elrendező hálóban „alulról felfelé” haladva a kielégített követelmények száma lépésenként eggyel csökken.

Ezek alapján a követelményekről különböző megállapítások tehetőek:

- A leggyengébb követelmények a legfelső fogalomban vannak, mert ezek teljesülését az összes mutáció garantálja.
- A jó rendszert reprezentáló legalsó pont és a legfelső pont közötti a hálóban vezető leghosszabb út hossza meghatározza azon követelmények maximális számát, amennyire egy feltételezett hiba hatással van. (Ebből a követelmény stabilitás index (RSI) becsülhető).

All concepts of conceptual lattice

ConceptID	Extent	Intent
c(0)	{Mutáció1; Mutáció2; Mutáció3; Mutáció4}	{köv2; köv3}
c(1)	{Mutáció1; Mutáció4}	{köv2; köv3; köv5}
c(2)	{Mutáció1; Mutáció2}	{köv2; köv3; köv4}
c(3)	{Mutáció1}	{köv2; köv3; köv4; köv5}
c(4)	{Mutáció2; Mutáció3; Mutáció4}	{köv1; köv2; köv3}
c(5)	{Mutáció4}	{köv1; köv2; köv3; köv5}
c(6)	{Mutáció2}	{köv1; köv2; köv3; köv4}
c(7)	{}	{köv1; köv2; köv3; köv4; köv5}



24. ábra Fogalmak és matematikai háló

6. Analízis feladatok és eszközök

6.1. Tipikus követelmény analízis feladatok

A követelmények specifikálása során a legfontosabb feladat a rendszerrel szemben támasztott folyamatosan változó követelményeket azonosítása és csoportosítása. Az egész folyamatot és a kimeneteként keletkező követelményeket dokumentálni kell. A folyamat során vizsgálni kell a rendszer logikai **helyességét**, azaz a *konzisztenciáját*, *teljességét* és *egyértelműségét*. A másik fontos analizálandó aspektus a **szolgáltatásbiztonság**, amibe beletartozik a változásokkal szembeni érzékenységanalízis is.

Ahogy azt a „*Bevezető*” fejezet már kifejtette, e vizsgálatok jelentősége hatalmas az implementálandó szoftver minősége és költsége szempontjából. Ugyanakkor azt is tisztázta, hogy ez egy igen összetett folyamat, amely során rengeteg kihívást kell legyőzni. Az általános eljárás szerint a követelményspecifikáció iteratív módon készül el: az adott szakterület specialistája által készített modellen egy tőle független szakember végez különböző analíziseket, melyek eredménye alapján további finomításokat végeznek el a modellen.

Első lépésben a szoftverfejlesztési projekttel kapcsolatban négy legfontosabb jellemzőjét kell definiálni. Az első az elérendő *célokra* vonatkozik, azaz annak tisztázása, hogy miért van szükség rá. Ezt követi a *funkcionális* feladatainak azonosítása, azaz hogy milyen szolgáltatásokat kell nyújtania. A harmadik a *megkötésekre* vonatkozik, azaz hogy milyen kereteken belül valósulhat meg. Ezek a

megkötések általában a projekt vállalton belül elfoglalt pozíciójából fakadnak, annak függvényében alakulnak ki, hogy hogyan illeszkedik a vállalti stratégiához. Általában költségekre, időkorlátokra és erőforrásokra vonatkozik. Az utolsó a *felelősségek* azonosítása: melyik SW komponensnek és melyik emberi erőforrásnak mi lesz a feladata.

6.2. Mit nem tud az ontológia és a hozzá tartozó bizonyító motorok?

Az ontológia építés a szakterülettel kapcsolatos fogalmak összegyűjtésére és a köztük lévő relációk és hierarchikus összefüggések ábrázolására alkalmas módszer, amelyet a leíró logikán (DL) alapuló OWL nyelv segítségével lehet megadni. Alapvető elemei az osztályok és a köztük lévő szabadon megadható tulajdonság relációk, melyek a leíró logikában a fogalmaknak és a szerepeknek feleltethetőek meg.

Mivel kifejezőereje egyezik a DL kifejező erejével, ezért csak az ilyen szintű lekérdezések megfogalmazására alkalmas. Ezen a korláton belül képes a konzisztencia globális ellenőrzésére, de nem lehet célzottan megfogalmazni például egy biztonságosság kritériumaira vonatkozó specifikus kérdést, gyakorlatilag csak a kielégíthetőséget vizsgálja.

Amennyiben inkonzisztensnek találja a rendszert, akkor sem képes olyan ellenpélda példány generálására, amelyre igaz, hogy ha a rendszerből elvennénk egy axiómát, még konzisztens lenne, de az elvett axióma miatt válik inkonzisztensé.

A segítségével csak a teljes rendszer vizsgálható, nem lehet egy axióma irányított analízisét megvalósítani.

6.3. Igény

Az „*Érzékenységanalízis*” című fejezet az implementáláshoz használandó eszközzel szembeni követelmények egy részét már tisztázta. Ezeken felül azonban léteznek még továbbiak is, melyek alapvetően a feladat azon sajátosságaiból erednek, hogy a vizsgálat halmazokon és azok közötti relációkon történik, a rendszerek állapotai és elemszámai korlátosak, működése nem előre determinált, működése során konkurens funkciók léphetnek fel, leírásához és a rá vonatkozó kérdések feltevéséhez is legalább első rendű logika szükséges.

A modellezés során a rendszer elemei halmazokként jelennek meg. Ezen halmazok közti hierarchia kifejezésére is képesnek kell lennie a kiválasztott eszköznek, hiszen a megvalósítandó rendszer elemei között fennálló hierarchiából számos következtetés vonható le. Ezek elengedhetetlenül fontosak az érzékenységanalízis szempontjából.

Mivel olyan rendszereket elemzünk, melyek elemszáma és állapota korlátos, ezért nem igény az extrém hosszú szekvenciák vizsgálata. Ez azért fontos, mert ennek köszönhetően elegendő az analizálást csak kisebb szekvenciákon elvégezni, azaz véges automaták kezelésére alkalmas eszközt keresünk. Az ilyen típusú vizsgálatokhoz sokféle eszköz áll rendelkezésre.

A rendszertervezés Knublauch—féle folyamatában az érzékenység analízis megvalósítása az utolsó fázisban van, tehát a rendszer fogalmairól már felvették a taxonómiát, a strukturált követelményrendszer és az ontológia is rendelkezésünkre áll. Ahhoz, hogy bizonyos kérdéseket lehessen feltenni, szükséges az ontológiában használt leíró logikánál erősebb logika, az első-rendű logika.

A fenti követelményeket az alábbi listában tételesen összegyűjtöm:

1. Legyen képes hierarchikus halmazok és azok közti relációk kezelésére
2. Legyen képes nem determinisztikus véges állapot automaták kezelésére
3. Legyen képes konkurencia kezelésére

4. Legyen képes elsőrendű logika kezelésére

6.4. Alloy – a kiválasztott analízis eszköz

Az ontológia szerkesztő eszközök a fenti kritériumok közül már nem mindenek felelnek meg, azonban az MIT által fejlesztett Alloy megfelel, ezt az alábbiakban bizonyítom be.

6.4.1. Alapok

Az Alloy egy modellező nyelv, mellyel szoftver rendszerek írhatóak le. A segítségével dekleratív modellek elkészítése lehetséges, melyekben komplex strukturális megkötések adhatóak meg. Az Alloy-ban megfogalmazott specifikációk analizáláshoz az Alloy Analyzer-t szolgál. Ennek működése verziótól függően valamilyen modell kereső megoldón („*solver*”) alapszik. Képes rá, hogy amennyiben a modell kielégíthető, akkor ad rá egy példát vagy egy adott követelmény nem teljesülésére ad ellenpéldát.

A matematikai alapjai a Z nyelvből fakadnak. Ennek köszönhetően a műveleteket halmazokon és azok közti relációkon értelmezi, és képes köztük lévő hierarchia kezelésére is. [1] A hierarchia kezelésének módjáról részletesebben a modell építés fejezetben írok. A követelmények megfogalmazása során első rendű logikát használ. [4]

Egyszerű szintaxisa az Obect Constraint Language (OCL) szintaxisához hasonló. Mivel támogatott benne az objektum orientált szemlélet is, ezért képes komplex adatstruktúrák ábrázolására. Ebből fakadóan állapotok leírására is alkalmas [2]

A modell építése során kétféle hiba léphet fel. Az egyikben a modell felépítése rossz, azaz valamilyen olyan hibát teszünk a modellbe, amitől nem pontosan azt írjuk le a rendszerről, amit szerettünk volna. Ennek két tipikus esete az aluldeterminálás („*underconstraining*”) és a túldeterminálás („*oerconstraining*”). Az aluldeterminálás kiszűrése a modell inkrementális fejlesztése során a modell aktuális állapotára adott példa vizsgálatával lehetséges. Amennyiben ennek esete áll fenn, valamilyen új követelményt kell felírni. A túldeterminálás észlelése csak extrém esetben könnyű. Ilyenkor önmagában ellentmondásos a specifikáció, és az Alloy Analyzer nem ad vissza példát a modellre. Ennek ellenére, ilyen esetekben nem állítja, hogy a rendszer inkonzisztens. Ez azért nagyon veszélyes, mert amikor a vizsgálunk egy olyan követelményt, ami a rendszerben nem axiómaként szerepel, akkor annak ellenére, hogy ez a követelmény nem köt, az Alloy Analyzer nem lesz képes ellenpélda visszaadására. A másik típusú hiba, amikor a modell felépítése helyes, azonban valamilyen megkötés nem teljesül rá. A modellépítés célja az ilyen hibák és okuk megtalálása, ezek elhárítása, ezáltal a modell javítása.

6.4.2. Az Alloy Analyzer tulajdonságai

Az Alloy Analyzer működése mindig *helyes*, azaz sosem ad vissza rossz eredményt. Amennyiben azt mondja, hogy egy modell kielégíthető, akkor vissza ad egy konkrét példát. Ennek a példának a helyessége garantált.

Azonban a működés *nem teljes*. Ez azt jelenti, hogy nem képes végtelen mennyiségű példát kipróbálni. A felhasználó feladata, hogy amikor arra kéri az Analyzer-t, hogy egy olyan követelmény teljesülését ellenőrizze, ami nem axiómaként van definiálva, akkor meg kell szabnia egy scope-ot, melyen belül az Analyzer dolgozik. Ez azt jelenti, hogy minden a rendszerben leírt halmaznak a scope méreténél kisebb lesz a példányszáma. A scope méretén belül a teljesség garantált. Ha tehát a fenti követelmény nem teljesül az adott scope-on belül, akkor garantáltan megtalálja az összes ellenpéldákat. A megtalálás sorrendje nem rögzített, lehetséges, hogy nem a legkisebb ellenpéldát találja meg először. Ez esetben a követelmény teljesülése nem csak a scope-on belül, hanem azon kívül is kizárt. Ennek köszönhetően az Alloy képes nem determinisztikus véges állapot automaták kezelésére. [2]

Ha azonban a scope-on belül a követelmény köt, akkor az Analyzer csak annyit állít, hogy nagy valószínűséggel a scope-on kívül is igaz az állítás. Ez a fél-döntés („*semidecision*”) típusú bizonyítások tipikus esete, azaz a bizonyító eszköz vagy azt állítja, hogy a feltevés nem igaz vagy azt, hogy „nem tudom biztosan”. Ezért az Alloy-ról nem lehet kimondani, hogy bizonyító eszköz lenne. Ez azonban azért nem baj, mert létezik egy hipotézis („*small scope hypothesis*”), mely szerint A hibák nagy százaléka megtalálható egy program tesztelése során, ha az összes inputtal ellenőrizhető egy kis scope-on (tartományon) belül. Tehát ha a kis scope-on belül nem létezik ellenpélda, akkor szinte elhanyagolható a valószínűsége annak, hogy a scope-on kívül létezik. Ennek köszönhetően az Alloy így is elég hatékony hiba-kereső eszköz.

Az Alloy egyik nagy előnye, hogy legtöbb modellező eszközzel szemben a rendszerépítés során nyílt világot feltételez. Tehát amikor leírjuk, hogy milyen komponensek vannak, akkor a legtöbb modellező nyelv filozófiája szerint más komponens nem létezhet a világban. A bizonyítások, elemzések során nem képesek olyan elemek figyelembe vételére, melyeket nem tettünk bele a modellbe. Ez azonban problémát okozhat, mert sokszor nem vagyunk képesek minden eshetőséget figyelembe venni.

Vegyünk például egy modellt, melyben vírusok és vírusok elleni védelmek szerepelnek, a követelmény pedig az, hogy a rendszer legyen biztonságos. Ennek feltétele, hogy mindig vírus ellen létezzen védelem. A legtöbb modellező nyelv a rendszert biztonságosnak fogja mondani, ha minden leírt vírus ellen van védelem. Azonban a való világban gyakran előfordul, hogy olyan vírusok lépnek fel, melyek létezéséről addig nem tudtunk. Az Alloy ennek eshetőségét is figyelembe veszi mindaddig, amíg explicit meg nem kötik, hogy a világban semmilyen más vírus nem létezhet, csak amiket a modellben megadtunk.

Az Alloy-ban tehát leírható, hogy milyen komponensek vannak a rendszerben és az is, hogy milyen relációk állnak fent köztük, azonban a komponensek száma nincs megszabva. Ennek köszönhetően feltételezhet olyan komponenst, ami nincs beleírva a modellbe. Az ellenpélda keresésnél ez igen hasznos, ez az Alloy egyik legnagyobb erőssége.

A modellek megalkotása során a dekleratív szemlélet valósul meg. A különböző megkötéseket vizsgálja, de nem annak módját keresi, hogyan lehet egy adott megkötést realizálni, mint az imperatív szemléletű nyelvek, hanem azt, hogy miként lehet felismerni, hogy egy adott megkötés realizálódott. Emiatt az axiómák megadásának sorrendje nem számít.

Az utolsó fontos tulajdonság, hogy az Alloy-ban elkészített modellek automatikusan futtathatóak, azaz az Analyzer képes generálni a modellnek egy olyan példányát, ami kielégíti a megkötéseket, vagy amennyiben ez nem lehetséges, akkor olyan ellenpéldát, amiben egy adott követelmény nem teljesül.

6.4.3. Modell építés

A modellépítés során az Alloy-ban a legalapvetőbb elem a szignatúra. Ez a halmazokhoz hasonlít a legjobban, azonban az objektum orientált szemléletből ismert osztályként is értelmezhető. A szignatúrák között relációk vehetőek fel más szignatúrákkal. Ezeket mezőknek („*field*”) nevezzük, hasonlítanak az osztályokban felvehető tagváltozókhoz. Kardinalitásuk is megadható, alapból egy-egy típusú viszony áll fent köztük, azonban ezt meg lehet változtatni. („*One*”, „*lone*”, „*some*”). A mezőkre egy szignatúrán belül a hivatkozási operátorok segítségével lehet hivatkozni. Ez az eljárás is az objektum orientált nyelvekből lehet ismerős. Az egyik ilyen a relációs kompozíció operátor (.), a másik pedig a reflexív- (*) és a nem-reflexív (^) tranzitív lezárt.

A szignatúrák között hierarchikus viszony is definiálható az „*extends*” kulcsszóval, ami az objektum orientált öröklléshez hasonlít. Ez halmaz-részhalmaz relációt takar, és alapból kizáró („*disjoint*”)

halmazként definiálja a részhalmazokat. Ha azonban a részhalmazok között előfordulhat átfedés is, akkor nem az „*extends*”, hanem az „*in*” kulcsszavat kell használni.

Léteznek absztrakt szignatúrák is, ezek a Java nyelvben létező absztrakt osztályokhoz hasonlóak. Lényege, hogy amennyiben egy szignatúra absztraktként van definiálva, akkor a belőle öröklött szignatúrák az ősszignatúrát teljesen le kell, hogy fedjék. Amennyiben egy szignatúrának nincsenek gyereke, akkor az absztrakt fogalom nem értelmezett rajta.

A másik fontos elem, a tény („*fact*”) célja, hogy a modellben érvényesülő axiómákat meg lehessen fogalmazni. Ezek olyan állítások, melyek a modellben feltétlenül igazak. Különböző kvantorok (minden, létezik, egy vagy nulla, nem létezik, pontosan egy) segítségével elsőrendű logikai állítások adhatóak meg. [4]

A tényeket meg lehet adni egy szignatúrához fűzve is („*appended fac*t”), az ilyen esetekben az adott tény a szignatúra összes példányára teljesül. Léteznek olyan tények is, amik csak bizonyos feltételek teljesülése mellett igazak. Az ilyen „ha… akkor…„ típusú állításokat az Alloy-ban predikátumként („*pred*”) kell megfogalmazni, melynek visszatérési értéke vagy igaz, vagy hamis. Ehhez hasonlítanak a függvények is („*func*”), de ezek visszatérési értéke vagy valamilyen konkrét érték vagy egy érték halmaz.

Amikor egy rendszerben megadtuk az összes axiómát, akkor a „*run*” parancs segítségével kérhetünk egy példányt, ami kielégíti a modellt. Ezt a példányt az Alloy képes vizualizálni. Az általa generált példán látszanak a konkrét példányok és a köztük fennálló viszonyok.

Létezik olyan eset, amikor egy tény triviálisan hamis. Ez a túldeterminálás („*overconstraining*”) extrém esete. Ha ilyen van a modellben, akkor az Alloy nem állítja a modellről, hogy inkonzisztens, csak egyszerűen a futtatás során nem ad semmilyen példányt. Ebből általában arra lehet következtetni, hogy a modell rosszul lett felépítve.

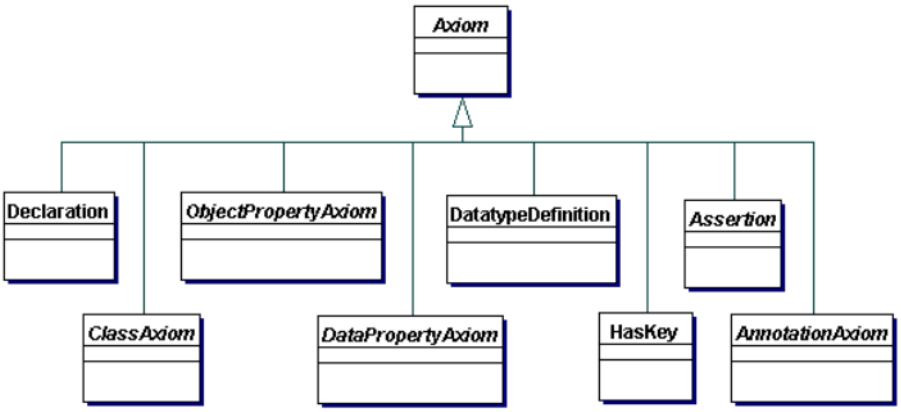
Eddig a modell leírása volt a cél, amit szignatúrákkal és tényekkel lírtunk, az mindig feltétlenül igaz. Ahhoz, hogy valamilyen vizsgálatot lehessen végezni az szükséges, hogy különböző követelmények ellenőrzése lehetséges legyen. Ennek megvalósítására léteznek az Alloy-ban a követelmények („*assert*”), melyek a modellben nem kötelezően igazak. Ezek szintaxisa hasonló a tényekéhez, azonban rendelkezniük kell egyedi névvel. Ez azért fontos, mert az Alloy mindig egy adott követelmény érvényességét képes ellenőrizni, és hogy éppen melyiket vizsgálja, arra név alapján lehet hivatkozni.

A vizsgálatot a „*check*” parancs segítségével lehet elvégezni. Ennek adni kell egy scope méretet. A scope méretének megfelelően megvizsgálja az összes lehetséges esetet, amiben a szignatúráknak a példányszáma maximum annyi, mint a scope méret. Ha scope méreten belül talál ellenpéldát, akkor a követelmény nem teljesül. Nem garantált, hogy a legkisebbet generálja elsőnek, de ha van ellenpélda, akkor mindenképp megtalálja. Ezt vizuálisan meg is lehet jeleníteni. Ha nem talál, akkor azt mondja, hogy valószínűleg teljesül a követelmény.

Alloy-ban történő modellépítés során tehát először fel kell venni a szignatúrákat, majd a modellben mindig igaz axiómákat. Ezután futtatni kell, és a példák tanulmányozása segítségével finomítani rajta, azaz új axiómákat kell beépíteni. Ezt követően lehet vizsgálni a rendszerre vonatkozó követelményeket. Ha egy követelmény nem teljesül, akkor a modellen valamit át kell alakítani, ha teljesül, akkor axiómává kell a további követelmények ellenőrzése során alakítani. A modell fejlesztése és ellenőrzése tehát inkrementálisan történik.

6.5. Hogyan reprezentálhatóak az OWL axiómák Alloy-ban

Az OWL 2 axiómákat a W3C által felállított hierarchia szerint lehet kezelni. [8] (25. ábra)



25. ábra OWL 2 axiómák hierarchiája

Ezek segítségével írhatóak le a modellek struktúrái, tehát a leíró logikában szereplő T-dobozok állítása. Az Alloy elsőrendű logikán alapul, az OWL 2 axiómák pedig leíró logikán. A kettő közötti kapcsolat, hogy a leíró logika T-dobozbeli elemei a következő módon értelmezhetőek az elsőrendű logikában: [17]

- Atomi fogalmak – unáris predikátumok
- Atomi szerepek – bináris predikátumok
- Hierarchikus alárendelés – levezetési szabály
- Fogalom-kifejezések: speciális elsőrendű formulák

A következő példa a golyóállómellénnyel rendelkező pisztolyos őr fogalmát írja le:

F $\doteq$ $\exists$ hasVédelem.GOLYÓÁLLÓMELLÉNY $\sqcap$ $\forall$ hasFegyver.PISZTOLY leíró logikai fogalom megfeleltethető a következő elsőrendű formulának

$\phi(x) = \exists y(\text{hasVédelem}(x, y) \wedge \text{GOLYÓÁLLÓMELLÉNY}(y)) \wedge \forall z(\text{hasFegyver}(x, z) \rightarrow \text{PISZTOLY}(z))$

Ebből az következik, hogy a $\phi(x)$ formula modellje egyben a F fogalom modellje is és megfordítva.

Az elsőrendű predikátumkalkulus kifejezőereje amennyiben csak unáris és bináris predikátummal és csak legfeljebb két szabad változót használva megegyezik a leíró logika kifejezőerejével (ALC nyelvcsalád kiegészítve N-nel, ahol N a minősítetlen számkorlátozást jelenti). Tehát a leíró logikák kifejező ereje gyengébb, mint az elsőrendű predikátumkalkulusé. Ebből következik, hogy minden, ami leírható OWL 2-ben, az leírható Alloy-ban is.

A fentiek miatt nem szükséges egyesével az összes OWL 2 axióma Alloy-ban történő reprezentálást bemutatni, hiszen bizonyított, hogy lehetséges. Az alábbiakban néhány fontosabb axióma átiratát mutatom be.

A következőkben csak néhányat mutatok

1. Declaration - deklaráció

Ezen axióma segítségével lehet egy új osztályt deklarálni, például az Fegyver osztályt létrehozni:

```
<owl:Class rdf:about="#Fegyver"/>
```

Az Alloy szignatúrákkal dolgozik, ami halmazokra emlékeztető entitás, de rendelkezik objektum orientált tulajdonságokkal is. Első megközelítésben egy OWL 2 osztály megfeleltethető egy Alloy szignatúrának:

```
sig Fegyver {}
```

2. SubClassOf - *Alosztályok*

Az osztály axiómák közé tartozik, azt jelenti, hogy egy osztály egy másiknak alosztálya.

```
<owl:Class rdf:about="#Pisztoly">
        <rdfs:subClassOf rdf:resource="# Fegyver "/>
 </owl:Class>
```

Annak ellenére, hogy az Alloy rendelkezik öröklés funkcióval („*extends*”), szerencsésebb nem ezt a megoldást választani, mert nem támogatja a többszörös öröklést. Például ha egy osztály két nem diszjunkt osztályból öröklődik, akkor ez nem okoz inkonzisztenciát a rendszerben. Alloy-ban a többszörös öröklés implementálásra a részhalmaz operátor („*in*”) segítségével van lehetőség.

Ha egy osztály csak egy őssel rendelkezik, akkor ezt könnyen lehet a deklarálásánál megadni:

```
sig Pisztoly in Fegyver {}
```

Ha azonban több ősosztálya is van, akkor ezt az Alloy-ban külön tényként („*fact*”) fel kell venni az alábbi módon (pl. az ember egyszerre erőforrás és célpont is lehet):

```
sig Ember in Celpont {}
fact azEmberEroforras {
    all e : Ember | e in Eroforras
}
```

1. ObjectPropertyRange/ObjectPropertyDomain – Kapcsolat/tartalmazás

Az OWL 2-ben ezen axióma segítségével lehet kifejezni két osztály közötti kapcsolatot.

```
<owl:ObjectProperty rdf:about="#vanVedelem">
        <rdfs:domain rdf:resource="#Celpont"/>
        <rdfs:range rdf:resource="#Vedelem"/>
</owl:ObjectProperty>
```

Az Alloy-ban ez is könnyen reprezentálható, hiszen az Alloy is kapcsolatok definiálásán alapul:

```
sig Bird{ hasFeather:Feather}
```

3. SubObjectPropertyOf – Kapcsolat finomítása

Előfordulhat, hogy a kapcsolatok felépítése is hierarchikus. Az példában az erőforrások és a fegyverek között fennálló reláció finomítása például az emberek és a pisztoly között fennálló reláció. Ez azt jelenti, hogy ha egy ember rendelkezik pisztollyal, abból az következik, hogy ez az ember rendelkezik fegyverrel:

```
  <owl:ObjectProperty rdf:about="#vanPisztolya">
    <rdfs:domain rdf:resource="#Ember"/>
    <rdfs:range rdf:resource="#Pisztoly"/>
    <rdfs:subPropertyOf rdf:resource="#vanFegyvere"/>
```

```
  </owl:ObjectProperty>
```

Az Alloyban ezen információ is egy külön ténnyel írható le:

```
fact subPropvanPisztolya{
        all e : Ember | e. vanPisztolya = Pisztoly implies e. vanFegyvere = Pisztoly
}
```

4. DisjointClasses – kizáró osztályok

Erről az axiómáról már volt szó, olyan osztály listákat ad meg, melyek nem tartalmazhatnak közös elemet. Ha valamire igaz az, hogy páncéltörő, akkor nem lehet sem kés sem pisztoly.

```
<rdf:type rdf:resource="&owl;AllDisjointClasses"/>
     <rdf:Description rdf:about="#PancelToro "/>
     <rdf:Description rdf:about="#Kes"/>
     <rdf:Description rdf:about="#Pisztoly "/>
<rdf:type rdf:resource="&owl;AllDisjointClasses"/>
```

Az Alloyban erre is több megoldás létezik. Az egyik, ha listában szereplő osztályok rendelkeznek egy közös absztrakt ősosztállyal (vagy ha nem rendelkeznek, akkor fel lehet venni hozzájuk egyet.)

```
abstract sig Fegyver {}
sig PancelToro extends Fegyver {}
sig Kes extends Fegyver {}
sig Pisztoly extends Fegyver {}
```

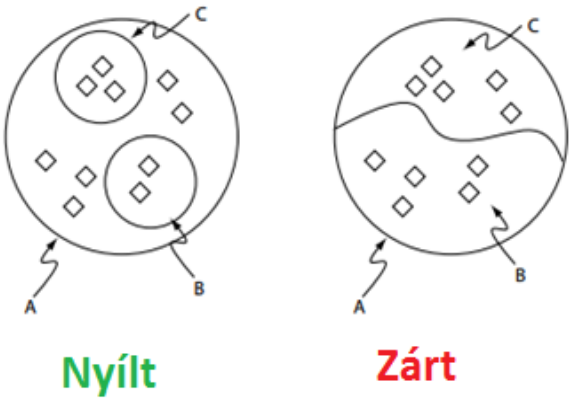
Ez a hivatalos megoldás a diszjunktság reprezentálására. Azonban az implementáció miatt nem kényelmes ezt választani. (Erről részletesebben az „Alkalmazás” fejezetben)

Helyette ez is megadható külső állításként:

```
fact disjointFact1{disjoint [PancelToro, Kes, Pisztoly] }
```

5. EquvivalentClasses – ekvivalens osztályok

Ezen axióma segítségével adható meg, hogy két osztály ekvivalens-e. Ennek igen nagy jelentősége van, mert így megadhatjuk, hogy egy osztályt például az alosztályai lefednek-e. Ezzel kiköthetjük, hogy a világ, amiben dolgozunk zárt-e vagy előfordulhatnak olyan entitások, melyekről a rendszernek nincs tudomása, azaz nyílt. Erről már esett szó az „Érzékenységanalízis” című fejezetben. Az alábbi ábra mutatja a két eset közti különbséget. (26. ábra)



26. ábra Nyíltvilág/Zártvilág

Nyitott esetben előfordulhat, hogy létezik olyan fegyver, amelyik se nem páncéltörő, se nem kés, se nem pisztoly. Zárt esetben ezek létezését kizárjuk. Úgy tudunk zárt esetet előállítani, hogy létrehozunk egy olyan axiómát, mely szerint a kés, a páncéltörő és a pisztoly osztályok uniója lefedi a fegyverek osztályát. Ez a fegyver deklarálásában jelenik meg:

```
<owl:Class rdf:about="#Fegyver">
      <owl:equivalentClass>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
          <rdf:Description rdf:about="# PancelToro "/>
          <rdf:Description rdf:about="# Kes "/>
          <rdf:Description rdf:about="# Pisztoly "/>
        </owl:unionOf>
      </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Az Alloy-ban létrehozandó tény:

```
fact partitionFact(){Fegyver = PancelToro + Kes +  Pisztoly}
```

A fenti axiómák segítségével már elég sok modell felépíthető. Természetesen a teljességhez még sok axióma hiányzik, de nem túl nagy modellekhez már ennyi is elegendő.

7. Alkalmazás

7.1. Megvalósítandó rendszer

A *„Bevezető ”* című fejezetben leírtak alapján a jó rendszerspecifikáció felállítása kritikus a helyes működés és az implementálási költségek szempontjából is. Ha hibatűrő rendszert kell létrehozni, akkor a konzisztencia biztosítása nem elégséges, a robusztusságot is törekedni kell. Ahhoz, hogy ezt elérjük az egyik lehetséges mód, ha a modellen érzékenységanalízis segítségével a megerősítendő pontokat kiválasztjuk és ezeket az implementálás során valóban megerősítjük.

A rendszerspecifikáció folyamatának első szakaszában a szakterület specifikus ismeretekkel rendelkező specialista vesz részt. Számára olyan eszközt kell biztosítani a tudás összegyűjtésére, amely egyszerűen használható, nem igényel nagy háttér informatikai tudást. A rendszerben szereplő fogalmak és az egyszerűbb axiómák összegyűjtésére az ontológia és az azt leíró OWL nyelv kitűnően alkalmas. A taxonómia összegyűjtésének folyamatát könnyen használható szerkesztő eszközök, mint például a Protegé támogatják. Ezek alkalmasak a felállított ontológia konzisztenciájának ellenőrzésére is, azonban olyan analízisek elvégzéséhez, amellyel fel lehetne mérni a funkcionális követelményekben vagy a környezetben bekövetkező változások hatását, már bonyolultabb modellezőkre van szükség.

Ezen modellek felállítását már nem a szakterület ismerője végzi, hanem a modellező eszközt jól ismerő analizáló szakember, aki nem rendelkezik szakterület-specifikus tudással. Ahogy az „*Analízis feladatok és eszközök*” fejezet már bebizonyította, az Alloy alkalmas az érzékenységanalízis megvalósítására. Az analizáló szakember által Alloy-ban felállított modellt már alá lehet vetni az érzékenységanalízisnek.

Az létrehozandó eszköznek tehát valamilyen átmenetet kell tudnia képezni a szakterület specialistája által felépített modell és az elemző számára használható modell között. Ehhez a „*Analízis feladatok és eszközök*” című fejezetben bemutatott OWL 2 és Alloy axiómák közötti konvertálást kell implementálni.

A másik fontos feladat az érzékenységanalízis megvalósítása kapcsolók segítségével. Így ellenőrizni lehet, hogy mi történik, ha valamilyen külső okból valamelyik axióma sérül. Akkor is konzisztens a modell? Ilyen esetekben bekövetkezhet-e nem várt jelenség? A kapcsolók állásától függően a modellnek különböző mutációi hozhatók létre, melyek segítségével a rendszer robusztussága vizsgálható.

7.2. Hiányosságok

Az implementálandó eszköznek (SARI) vannak bizonyos hiányosságai, melyek megszűntetése érdekében további kutatás szükséges. Ezek a következők:

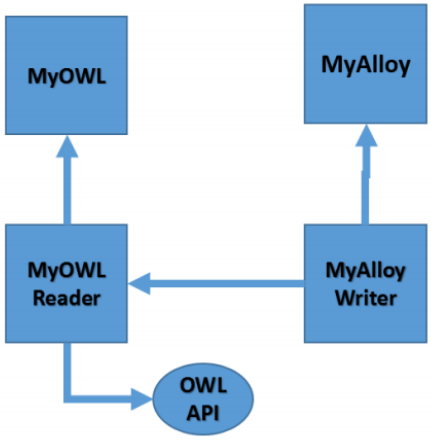
- A kapcsolók beállítása nem automatikus
- Nincs még az összes OWL axióma konvertálása készen: csak olyan rendszerben működik, melyekben csak a korábban felsorolt axióma típusok szerepelnek
- Az érzékenységanalízis esteinek kezelése egyben történik: Mivel az Alloy-ban a külvilágra és a rendszerre vonatkozó követelmények megfogalmazása azonos szintaktikával történik, ezért az implementálandó eszközben (SARI) az érzékenységanalízis esetei nem válnak olyan egyértelműen szét, mint ahogy az az „*Érzékenységanalízis*” című fejezet kifejtette. Emiatt az eredmények kiértékelése nehezebb
- Nincs automatikus kiértékelés
- A mutáció-generálás az Alloy példánygenerálása segítségével történik
- Az Alloy nem bizonyító nyelv: az érzékenységanalízis során kapott eredmények közül nem mindegyik lesz teljesen bizonyos, csak nagy valószínűséggel igaz
- Parsolás során a teljes modellt a memóriában kell tartani, ami nagy modell esetén nagy memóriaigényt jelent.

7.3. Felépítés

Az konvertáló eszköz implementálását Java nyelven kezdtem el, mert létezik egy OWL ontológiák készítésére, változtatására és kinyerésére alkalmas OWL Java API [9]. Előnye, hogy egyszerű használni, csupán egy általános célú, XML elemeket Java entitásokká alakító ingyenes könyvtár (egyetlen JAR fájl) importálására van szükség.

Miután az OWL fájlok XML alapúak, ezért ez a könyvtár további programozás nélkül is képes az OWL modellek elemeit automatikusan Java entitásokká transzformálni. A könyvtár a W3C („*World Wide Web Consortium*”) által kifejlesztett DOM („*Document Object Model*”) [49] szabványon alapul. Ez egy olyan objektummodell, ami lehetővé teszi platform- és nyelv-független módon XML formátumú objektumok és azok közötti interakciók modellezését.

Ennek köszönhetően az OWL Java API független az ontológia kezelését biztosító környezettől. Lényege, hogy az XML-ből származtatott Java entitásokat egy fa struktúrában reprezentálja, hatékony navigációs támogatást adva. Hátránya, hogy a teljes struktúrát a memóriában tárolja, ez nagy modellek esetén nem előnyös, mert nagyon megnövelheti a memória igényt.

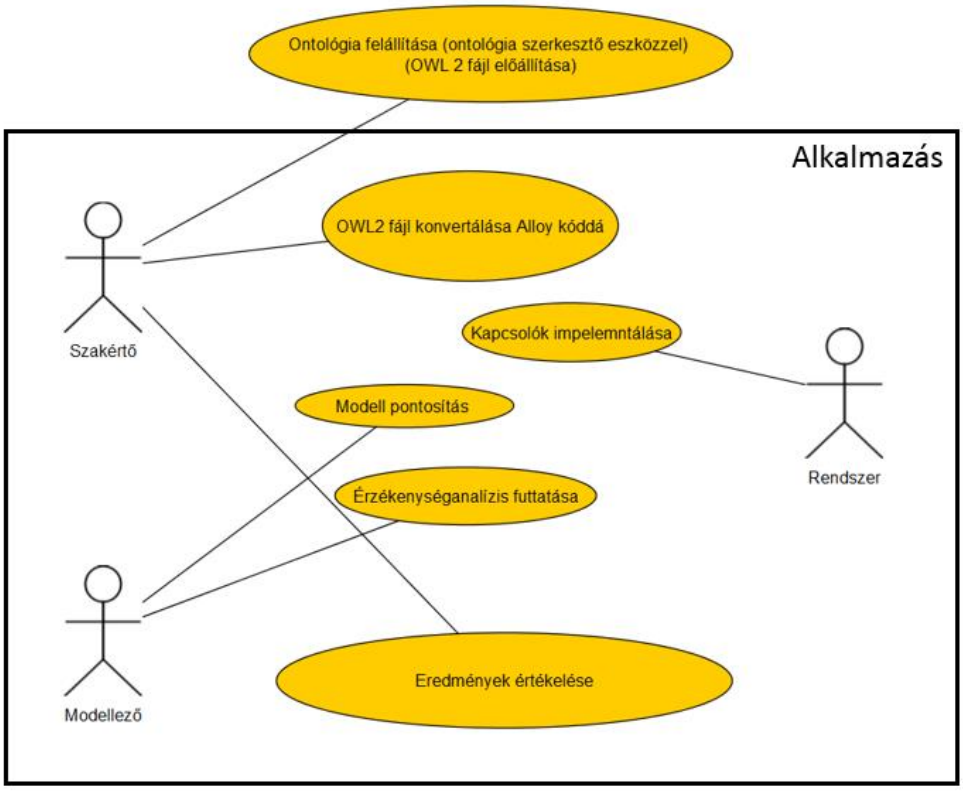


27. ábra Architektúra

Az architektúrális felépítést (27. ábra) a következő módon kell értelmezni: A MyOWL osztály feladata, hogy az OWL API által beolvasott entitásokat MyOWL entitásokká konvertálja. A MyOWL csomagban a különböző OWL entitások (axióma, osztály stb.) implementáltam. Erre azért volt szükség, mert bár az OWL API elvégzi az XML parsolást, az egy entitáshoz tartozó információk néha különböző helyeken jelennek meg. Például ha egy osztály alosztálya egy másiknak, akkor ez az információ nem jelenik meg az adott osztály definiálásában, hanem egy külön axiómaként lesz reprezentálva, pedig az Alloy osztály reprezentációban szükség van erre is. Ezért első megközelítésben kényelmesebb, ha ezen információk egy helyen vannak tárolva. Azonban előfordulhat, hogy a jövőben ez túlságosan megnöveli a memória igényt és változtatásra szorul. A MyAlloyWriter osztály rendelkezik egy MyOWLReader-el, és a benne tárolt MyOWL entitásokat alakítja át MyAlloy entitásokká. Ez az osztály tartalmazza a valódi logikát, ez dönti el, hogy mit mivé kell konvertálni. Ezen felül jelenleg ez tartalmazza az érzékenységanalízist megvalósító funkciót is. A MyAlloy csomagban lévő osztályok felelőssége, hogy képesek legyen maguk kiírásra az Alloy szintaktikájának megfelelően.

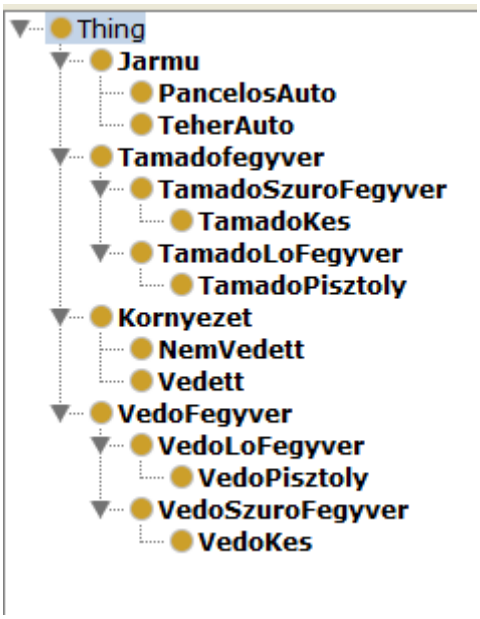
7.4. Példa az érzékenységanalízis egy lehetséges kimenetelére

Az alkalmazás használatát a szállítmányozás példa segítségével mutatom be. Működtetéséhez két, egymástól független felhasználóra van szükség: a szakterület specifikus tudással rendelkező szakértőre és a modellező eszközt ismerő modellezőre. Az lehetséges felhasználói esetek a következők (28. ábra):



28. ábra Use-case diagram

A szakértőnek az alkalmazástól függetlenül egy ontológia szerkesztő program segítségével (pl. Protegé) fel kell vennie a kiinduló ontológiát és létre kell hoznia a hozzá tartozó OWL fájlt (29. ábra). Ebben segítségére lehetnek akár olyan eszközök is, amik az informálisan elmondott rendszerkövetelményeket ontológiává alakítják, mint például a Reuse [50] nevű szoftver.



29. ábra Ontológia

Ezt követően az alkalmazás segítségével a kimeneti fájlból Alloy kódokat kell generáltatni. A rendszer a generált scriptet kiegészíti további axiómákkal, melyek a kapcsolók implementálására szolgálnak. A kapcsolók megvalósítása az alábbi típusú axiómákkal történik:

```
sig Kapcsolo1{
    isAllas: Int
}
```

Minden axiómához tartozik egy ilyen kapcsoló és egy kapcsoló axióma (kivéve a deklaráló axiómákat). A jövőben a biztonságosság eldöntésére vonatkozó kérdésnél a kapcsolók állását kézzel kell beállítani, így létrehozni a mutációkat.

```
assert biztonsagos{
        all e: eset | Kapcsolo.isAllas = 0 => e.biztonsagos = True
}
```

Jelenleg az alkalmazás csak egy axiómát tud kikapcsolni, ezért nem a kérdésben kell beállítani az állásokat, hanem annyi különböző Alloy scriptet generál txt formátumban, ahány axióma a modellben szerepel. A scriptek abban térnek el egymástól, hogy mindegyikben egy-egy axióma kikapcsolt állapotban szerepel. A scriptek pontosítása és futtatása Alloy Analyzer segítségével a modellező feladata, majd az értékelést a szakértőnek kell elvégezni.

A jelenlegi példában egy „EquivalentClasses” típusú OWL axiómához tartozó kapcsolót implementáló scriptet mutatok be. Azért választottam ezt pilot axiómának, mert ez a legfontosabb a biztonság szempontjából, hiszen ennek a segítségével lehet a modell zártságát relaxálni.

A felépített modell alapból konzisztens, futtatása során nem talál ellenpéldát. A „C függelékben” e modell kódja található, a „Példa” fejezetben pedig a modellhez tartozó magyarázat.

Azonban, ha a támadó lőfegyverek zártságára vonatkozó axiómát a hozzá tartozó kapcsolós segítségével érvénytelenítjük, akkor az Alloy már képes ellenpéldát generálni.

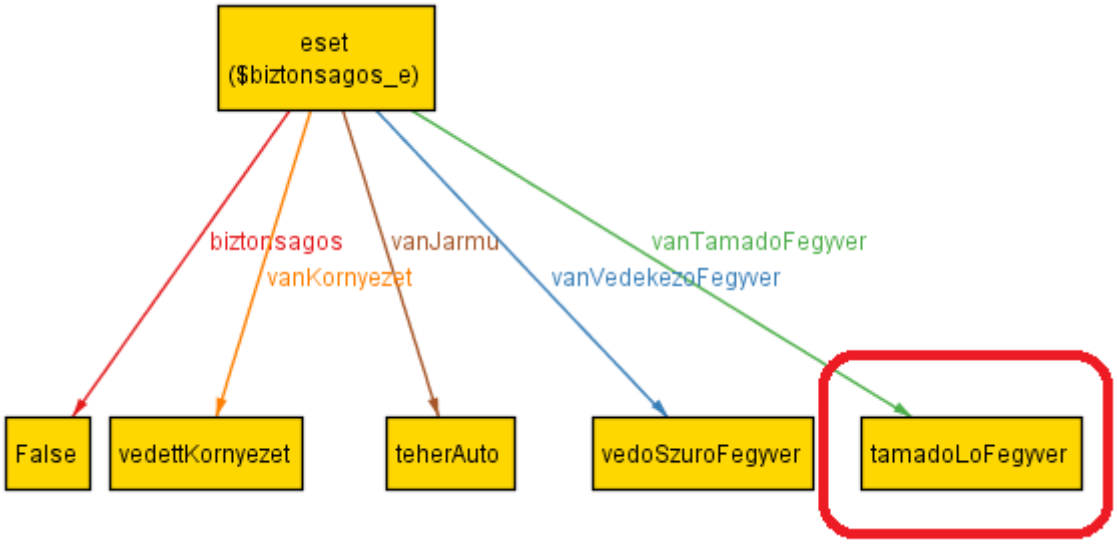
```
fact zartTamadoLoFegyverKapcsolo{
        Kapcsolo.isAllas = 1 =>  tamadoLoFegyver = tamadoPisztoly
}
```

Ennek oka, hogy zárt világ esetén a tamadoLoFegyver és a tamadoPisztoly osztályok megegyeznek. A védelemmel kapcsolatos követelmények csak pisztolyra vonatkoznak, nem számolnak azzal az esettel, ha létezhet más lőfegyver is, például a páncéltörő rakéta.

Az érzékenységanalízis rámutat egy a környezeti feltételek változásakor fellépő potenciális veszélyforrásra.

Ezzel bizonyítható, hogy van olyan környezeti faktor, amire nézve a rendszer nem robusztus.

Kiemelendő, hogy a tradicionális formális követelményanalízis eszközök a zárt világ paradigmán belül maradnak és nem alkalmasak egy ilyen jellegű veszély felismerésére.



30. ábra Ellenpélda

8. Példa

Ebben a fejezetben a dolgozat során többször említett szállítmányozási példa paramétereit adom meg. Az entitásokat és a köztük lévő hierarchikus viszonyokat vettem csak OWL-ben fel. Ezt követően az alkalmazás segítségével kigeneráltam az Alloy kódot és kiegészítettem további két tervezési sajátosságból fakadó axiómával.

Szállítmányozással kapcsolatos entitások:

A feladat egy szállítmányozással kapcsolatos modell elkészítése és analizálása. A teljes OWL 2 fájl és az Alloy kódok a B és C függelékben találhatóak.

Az 29. ábra mutatja a modellben szereplő entitásokat. Ezeket zártvilág szemléletben írtam le: az öröklési hierarchiában egy szinten lévő osztályok egymással kizáróak és teljesen lefedik az ősosztályt.

A mutációk előállítását egy „Eset” nevű entitás vezérli. Ennek segítségével le lehet írni az Alloy kódban a rendszer állapotait: minden ősosztállyal relációban áll. Attól függően állíthatóak a rendszer állapotai, hogy az ősosztályok melyik gyerekében található a reláció értéke. A rendszer minden állapotában szerepel egy jármű típus, egy környezet típus, egy támadó- és egy védekező fegyver típus. Ezeknek az alábbi táblázatban jelölt kombinációi vezetnek biztonságos esetekhez (a dc – „*don’t care*” állapotot tükröz):

Környezet	VédőFegyver	Jármű	Támadófegyver
VédettKörnyezet	VédőLőFegyver	dc	dc
VédettKörnyezet	dc	PáncélosAutó	dc
VédettKörnyezet	dc	dc	!TámadóLőFegyver
NemVédettKörnyezet	VédőLőFegyver	PáncélosAutó	dc
NemVédettKörnyezet	dc	PáncélosAutó	!TámadóLőFegyver
NemVédettKörnyezet	VédőLőFegyver	dc	!TámadóLőFegyver

31. ábra Biztonságos esetek

Ezek implementálása Alloy-ban:

```
//biztonsagossag kriteriuma:
fact nemBiztonsagos{
    all e:eset | ((e.vanKornyezet in vedettKornyezet and
                   e.vanVedekezoFegyver in vedoLoFegyver and
                   e.vanJarmu  in jarmu and
                   e.vanTamadoFegyver in tamadoFegyver)

                   or

                  (e.vanKornyezet in vedettKornyezet and
                   e.vanVedekezoFegyver in vedoFegyver and
                   e.vanJarmu  in pancelosAuto and
                   e.vanTamadoFegyver in tamadoFegyver)

                   or

                  (e.vanKornyezet in vedettKornyezet and
                   e.vanVedekezoFegyver in vedoFegyver and
                   e.vanJarmu  in jarmu and
                   e.vanTamadoFegyver not in tamadoLoFegyver)

                  or

                  (e.vanKornyezet in nemVedettKornyezet and
                   e.vanVedekezoFegyver in vedoLoFegyver and
                   e.vanJarmu  in pancelosAuto and
                   e.vanTamadoFegyver in tamadoFegyver)

                  or

                  (e.vanKornyezet in nemVedettKornyezet and
                   e.vanVedekezoFegyver in vedoFegyver and
                   e.vanJarmu  in pancelosAuto and
                   e.vanTamadoFegyver not in tamadoLoFegyver)

                  or

                  (e.vanKornyezet in nemVedettKornyezet and
                   e.vanVedekezoFegyver in vedoLoFegyver and
                   e.vanJarmu  in jarmu and
                   e.vanTamadoFegyver not in tamadoLoFegyver))


                  =>
                  e.biztonsagos = True
                  else e.biztonsagos = False
}
```

32. ábra Alloy - Biztonságos esetek

A rendszerben további két axióma az alábbi követelmények tervezési sajátosság szintű reprezentációi:

- Ha nem védett a környezet, akkor páncélos autóval kell utazni és a védekezéshez lőfegyvert kell használni.

```
fact haNemVedettAkornyezet{
        all e:eset | e.vanKornyezet in nemVedettKornyezet
                =>
                (e.vanJarmu in  pancelosAuto
                and
                e.vanVedekezoFegyver in  vedoLoFegyver)
}
```

- Pisztolyos támadás lehetőségének fennállása esetén páncélos autóval, védett útvonalon szabad utazni.

```
fact erosTamadas{
        all e : eset | e.vanTamadoFegyver in tamadoPisztoly
                        =>
                (e.vanKornyezet in vedettKornyezet
                and
                e.vanJarmu in  pancelosAuto)
}
```

A modell konzisztens, futtatása során nem talál ellenpéldát.

9. Értékelés

9.1. Tudományos értékelés

Kapcsolódás a tervezési folyamathoz

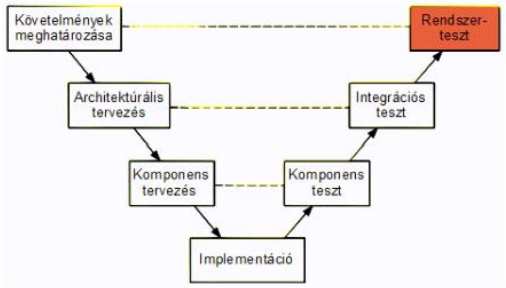
A TDK dolgozatomban bemutatott érzékenységanalízis célja az informatikai rendszerekkel szembeni követelményekből felállított specifikáció minőségének javítása. Az érzékenységanalízis közvetlen haszna a bevezetőben említett gazdasági okok miatt a teljes fejlesztési folyamat hatékonyságára kiterjed.

Alkalmas a kritikus követelmények azonosításán túl a specifikációban lévő strukturális redundancia feltárására is. Legcélszerűbb alkalmazási pontja a specifikációs folyamatban a felállított modell konzisztenciájának és helyességének bizonyítását közvetlenül követő fázis, hiszen ekkor már egy formailag helyes modell áll rendelkezésre.

A módszer sajátossága az, hogy formális analízissel képes rámutatni a rendszer azon pontjaira, amelyek esetében valamely jövőbeli változás miatt azt módosítani kell, illetve megmutatja, melyek azok a tervezéskor még nem ismert változások, melyeket a rendszer elvisel.

Ebben a megközelítésben például meg tudtam oldani azt az irodalomban eddig nem vizsgált esetet, hogy mi történik akkor, ha a tervezéskori zártvilág modell zártsága többé nem érvényes és egy új szituációban egy olyan nyíltvilág modellt kell bevezetni, amely képes a működési modellbeli változásokat is befogadni.

Ha a rendszer kidolgozását V-modell típusú folyamatként értelmezzük, akkor az érzékenységanalízis helye a követelmények meghatározásakor és az erre irányuló rendszer-teszt szakaszban van. (33. ábra)



33. ábra V-modell

Kapcsolódás az ipari szabványokhoz, korlátozások

A TDK-ban ismertetett módszerem számos ponton támaszkodik az ipari gyakorlatban használt metodikákra, szabványokra és eszközökre. A rutinfeladatok zömére így professzionális háttér áll rendelkezésre. Viszonylag új elem azonban a követelmények részletes formális vizsgálat, hiszen azt megelőzően alapvetően csak a követelménymodell jól formáltságát és formális teljességét és ellentmondás-mentességét vizsgálták.

Az én megközelítésemben az a sajátos eredmény, hogy formális analízis segítségével tárja fel a követelményrendszer kritikus pontjait és ezzel megteremti a feltételét a fókuszált szakértői lektorálásnak.

Ez célját tekintve szűkebb, mint a teljes formalizáláson alapuló vizsgálat (például amit a RODIN EU projekt alapján kidolgozott Formal Mind [56] eszköz család végez), de ez utóbbi alkalmazási köre jelenleg korlátozott, hiszen a kézi formalizálás Event-B szakértőt igényel.

Módszeremben a ReqIF ipari szabvány segítségével definiálható követelmények vizsgálata lehetséges, amit az iparban széles körben elterjedt eszközök (pl. IBM DOORS, Eclipse RMF) támogatnak. Az általam alkalmazott megkötések a módszer alkalmazását a strukturális vizsgálathoz kötik.

A részletes specifikáció ellenőrzése ennek az első fázisnak nem volt célja, hiszen az adattípusok kezelése és az időbeliség vizsgálata már a modellalapú tesztelés egy olyan változatát igényli, amely önmagában sem lezárt kutatási téma. Megjegyzendő ugyanakkor, hogy a követelményrendszer általam megcélzott strukturális helyessége esetén az egyedi részletes követelményeknek a további tervezési fázisokban történő létrehozása és kezelése is jóval könnyebb.

Ennek a korlátozásnak oka egyébként a felhasznált matematikai apparátus és az azt implementáló eszközkészlet. Az adat és időtartománybeli viselkedés részletes vizsgálata ugyanis hatalmas tervezési tér bejárását igényelné, amire a jelenlegi megközelítés ipari méretű feladatoknál alkalmatlan.

Egy köztes megoldást jelenthet a kvalitatív absztrakción alapuló megközelítés, amelynél az egyes működési tartományokat egyetlen elemmé vonnánk össze és ez jó közelítő eredményeket tudna szolgáltatni. [57] Az idő kezelésére az Allen-féle intervallum logika [44] adhatna lehetőséget, megvalósíthatóságának bizonyítása további kutatást igényel.

9.2. Technikai értékelés

Az érzékenységanalízis során a mutációkat az ontológia fogalmi elem készlete alapján generálom. Ez az ötlet hasonló ahhoz, mint amikor szoftver hibákat úgy modelleznek, hogy a programozási elv egyes alapkonstrukcióit helyettesítik másokkal. Ennek a hipotetikus mutációs modellnek a valósághűségét egyelőre csak szakemberekkel való interjú alapján lehet eldönteni.

Ennek a folyamatnak még az elején vagyok, de néhány jó nevű szakértő a modellben ráismert a korábbi gyakorlatából tipikus problémákra. [58] Szervezettebb analízist tesz majd lehetővé a SARI bevonása a Cecris [59] projektbe.

A követelményspecifikációs folyamat egy lehetséges indítása a követelmények összegyűjtése majd azok ontológiában történő reprezentálása. Erre már léteznek az iparban használt eszközök, például a „Reuse”. Azonban az ontológiák és a hozzájuk tartozó bizonyító motorok nem képesek a modell konzisztenciáján és a helyességén kívül mást is vizsgálni, ezért a bennük megfogalmazott axiómákat érdemes egy másik technológiával is reprezentálni.

Az érzékenységanalízis megvalósítására egy alkalmas eszköz az Alloy, ami magasabb rendű logikán alapszik, mint az ontológiák, ezért benne az összes ontológia-beli axióma megadható.

Az SARI jelenlegi kidolgozottság szintje egy koncepcionális demonstrációs szint. Jelenleg csak néhány, az ontológiát leíró OWL 2 konstrukció Alloy kóddá történő átalakítás lett kidolgozva, ezek azonban már megvalósítják a legizgalmasabb kérdéseket, mint nyíltvilág/zártvilág, helyes és helytelen tartalmazási relációk stb. A továbbiakban a többi konstrukció konvertálásának implementálása folyamatosan történik, ennek időigénye az konstrukciók számával arányos.

Az érzékenységanalízis eredményeinek az eredeti modellre való visszavetítése egyelőre kézi, ennek automatizálása szintén további kutatást igényel. Ehhez technikailag éppen a követelménykezelés egyik legfontosabb alapelemét kell majd implementálni, a nyomon követhetőséget. Szerencsére a

követelmény-ontológia-SARI lánc során az egyes elemek nagyjából önállóan transzformálódnak, így ez viszonylag egyszerű lesz.

9.3. Összefoglaló

Összefoglalva a módszer a követelményrendszer strukturális vizsgálatát teszi lehetővé. Az Alloy bizonyítási módszereit támogató különböző SAT megoldók nem teszik lehetővé a függéseken túl a követelmények közötti kapcsolatok szemantikájának interpretálását.

Az előzőekből következően az alábbi továbbfejlesztési lehetőségek foglalhatóak össze:

- adattípusok és kvantitatív tartomány vizsgálata predikátum absztrakció [45] segítségével
- időbeliség kezelése
- eredmények automatikus visszacsatolása a modellbe

A TDK dolgozat célja az informatikai rendszerek és az általuk vezérelt fizikai vagy üzleti folyamatok követelményrendszereinek érzékenységanalízissel történő vizsgálatának bemutatása és az erre a célra alkalmas implementálandó alkalmazás (SARI) megvalósíthatóságának demonstrálása.

Az érzékenységanalízis követelményrendszer specifikációban elfoglalt helye miatt szükség van a folyamatban előtte lévő technológiákkal való együttműködés megteremtésére, amely szintén az SARI feladata.

9.4. Köszönetnyilvánítás

Az alábbiakban szeretnék köszönetet mondani azon szakértőknek, akik segítették munkámat:

Dr.-habil Majzik István, egyetemi docens,

Gönczy László, egyetemi tanársegéd.

A munkám során módom volt több olyan neves nemzetközi elméleti kutatóval illetve gyakorlati szakembernek bemutatni munkámat, akik mind elméleti, mind gyakorlati szempontból hasznos kérdésekkel és tanácsokkal valamint nem utolsó sorban bíztatással támogatták munkámat:

Nuno Pedro Silva (Critical Software),

Alexandre Esper (Critical Software),

Professor Alexander Romanovsky (Coordinator of the FP6 ICT Rigorous Open Development Environment for Complex Systems Project (RODIN))

10. Irodalomjegyzék

[1.] A Survey on how Description Logic Ontologies Benefit from Formal Concept Analysis, Barıs Sertkaya (2011)
http://arxiv.org/pdf/1107.2822.pdf

[2.] OntoComP: A Protege Plugin for Completing OWL Ontologies, Barıs Sertkaya
http://wwwtcs.inf.tu-dresden.de/~sertkaya/publications/Sert09b.pdf

[3.] https://code.google.com/p/ontocomp/

[4.] Enhancing OWL Ontologies via Formal Concept Analysis, Sebastian Rudolph and Barış Sertkaya (2011)
http://ijcai-11.iiia.csic.es/files/proceedings/T13-ijcai11Tutorial.pdf

[5.] OntoComP System Description, Barıs Sertkaya
http://ceur-ws.org/Vol-477/paper_55.pdf

[6.] Ontology Design with Formal Concept Analysis, Marek Obitko, Václav Snásel, and Jan Smid http://ceur-ws.org/Vol-110/paper12.pdf
[7.] Conceptual Knowledge Processing with Formal Concept Analysis and Ontologies, Philipp Cimiano, Andreas Hotho, Gerd Stumme, Julien Tane http://www.kde.cs.uni-kassel.de/hotho/pub/2004/icfca04.pdf
[8.] http://www.w3.org/TR/owl2-syntax/#Axioms
[9.] http://owlapi.sourceforge.net/
[10.] Automatic Mapping of OWL Ontologies into Java, Aditya Kalyanpur ,Daniel JimÈnez Steve Battle Julian Padget http://www.itu.dk/~martynas/Thesis/Articles/Automatic%20Mapping%20of%20OWL%20Ontologies%20to%20Java.pdf
[11.] Conceptual Knowledge Processing with Formal Concept Analysis and Ontologies, Philipp Cimiano, Andreas Hotho, Gerd Stumme, Julien Tane http://www.kde.cs.uni-kassel.de/hotho/pub/2004/icfca04.pdf
[12.] Analysing Web Ontology in Alloy: A Military Case Study, Jin Song Dong Jun Sun Hai Wang, Chew Hung Lee Hian Beng Lee http://www.macs.hw.ac.uk/~lilia/PapersForCoursework/WebOntologyinAlloyMilitaryCaseStudy.pdf
[13.] Ontologies and Description Logics, Carlos Areces (2003) http://www.w3.org/TR/owl2-syntax/
[14.] http://www.w3.org/TR/owl2-manchester-syntax/
[15.] The Manchester OWL Syntax, Matthew Horridge, Nick Drummond, John Goodwin, Alan Rector, Robert Stevens , and Hai H Wang http://webont.org/owled/2006/acceptedLong/submission_9.pdf
[16.] Rules and Ontologies in F-logic, Michael Kifer http://www3.cs.stonybrook.edu/~kifer/TechReports/ontologies-rules-flogic.pdf
[17.] Tudás alapú rendszerek és technológiák (2010), Dr. Bognár Katalin http://www.inf.unideb.hu/~bognar/mestint4/mestint_konyv.pdf
[18.] Description Logics over Lattices with Multi-valued Ontologies, Stefan Borgwardt and Rafael Penaloza http://lat.inf.tu-dresden.de/research/papers/2011/BoPe-IJCAI11.pdf
[19.] Completing Description Logic Knowledge Bases using Formal Concept Analysis, Franz Baader, Bernhard Ganter, Barıs¸ Sertkaya, and Ulrike Sattler http://www.cs.man.ac.uk/~sattler/publications/InstExp-IJCAI-07.pdf
[20.] http://alloy.mit.edu/alloy/
[21.] https://www.doc.ic.ac.uk/project/examples/2007/271j/suprema_on_alloy/Web/
[22.] http://en.wikipedia.org/wiki/Partially_ordered_set
[23.] http://www.fca.radvansky.net/downloads.php
[24.] Formal Concept Analysis in knowledge processing: A survey on models and techniques (2013), Jonas Poelmansa, Sergei O. Kuznetsovb, Dmitry I. Ignatovb, Guido Dedenea
[25.] Formal concept analysis in knowledge processing: A survey on applications, Jonas Poelmansa, Sergei O. Kuznetsovb, Dmitry I. Ignatovb, Guido Dedenea http://www.sciencedirect.com/science/article/pii/S0957417413002959
[26.] An Introduction to using Relation Algebra with FCA, Uta Priss (2009) http://www.upriss.org.uk/papers/relalgintro.pdf
[27.] http://www.omg.org/spec/ReqIF/1.1/
[28.] http://www.computer.org/portal/web/guest/home
[29.] IEEE 830 http://www.midori-global.com/downloads/jpdf/jira-software-requirement-specification.pdf
[30.] http://sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK)
[31.] http://sebokwiki.org/wiki/Glossary_of_Terms

[32.] http:// www.standishgroup.com/chaos.html
[33.] Towards Effort and Quality Estimation of V&V Processes Dr. András Pataricza
[34.] http://searchsap.techtarget.com/definition/requirements-stability-index
[35.] Ontology Design and Software Technology, Colloquium (2003), H. Knublauch
[36.] Building Formal Requirements Models for Reliable Software, Axel van Lamsweerde http://www.info.ucl.ac.be/~avl/files/avl-AdaEurope.pdf
[37.] Requirements Engineering by Use Case Analysis Dániel Varró https://inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/msc-t%C3%A1rgyak/modellalap%C3%BA-szoftvertervez%C3%A9s/11/UseCaseModeling.pdf
[38.] http://www.cs.unc.edu/~stotts/145/CRC/state.html
[39.] An approach to security requirements engineering for a high assurance system (2002), Cynthia E. Irvine , Timothy Levin , Jeffery D. Wilson , David Shifflett , Barbara Pereira http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.doors.doc/pdf92/doors_getting_started.pdf
[40.] http:// .mff.cuni.cz/~bartak/constraints/binary.html
[41.] IBM Rational DOORS Introduction Balázs Polgár
[42.] https://www.youtube.com/watch?v=Vqwgty4GqLU
[43.] A szemantikus világháló elmélete és gyakorlata, Szeredi P., Lukácsy G., Benkő T. - 193-195.o
[44.] Maintaining knowledge about temporal intervals, J. F. Allen, (Communications of the ACM , vol. 26, Issue no. 11, pp. 832-843, November 1983.)
[45.] http://www.researchgate.net/publication/226562449_Verification_by_Abstraction
[46.] Formális módszerek az informatikában (2005), Pataricza András
[47.] Bevezetés a modern logikába (2000), Ruzsa Imre
[48.] http://www.omg.org/
[49.] http://hu.wikipedia.org/wiki/Document_Object_Model
[50.] http://www.reusecompany.com/
[51.] Information technology -- Common Logic (CL): a framework for a family of logic-based languages http://www.iso.org/iso/catalogue_detail.htm?csnumber=39175
[52.] A Lightweight Approach for Defining the Formal Semantics of a Modeling Language (2008), Pierre Kelsen, Qin Ma http://link.springer.com/chapter/10.1007%2F978-3-540-87875-9_48?LI=true
[53.] OWL Semantics http://www.w3.org/TR/owl-semantics/
[54.] Towards Effort and Quality Estimation of V&V Processes, András Pataricza (DEVVARTS'14 Workshop)
[55.] Towards Unified Dependability Modeling and Analysis, András Pataricza, Ferenc Győr http://subs.emis.de/LNI/Proceedings/Proceedings41/GI-Proceedings.41-11.pdf
[56.] Formal Mind eszköz család http://www.formalmind.com/
[57.] Invisible formal methods for embedded control systems (2003), A. Tiwari, N. Shankar, and J. Rushby (Proceedings of the IEEE, 91(1):29–39, January 2003.)
[58.] Alexandre Esper, Nuno Pedro Silva személyes közlés
[59.] Cecris EU projekt http://www.cecris-project.eu/

11. Ábrajegyzék

12. Függelékek

12.1. A függelék – Fogalmak

12.1.1. Tulajdonságok

- **Robusztusság** („*Robustness*”) – Annak a mértéke, hogy egy rendszer vagy egy komponens mennyire tud továbbra is helyesen működni valamilyen érvénytelen bemenet vagy kihívásokkal teli környezeti feltételek mellett.
- **Biztonságosság** („*Safety*”) – Arra vonatkozó elvárás, hogy egy rendszer előre definiált feltételek mellett nem jut olyan állapotba, amelyben emberi élet, egészség, vagyon vagy a környezet megsérülhet.

- **Karbantarthatóság** („*Maintainability*”) – Annak a valószínűsége, hogy egy rendszer vagy egy rendszer elem egy meghatározott környezetben, meghatározott erőforrásokkal, meghatározott időn belül helyreállítható.
- **Követhetőség** („*Tracebility*”) – Annak a mértéke, hogy a fejlesztés folyamatának két vagy több terméke között a kapcsolat létrehozható

12.1.2. Rendszerleírók

- **Absztrakt modell** („*Abstract model*”) – Egy absztrakt vagy koncepcionális reprezentációja egy olyan rendszernek, amelynek nincs fizikai vagy konkrét léte.
- **Attribútum** („*Attribute*”) – Egy entitás velejáró tulajdonsága vagy jellemzője, ami kifejezhető kvalitatív vagy kvantitatív emberi vagy automatizált eszközökkel.
- **Hierarchia** („*Hierarchy*”) – Rendezés elemek között (objektumok, nevek, értékek, kategóriák…), amely során az elemek között alá-és fölérendeltségi valamint azonos szinten levési viszonyok definiáltak. A szintek a hierarchiában reprezentálhatnak irányítást, tulajdonlást vagy tartalmazást a magasabb és az alacsonyabb szinteken lévő elemek között.

12.2. B függelék – OWL 2 kód

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://www.semanticweb.org/agi/ontologies/2014/9/untitled-ontology-42"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     ontologyIRI="http://www.semanticweb.org/agi/ontologies/2014/9/untitled-ontology-42">
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
    <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Declaration>
        <Class IRI="#Jarmu"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Kornyezet"/>
    </Declaration>
    <Declaration>
        <Class IRI="#NemVedett"/>
    </Declaration>
    <Declaration>
        <Class IRI="#PancelosAuto"/>
    </Declaration>
    <Declaration>
        <Class IRI="#TamadoKes"/>
    </Declaration>
    <Declaration>
        <Class IRI="#TamadoLoFegyver"/>
    </Declaration>
    <Declaration>
        <Class IRI="#TamadoPisztoly"/>
    </Declaration>
    <Declaration>
        <Class IRI="#TamadoSzuroFegyver"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Tamadofegyver"/>
    </Declaration>
    <Declaration>
        <Class IRI="#TeherAuto"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Vedett"/>
```

```
</Declaration>
<Declaration>
    <Class IRI="#VedoFegyver"/>
</Declaration>
<Declaration>
    <Class IRI="#VedoKes"/>
</Declaration>
<Declaration>
    <Class IRI="#VedoLoFegyver"/>
</Declaration>
<Declaration>
    <Class IRI="#VedoPisztoly"/>
</Declaration>
<Declaration>
    <Class IRI="#VedoSzuroFegyver"/>
</Declaration>
<SubClassOf>
    <Class IRI="#Jarmu"/>
    <Class abbreviatedIRI="owl:Thing"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Kornyezet"/>
    <Class abbreviatedIRI="owl:Thing"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#NemVedett"/>
    <Class IRI="#Kornyezet"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#PancelosAuto"/>
    <Class IRI="#Jarmu"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#TamadoKes"/>
    <Class IRI="#TamadoSzuroFegyver"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#TamadoLoFegyver"/>
    <Class IRI="#Tamadofegyver"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#TamadoPisztoly"/>
    <Class IRI="#TamadoLoFegyver"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#TamadoSzuroFegyver"/>
    <Class IRI="#Tamadofegyver"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Tamadofegyver"/>
    <Class abbreviatedIRI="owl:Thing"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#TeherAuto"/>
    <Class IRI="#Jarmu"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Vedett"/>
    <Class IRI="#Kornyezet"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#VedoFegyver"/>
    <Class abbreviatedIRI="owl:Thing"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#VedoKes"/>
    <Class IRI="#VedoSzuroFegyver"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#VedoLoFegyver"/>
    <Class IRI="#VedoFegyver"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#VedoPisztoly"/>
    <Class IRI="#VedoLoFegyver"/>
</SubClassOf>
<SubClassOf>
```

```
        <Class IRI="#VedoSzuroFegyver"/>
        <Class IRI="#VedoFegyver"/>
    </SubClassOf>
    <DisjointClasses>
        <Class IRI="#Jarmu"/>
        <Class IRI="#Kornyezet"/>
        <Class IRI="#Tamadofegyver"/>
        <Class IRI="#VedoFegyver"/>
    </DisjointClasses>
    <DisjointClasses>
        <Class IRI="#NemVedett"/>
        <Class IRI="#Vedett"/>
    </DisjointClasses>
    <DisjointClasses>
        <Class IRI="#PancelosAuto"/>
        <Class IRI="#TeherAuto"/>
    </DisjointClasses>
    <DisjointClasses>
        <Class IRI="#VedoLoFegyver"/>
        <Class IRI="#VedoSzuroFegyver"/>
    </DisjointClasses>
    <DisjointUnion>
        <Class IRI="#Jarmu"/>
        <Class IRI="#PancelosAuto"/>
        <Class IRI="#TeherAuto"/>
    </DisjointUnion>
    <DisjointUnion>
        <Class IRI="#Kornyezet"/>
        <Class IRI="#NemVedett"/>
        <Class IRI="#Vedett"/>
    </DisjointUnion>
    <DisjointUnion>
        <Class IRI="#Tamadofegyver"/>
        <Class IRI="#TamadoLoFegyver"/>
        <Class IRI="#TamadoSzuroFegyver"/>
    </DisjointUnion>
    <DisjointUnion>
        <Class IRI="#VedoFegyver"/>
        <Class IRI="#VedoLoFegyver"/>
        <Class IRI="#VedoSzuroFegyver"/>
    </DisjointUnion>
</Ontology>
```

12.3. C függelék – Alloy kód

```
sig Kapcsolo{
	isAllas:Int
}

sig kornyezet {}
sig tamadoFegyver {}
sig vedoFegyver {}
sig jarmu{}

sig auto extends jarmu {}
sig hajo extends jarmu{}
sig vonat extends jarmu{}
//jarmu zart
fact zartJarmu{
	jarmu = auto + hajo + vonat
}

sig teherAuto extends auto{}
sig pancelosAuto extends auto{}
//auto zart
fact zartAuto{
	auto = teherAuto + pancelosAuto
}


sig vedettKornyezet extends kornyezet{}
sig nemVedettKornyezet extends kornyezet{}
//Kornyezet zart
fact zartKornyezet{
	kornyezet = vedettKornyezet + nemVedettKornyezet
}

sig tamadoSzuroFegyver extends tamadoFegyver {}
sig tamadoKes extends tamadoSzuroFegyver {}
sig tamadoLoFegyver extends tamadoFegyver {}
sig tamadoPisztoly extends tamadoLoFegyver{}
sig tamadoPanceltoro extends tamadoLoFegyver{}
//tamadofegyver zart (FIGYELEM, nincs megadva az osszes tamadoFegyver,
//hiszen a tamadoSzurofegyveren es a tamadoLofegyveren belul sok fegyver letezhet
fact zartTamadoFegyver{
	tamadoFegyver = tamadoSzuroFegyver + tamadoLoFegyver
}


fact zartTamadoLoFegyverKapcsolo{
	Kapcsolo.isAllas = 1 =>  tamadoLoFegyver = tamadoPisztoly
}
/*
//tamadoLoFegyver zart
fact zartTamadoLoFegyver{
	tamadoLoFegyver = tamadoPisztoly
}
*/




sig vedoSzuroFegyver extends vedoFegyver {}
sig vedoKes extends vedoSzuroFegyver {}
sig vedoLoFegyver extends vedoFegyver {}
sig vedoPisztoly extends vedoLoFegyver{}
//sig vedoPanceltoro extends vedoLoFegyver{}
//vedo fegyver zart (FIGYELEM, nincs megadva az osszes vedoFegyver,
//hiszen a vedoSzurofegyveren es a vedoLofegyveren belul sok fegyver letezhet
fact zartVedoFegyver{
	vedoFegyver = vedoSzuroFegyver + vedoLoFegyver
}


sig eset{
	vanKornyezet: one kornyezet,
	vanTamadoFegyver: one tamadoFegyver,
	vanVedekezoFegyver: one vedoFegyver,
	vanJarmu: one jarmu,
	biztonsagos: one Bool
}


//Ha nem vedett a kornyezet, akkor (pancelos autoval kell menni vagy
//vedoLofegyverrel kell vedekezni)
fact haNemVedettAkornyezet{
		all e:eset | e.vanKornyezet in nemVedettKornyezet
				=>
				(e.vanJarmu in  pancelosAuto
				and
				e.vanVedekezoFegyver in  vedoLoFegyver)
}


fact erosTamadas{
	all e : eset | e.vanTamadoFegyver in tamadoPisztoly
				=>
			(e.vanKornyezet in vedettKornyezet
			and
			e.vanJarmu in  pancelosAuto)
}
```

```
//biztonsagossag kriteriuma:
fact nemBiztonsagos1{
        all e:eset | ((e.vanKornyezet in vedettKornyezet and
                                e.vanVedekezoFegyver in vedoLoFegyver and
                                e.vanJarmu  in jarmu and
                                e.vanTamadoFegyver in tamadoFegyver)

                                or

                               (e.vanKornyezet in vedettKornyezet and
                                e.vanVedekezoFegyver in vedoFegyver and
                                e.vanJarmu  in pancelosAuto and
                                e.vanTamadoFegyver in tamadoFegyver)

                                or

                               (e.vanKornyezet in vedettKornyezet and
                                e.vanVedekezoFegyver in vedoFegyver and
                                e.vanJarmu  in jarmu and
                                e.vanTamadoFegyver not in tamadoLoFegyver)

                               or

                               (e.vanKornyezet in nemVedettKornyezet and
                                e.vanVedekezoFegyver in vedoLoFegyver and
                                e.vanJarmu  in pancelosAuto and
                                e.vanTamadoFegyver in tamadoFegyver)

                               or

                               (e.vanKornyezet in nemVedettKornyezet and
                                e.vanVedekezoFegyver in vedoFegyver and
                                e.vanJarmu  in pancelosAuto and
                                e.vanTamadoFegyver not in tamadoLoFegyver)

                               or

                               (e.vanKornyezet in nemVedettKornyezet and
                                e.vanVedekezoFegyver in vedoLoFegyver and
                                e.vanJarmu  in jarmu and
                                e.vanTamadoFegyver not in tamadoLoFegyver))


                               =>
                               e.biztonsagos = True
                               else e.biztonsagos = False
}


assert biztonsagos{
        all e: eset | Kapcsolo.isAllas = 1 => e.biztonsagos = True
}

check biztonsagos for 1
```