

TDK-dolgozat

Novák Gábor

Szegedi Tudományegyetem
Informatikai Tanszékcsoport
Szoftverfejlesztés Tanszék

Regressziós teszt szelekció Magic rendszerekben

Novák Gábor

Programtervező Informatikus (MSc) szakos hallgató

Témavezetők:

Dr. Ferenc Rudolf egyetemi adjunktus

Nagy Csaba doktorjelölt

2012

Tartalomjegyzék

1. Bevezetés	4
2. Magic, egy 4. generációs programozási nyelv	6
2.1. A nyelv felépítése	7
2.1.1. Egy Magic alkalmazás gépi reprezentációja, Magic ASG.....	9
2.1.2. Tesztesetek Magic-ben.....	10
3. Hatásanalízis 4GL nyelvekben.....	13
4. Tesztszelekció Magic/uniPaaS környezetben	15
4.1. Algoritmus	17
4.2. Implementáció.....	26
4.3. Kritériumok.....	26
4.3.1. Funkciók	26
5. Eredmények	30
6. Kapcsolódó munkák	31
7. Összegzés és jövőbeli tervek.....	33
8. Irodalomjegyzék.....	34

1. Bevezetés

A programozási nyelvek fejlődésével, a programozási hatékonyság is rohamos iramban fejlődött és mindez szükségessé tette, hogy a gyorsan elkészült munkát, gyorsan lehessen ellenőrizni, tesztelni is. Ennek a megoldását a fejlesztést végig követő tesztek írásában és hatékony futtatásában találták meg. A tesztek osztályozásának, felépítésének és szerkezetének pontos definiálása egészen 1998-ig váratott magára, amikor is kiadtak egy szabványt ennek a specifikálására. [2, 3]

Egy tesztelési folyamat legalapvetőbb építőkövének a teszteseteket szokták tekinteni. Egy teszteset a program egy futásának a leírása valamilyen adott bemenetre annak elvárt eredményével. A teszteset végrehajtásának a kimenetele tipikusan sikeres vagy sikertelen lehet (A sikertelenséget tovább lehet bontani, a sikertelenség okával). Ilyen teszt lehet például egy Szövegszerkesztő (MS Word, Open Office...) „Megnyitás” funkciójának a tesztje, ami az alábbi műveleteket tartalmazza: Program elindítás; Fájl menü/Megnyitás menüpont kiválasztása; Egy, a program számára olvasható bemeneti fájl kiválasztása és megnyitása. A teszteset sikeres, ha a megnyitás után megjelenik a kért fájl, minden más esetben a teszteset futása hibás.

Számos esetben a programozási környezetben, IDE-ben lehetőség van egy teszteset futásának a nyomon követésére, ezt pedig egy úgynevezett dinamikus futási naplóval (profile trace) teszik meg. Ekkor tulajdonképpen az történik, hogy minden egyes kiadott műveletet a futás közben a környezet rögzít, így egy jól specifikált leírást kapunk arról, hogy a futás során mit hajtottunk végre. Az előző példában a megnyitás menüpontra való kattintást egy ilyen profile-ozó eszköz például úgy jegyeztet volna fel, hogy a grafikus felület egy (x,y) pontján történt egy bal egérgomb kattintás.

A fejlesztés során rengeteg ilyen teszteset készül és egy idő elteltével szükségessé válik bizonyos tesztek újrafuttatása. Az újrafuttatandó tesztesetek meghatározása komoly feladat. Ezzel a témakörrel foglalkozik az úgynevezett teszt szelekció [4], mely segít meghatározni azt, hogy mely tesztesetek szükségesek az adott szempont szerint és melyek nem. Ez a probléma leggyakrabban az úgynevezett regressziós tesztelés [4] során merül fel. A regressziós tesztelés egyik legnehezebb feladata meghatározni azokat a teszteseteket, amelyekre hat a rendszerben elkövetet új változás, a másik pedig ezeket tesztelni. A

regressziós és szelekciós tesztek általában statikusan történnek, azaz a rendszernek nem kell futnia az elemzés végrehajtásához. A statikus elemzéshez elegendőek a futás nélkül a rendelkezésünkre álló adatok (többnyire csak a forráskód). Ezeket a teszteléssel kapcsolatos tulajdonságokat képesek vagyunk mérni is, erre a mérésre szolgál a teszt lefedettség [4]. A teszt lefedettséggel képesek vagyunk meghatározni azt, hogy a rendszer mekkora hányada van valójában a tesztek által érintve, lefedve.

Ezeknek a problémáknak a megoldására (pl. lefedettség mérésére, teszt szelekciós módszerek) 3. generációs nyelvekben már számos eszköz létezik (pl.: Squish, Appperfect). A 4. generációs nyelvekben (röviden, 4GL) azonban ezek a feladatok még váratnak magukra és megoldásuk - egy-egy nyelv sajátosságai miatt - számos olyan kihívást rejtenek magukban, amik miatt ez egy érdekes és nehéz terület.

Dolgozatomban egy ilyen, 4 GL nyelvben készült alkalmazás regressziós teszt szelekciójának az elméleti hátterét, és egy ezt magvalósító alkalmazást mutatok be. A teszt szelekció és a regressziós teszt szelekció költségbecslés szempontból nagyon fontos feladat, mivel, csak a szükséges tesztesetek újrafuttatásával súlyos összegeket lehet megspórolni. Ugyanakkor a bemutatott algoritmus ennél sokkal többre képes, mivel a regressziós teszt szelekción kívül rávilágít a tesztek lefedettségére is, és segítséget nyújt a rendszer refaktorálásában is.

Dolgozatom legfőbb eredményei:

- egy teszt szelekciós algoritmus kidolgozása és bemutatása Magic 4GL környezetben,
- az algoritmus bővítése hatásanalízis eszközökkel.

A dolgozatot későbbi felépítése az alábbi szerkezetet követi. A 2. fejezetben a Magic-et [5], napjaink egy ismert és elterjedt 4 GL programozási nyelvét, rávilágítok a nyelv erősségére és a teszt szelekció során számunkra hasznos elemekre. A 3. fejezetben a teszt szelekció általános végrehajtását majd az általam kifejlesztett algoritmust mutatom be. A 3. fejezet hátralévő részében az algoritmust megvalósító Java nyelven írt webes alkalmazást mutatom be, mely után az elért eredményeket és a jövőbeli terveket taglalom.

2. Magic, egy 4. generációs programozási nyelv

A programozási nyelvek fejlődése során számos változás történt a programok megírásának és futtatásának menetében. Az 1. generációs programozási nyelv alatt magát a gépi kódot szokás érteni (bináris 1-0 kód). Ennél magasabb szintű nyelvként, 2. generációs nyelveknek tekintik azokat a nyelveket, ahol még közvetlenül a gép számára feldolgozható kódot kell írni. Ebbe a generációba tartozik még az Assembly, mint programozási nyelv is. A mnemonikok már sok könnyítést hoztak, de még mindig gépfüggő kódokat kellett írni. A számítógépek alkalmazási területének bővülése miatt felmerült az igény, hogy a programokat minél gyorsabban írják meg a programozók. A gépi kód és az assembly nehézkessége, géphez igazodása miatt nem volt erre alkalmas, ezért megjelentek a 3. generációs programozási nyelvek is.

A 60-as évek elején jelentek meg az első magas szintű programozási nyelvek. A magas szintű programozási nyelvek 1:N típusú nyelvek, azaz egyetlen magas szintű programnyelvi utasítás több gépi kódú utasítást reprezentál. Természetesen a magas szintű nyelven írt programokat is le kell fordítani - az Assembly-hez hasonlóan - de komoly előny, hogy ezeknél a nyelveknél már sokkal kisebb a hibalehetőség, a forrásprogram sokkal áttekinthetőbb lett. Az első ilyen nyelv a FORTRAN volt, majd később számos másik követte (ALGOL, COBOL...). A későbbiekben csatlakozott a 3. generációs nyelvek családjához az Objektum Orientált programozási szemlélet és a hozzá kapcsolódó OO nyelvek (C++, Java...).

A XX. század második felében a számítástechnika forradalmi változáson ment keresztül. A hardver eszközök rohamos fejlődésére támaszkodva kezdett kialakulni a programozás során egy új szemlélet, mely az eddigi alapelvekkel szöges ellentétben állt. A 4 GL nyelvek előnyösen egyesítik a procedurális és a non-procedurális nyelvek elemeit, lehetőségeit, kihasználva mindkét nyelv előnyeit, és kiküszöbölve a hátrányok jelentős részét. A programok (programrendszerek) tervezése során a programozónak már nem a hogyan kérdésre kell válaszolnia, csak azzal kell törődnie, hogy mit szeretne megvalósítani. A legújabb programozási nyelvek esetében már nem az algoritmust, hanem az elérendő célt kell megfogalmazni, a kód generálását legtöbb esetben a fejlesztő rendszer automatikusan, a programozó elől elrejtve végzi el. Fontos jellemzője a nyelveknek, hogy nemcsak a program

szintaktikai helyességét ellenőrzik, hanem megvizsgálják a program hatékonyságát, rendszerbe illeszthetőségét is [1].

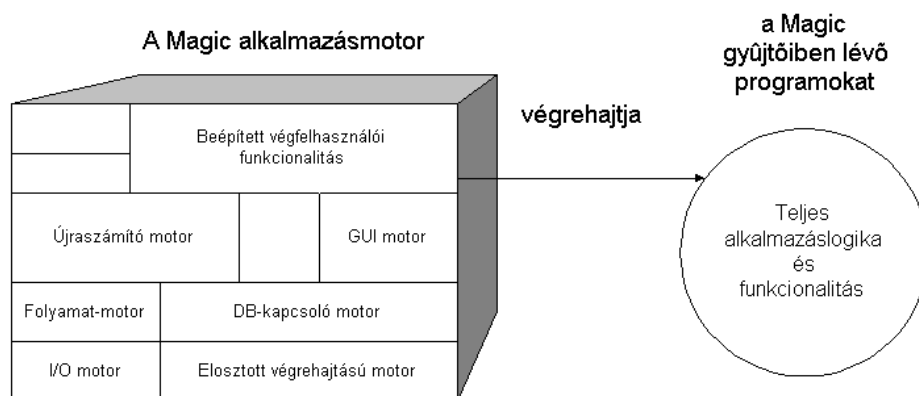
Alapvetően a 4GL nyelvek szakítanak a forráskóddal, természetesen csak a szigorú értelemben vett, kézzel írt forráskóddal. Az alap elgondolás e nyelvekben az, hogy a forráskódot már nem a programozó szerkeszti, a forráskód innentől kezdve, csak a rendszer leírására használható, de a programozó nem is láthatja, nem is szükséges tudnia, hogy van ilyen, illetve milyen szintaktikával rendelkezik. A programokat grafikus felületen kell létrehozni, az adatbázisokat, a form-okat, a hozzájuk tartozó logikai megvalósítást is mind a grafikus felületen kell beállítani. Minden komolyabb feladatot elrejt előlünk a nyelv.

A Magic már régóta létezik, eddig is számos területen alkalmazták szinte mindenhol sikerrel. A legfrissebb kiadás a uniPaaS [7] névre hallgat, ami számos új funkciót (RIA, SOA, Android alkalmazások [5]) és teljesen átszabott kinézetet ad a nyelv köré épülő IDE-nek.

2.1. A nyelv felépítése

"A Magic nagyon összetett, hordozható kliens/szerver alkalmazások építésére való nagy hatékonyságú RAD eszköz. Ez az eszköz egy egyedi megközelítésre alapszik, amely az alkalmazást az adatbázistáblában absztrakt bejegyzések sorozataként tárolja. Futás alatt a Magic motor végig lépked a fájlban, így végrehajtva az alkalmazást." [6]

A Magic rendszerek futtatása alapvetően egy jól összerakott alkalmazásmotor segítségével történik. Egy adott IDE-ben (uniPaaS) elkészítjük a kívánt rendszert, majd amennyiben futtatni szeretnénk ezt, nincs más dolgunk, mint hogy indítsuk azt. Ekkor a háttérben elindul a Magic saját alkalmazás motorja (tulajdonképpen már fut is), amely a kívánt kódot, egyszerűen értelmezi és végrehajtja. Így látszólag nem kell fordítani, linkelni a programot, hanem azonnal fut is a megfelelő változásokkal. A fordítás és linkelés fázis azonban nem szűnik meg teljesen, csupán csak a programozó elől tűnik el. Valójában készül egy bináris állomány, és ennek az állománynak a futtatása történik az alkalmazás motorban, azonban olyan jól kidolgozott ennek a fordítása és futtatása, hogy ez a programozók számára szinte láthatatlan. A Magic alkalmazásmotorjának a mechanizmusát szemlélteti az 1. ábra.



1. ábra A Magic alkalmazásmotorja [6].

A Magic esetében, az olyan fogalmak, mint a forráskód, lefordított kód, tárgykód, félig lefordított kód, pszeudokód, meta-nyelvi kód, és végrehajtható betölthető modulok, mind elhagyhatók. Ehelyett, a Magic alkalmazások „kódjának” programozó által történő bevitelének formája és a forma, ahogy ezt a Magic alkalmazásmotor számára végrehajtásra tálalja, fogalmilag megegyeznek. Ez egy egyfázisú folyamat, ahol a fejlesztő által bevitt „kód” azonnal végrehajtható a motor által, bármilyen közbülső feldolgozás nélkül.

Fontos megjegyezni, hogy a Magic csökkentette a nyelv szintaxisát, míg megtartotta a szemantika gazdagságát, így megszüntetve az interpreteres nyelvekhez kapcsolódó tipikus lassulást. A Magicben a feldolgozási idő nagy része az alkalmazás különböző funkcióinak végrehajtásával telik. Ezek a funkciók az alkalmazásmotor részeként beépítettek, és erősen optimalizált, natív kódként kerültek megvalósításra. Ugyanakkor, ezek a funkciók elegendően paraméterezettek, így lehetővé téve a fejlesztő számára az alkalmazások széles választékának a leírását.

A nyelv segítségével, a programjainkat kis építőelemekből építhetjük fel, ezek az elemek az ún. taszkok. Egy taszk rendelkezhet logikai résszel, kapcsolódhat adat objektumhoz (később) és szinte bármilyen funkciót elláthat. Fontos továbbá megjegyezni, hogy egy taszk tartalmazhat altaszkokat, így kialakítva egy fa hierarchiát, melynek a gyökér eleme maga a „főprogram”, ami szintén egy taszk.

Az adatelérést is egyszerűvé és homogénné teszi a Magic. Mindenféle adat beolvasást vagy írást egy konzisztens típuson keresztül végez, ez a típus pedig az adat objektum. Amikor a Magic programunk egy tetszőleges adathoz akar hozzáférni (fájl, adatbázis, memória, xml...),

akkor csak az ehhez az adathoz tartozó adat objektumot kell felkeresnie, így téve konzisztensé az adatok elérését.

2.1.1. Egy Magic alkalmazás gépi reprezentációja, Magic ASG

A Magic nyelv egyik nagy újítása a régebbi generációkhoz képest, hogy a forráskódot nem csak, hogy saját maga generálja és módosítja, de kifejezetten el is rejti a programozó elől. Ennek oka az, hogy a forráskódot nem a programozó által írt „egységnek” tekinti a nyelv, sokkal inkább saját magának, mint egy leírásnak készíti azt. Leveszi a kódírás terhet a programozó válláról így ettől a ponttól kezdve értelmetlen szintaktikai hibás kódról beszélni. A Magic projektek „forráskódja” mindazonáltal létezik, ha nem is annyira triviális módon, de elérhető számunkra. A forrás valójában egy XML alapú fájl halmaz, ami valójában a fejlesztett alkalmazásnak egy saját Magic belső formátumú mentése. Nem sokkal másabb ez, mint amikor például egy stratégiai játékban megnyomunk egy save gombot és minden aktuális állapotot elment a játék egy állományba. Ez a mentés az, ami felfogható egy „forrás” állománynak a Magic esetében, mert a fejlesztett alkalmazással kapcsolatban minden fontos információ ebben van eltárolva. Ezt a programozó nyilvánvalóan kézzel nem szerkeszti, számunkra viszont tökéletesen alkalmas arra, hogy statikusan elemezzük. Fontos megjegyezni, hogy az előbb leírtak miatt a Magic fejlesztők nem is szívesen hívják ezt az állományt forráskódnak, sőt, az ő álláspontjuk szerint nincs is forráskódja egy Magic alkalmazásnak, viszont számunkra a statikus analízis terminológiájának egyszerűbb használatáért forráskód névvel fogjuk illetni ezt a bináris mentést a későbbiekben.

A statikus elemző módszerek alapját leggyakrabban ún. Absztrakt Szemantikus Gráf (Abstract Semantic Graph, ASG) adja. Ettől a gráftól nem sokban tér el maga az XML szerkezet, azonban, hogy ezt belássuk, először tekintsük át az ASG gráfot és annak alapjául szolgáló AST fát.

Az Absztrakt Szintaxisfa (Abstract Syntax Tree, AST) a szintaktikai szabályok dekompozícióját tartalmazza egy fa-szerkezetben, mely nem tartalmaz „felesleges” elemeket. Tulajdonképpen az AST egy konkrét programozási nyelvre érvényes levezetési fa (a nyelv struktúrája által definiált fa) duplikációk nélkül. Az AST segítségével képesek vagyunk létrehozni a rendszer modelljét (az ASG-t). Ezt úgy tesszük meg, hogy „kidekoráljuk” a fát extra információkkal, továbbá felveszünk extra kereszt éleket, mint például függvényhívások,

típushasználat, argumentum átadás, stb. Ezen keresztül élek hatására a fa gráf szerkezetű lesz, amely rendelkezik egy a szintaxisnak megfelelő feszítőfával. Ez a gráf szerkezet lesz az Absztrakt Szemantikus Gráf, ASG, a rendszerünk modellje, melyet a statikus elemzők előszeretettel használnak [8].

A Magic forráskód számos jó tulajdonságot hordoz magában a statikus elemzés szempontjából. Például, hogy a kód, annak ellenére, hogy automatikusan generált az ember számára is viszonylag könnyen értelmezhető és ASG szerkezetű (Az XML hierarchiának köszönhetően, szinte azonnal számítható belőle az ASG). Az is fontos szempont jelen esetben, hogy automatizálható az XML fájlok elemzése (pl. DOM [9]). Az alkalmazás „kódja” az alábbi számunkra releváns elemeket tartalmazza:

- **DataSource.xml** és **DataSourceIndex.xml**: Ebben a két állományban szerepelnek az adatforrások leírásai. Minden számunkra hasznos információ a használt, adat objektumokról itt található, többek között az egyedi azonosítójuk is.
- **ProgramHeaders.xml**, **Progs.xml** és **Prg_{n}.xml**: Ezekben a fájlokban kapnak helyet a program taszkjainak a leírásai. Minden egyes taszk teljes definíciója (beleértve a bennük definiált altaszkokat is) a Prg_{n}.xml fájlban kapott helyet. Természetesen az {n}-edik taszk leírása a Prg_{n}.xml.

A ProgramHeaders.xml fájl az összes program fejlécét tartalmazza, míg a Progs.xml csak a programok azonosítóit (Az {n} értéke a Prg_{n}.xml-nél) tartalmazza azért, hogy könnyen meg lehessen keresni a programhoz tartozó taszkokat.

- A teljesség igénye nélkül, még létezik a Rights.xml, Helps.xml, ProjProps.xml, Comps.xml, Menus.xml, Models.xml. Láthatjuk, hogy a nevek önmagában elég beszédesek ahhoz, hogy ne kelljen őket kifejtetni, továbbá az elemzés szempontjából nem is relevánsak ezek az állományok.

2.1.2. Tesztesetek Magic-ben

Magic környezetben a támogatott funkciók közé tartozik a programok futásának a nyomon követése (a már korábban említett dinamikus futási napló készítés). Egy tesztelés során nagyon hasznos ez a funkció, de tekintsük át, mégis hogyan képes megvalósítani ezt is a Magic könnyűszerrel.

Az egyszerűség oka a Magic futtatómotorban rejlik. Mivel a motor hajtja végre az utasításokat, így könnyűszerrel képes megvalósítani azt, hogy a futás során egy speciális log fájlt készít arról amit éppen végrehajt, így jellemezve a rendszerben egy futást. Ezt a log fájlt nevezi a Magic **trace** fájlnek.

A trace fájlok szerkezeti felépítése elég egyszerű. Egyszerű szöveggént tárolják a különböző műveletek különböző eredményeit. A logolás mélysége bizonyos feladatok megoldásához elégtelen. Például nem jelzi, ha a felhasználó beírt valamit, így azt képtelenek vagyunk a trace fájlból meghatározni. Viszont egy egérekattintás eseményt jelez, így ez követhető.

A következő példa egy trace fájl tartalma. A futtatás során nem csináltunk semmit, csupán elindítottuk a programot majd néhány kattintást tettünk a menüben. Ennek az eredménye az alábbi trace fájl:

```
<253693576794033600>    13:50:58.406    [Action ] -    >>Starts load Batch Task -
'Main Program (Online Samples)' in Query mode (Task Instance: 1)

<253693576794033600>    13:50:58.412    [Action ] -    Ends Load Task

<253693576794033600>    13:50:58.417    [Action ] -    Starts Task Prefix

<253693576794033600>    13:50:58.422    [Action ] -    Flow - Remark:

<253693576794033600>    13:50:58.427    [Action ] -    Flow - Update: 'A: Start
Time' <== exp #1

<253693576794033600>    13:50:58.453    [Action ] -    Flow - Update: 'B: Context
Name' <== exp #2

<253693576794033600>    13:50:58.458    [Action ] -    Flow - Update: 'C: Field 1'
<== exp #3

<253693576794033600>    13:50:58.462    [Action ] -    Flow - Remark:

<253693576794033600>    13:50:58.466    [Action ] -    Ends Task Prefix

<253693576794033600>    13:50:58.472    [Action ] -    Load records

<253693576794033600>    13:51:06.475    [Action ] -    >> Processing Event    [Printer
Setup] Control Name :    [ N/A ]

<253693576794033600>    13:51:10.789    [Action ] -    >> Processing Event    [Exit
System] Control Name :    [ N/A ]

<253693576794033600>    13:51:10.794    [Info ] -    Context released :

<-1                >    13:51:10.800    [Action ] -    Starts Closing Batch Task -
'Main Program (Online Samples)' (Task Instance: 1)

<-1                >    13:51:10.805    [Action ] -    <<Ends Close Task
```

A mi szempontunkból a trace fájl minden fontos adatot tartalmaz. Mivel mi a trace fájlból csak az érintett taszkokról kinyert információkra vagyunk kíváncsiak, ezért ezeket az elemeket kell figyelni, a trace fájlban:

- **Task hívása:** Kétféle taszk létezik, így ennek megfelelően kétféle task hívás engedélyezett.

- **Online taszk:**

```
<460694066147544000> 22:52:08.280 [Action ] - >>Starts load  
Online Task - 'Main Menu' in Modify mode (Task Instance: 2, Resident)
```

- **Batch taszk:**

```
<253693576794033600> 13:50:58.406 [Action ] - >>Starts load  
Batch Task - 'Main Program (Calculator)' in Query mode (Task Instance:  
1)
```

- **Altaszk hívása:** A trace fájlban egy altaszk hívást speciális módon jelez a Magic.

- **Online/Batch altaszk:**

```
<19810432050544716> 22:51:06.287 [Action ] - Flow - Call  
SubTask :: == Create Tree
```

- **Taszk elhagyása:** A Magic a trace fájlokban jelzi, amennyiben egy taszk futása véget ért.

- **Batch taszk:**

```
<19810432050544716> 22:51:06.565 [Action ] - Starts  
Closing Batch Task - '== Write Tree' (Task Instance: 4)
```

- **Online taszk:**

```
<19810432050544716> 22:52:09.478 [Action ] - Starts  
Closing Online Task - '== Write Tree' (Task Instance: 6)
```

Egy további fontos információ a Magic trace fájljával kapcsolatban az, hogy minden egyes megkezdett művelet végrehajtásának sikerességét visszajelzi a trace. Ez azt jelenti, hogy ha elindított egy taszkot, akkor annak a műveletnek a sikerességét jelzi az alábbi sorral:

```
<19810432050544716> 22:51:06.590 [Action ] - Ends Load Task
```

Ezzel jelezve, hogy sikeresen elindította a taszkot. A taszk lezárásának a műveletét is visszajelzi számunkra a trace fájl. Ezt az alábbi sorral teszi:

```
<19810432050544716> 22:51:06.649 [Action ] - <<Ends Close  
Task
```

3. Hatásanalízis 4GL nyelvekben

A hatásanalízis [15, 16] feladata egy rendszerbeli változás hatásának, következményeinek az azonosítása. A rendszer fejlesztése során módosítások mennek végbe a már elkészült elemeket, legyen ennek az oka akár refaktoring akár egy bug javítása, akár a kódolási stílus módosítása az eredmény ugyanaz, a kód megváltozott.

A programozó, aki a változást véghezviszi sokkal több területen végzett módosítást, mint azt elsőre gondolná. A módosítás közvetlen érintett eleme a módosult kódrészlet, de a hatásanalízis során a feladatunk az, hogy azokat a rendszer elemeket meghatározzuk, amelyekre kiterjedt, kiterjedhetett a változás. Ezt a kiterjesztett elemhalmazt nevezzük hatáshalmaznak.

A feladat nem egyszerű, mivel a rejtett függőségek meghatározását magában foglalja a hatáshalmaz kiszámítása. A hatásanalízis történhet statikusan, illetve dinamikusan is. A statikus elemzés során a forráskódot tekintjük a legalapvetőbb információforrásnak és ebben az esetben a program összes lehetséges futását szemügyre kell vennünk, hogy pontos eredményeket kapjunk. Ezzel szemben a dinamikus hatásanalízis a program futásai információján alapszik, általában jobb eredményt ad egy adott futásra nézve, mint a statikus társa. A hatáshalmaz meghatározása természetesen programozási nyelvfüggő, és nagyban meghatározza a választott technikát (statikus vagy dinamikus) a rendelkezésre álló információk és azok struktúrája. Egy gráf szerkezet sokkal hatásosabb ilyen szempontból, mivel a gráfban élek behúzásával vagy utak keresésével képesek vagyunk meghatározni a kapcsolatot két elem között, legyen az akár közvetlen vagy közvetett kapcsolat. Éppen ezért, előszeretettel használják az ASG gráf reprezentációt a hatásanalízis területén. [17]

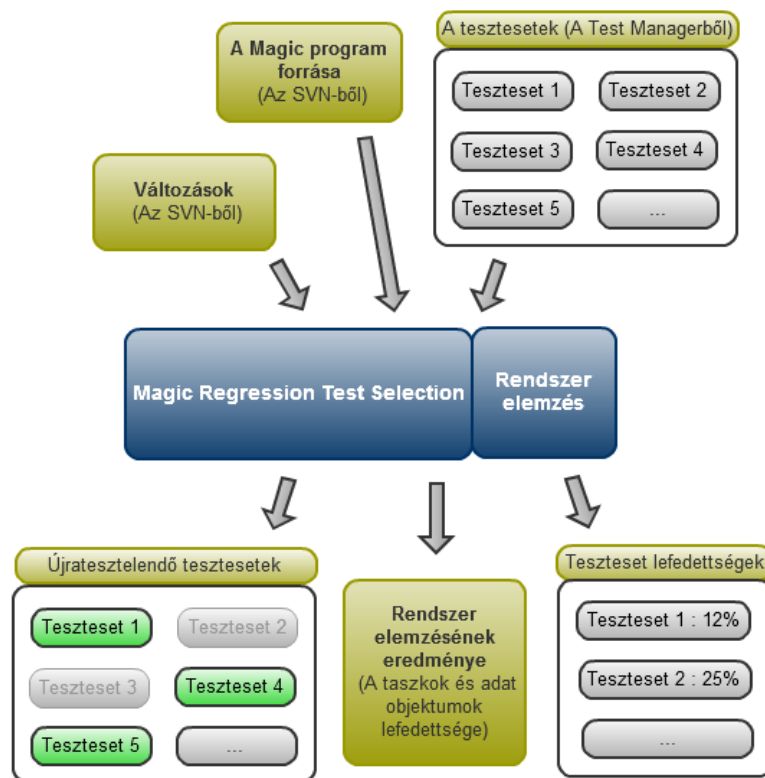
A Magic-ben a nyelv adottságait kihasználva sajátos hatásanalízis megoldásokat kell adnunk, mivel itt a forráskódot, mint a rendszert leíró fájlt nem a programozó írja, hanem az IDE generálja, módosítja szükség esetén. Ezen felül a rendszerek szerkezeti felépítése is más aspektusba kerül, mivel Magic-ben teljesen más a rendszer hierarchia, más építő elemekből épül fel a program.

Ez az új aspektus, miszerint minden taszkokból épül fel, amelyeknek konzisztens adatelérés biztosított az adat objektumokon keresztül, a legfontosabb szempont Magic alkalmazások

hatásanalízis műveleteinél. A rendszer teljes struktúrája eleve gráf szerkezetű (a taszkok egymásba ágyazhatósága és az adat objektumokkal való kapcsolatuk) ez sokban segíti az optimális hatáshalmaz meghatározását.

4. Tesztszelekció Magic/uniPaaS környezetben

Egy rendszer fejlesztése során a tesztelés az egyik legnagyobb fontosságú feladat, mégpedig azért, mert a tesztek segítségével lehet megállapítani a legkönnyebben, hogy a rendszer bizonyos részei megfelelően működnek-e. Ezt a tényt támasztja alá a manapság egyre felkapottabb úgynevezett test driven development [14] modell. Ebben a teszt vezérelt fejlesztésben a fő cél az, hogy először megalkotjuk a tesztek, majd a fejlesztést úgy hajtjuk végre, hogy a cél a tesztek hibátlan futása. Mindazonáltal bármilyen fejlesztési modellt is alkalmazunk, a fejlesztés előrehaladásával a tesztek készítése és futtatása helyett a tesztek újrafuttatása egyre fontosabb feladat. Ettől a ponttól kezdve óriási költségeket fordítanak a tesztek újrafuttatására, mely költséget egy jól megtervezett eszközzel minimálisra lehet szorítani.



2. ábra A regressziós teszt szelekció menete.

Egy ilyen eszköznek a feladata, hogy meghatározza, hogy mely tesztesetek tartoznak közvetlenül egy módosításhoz, pl. egy refaktorált kódrészhez. Ezek után nincs más dolgunk, mint a meghatározott teszteseteket újrafuttatva ellenőrizni a rendszer működését. Több tízezer teszteset esetén rengeteg teszt újrafuttatásától kímél meg minket az, ha csak a

ténylegesen szükséges teszteket futtatjuk újra. Ebben nyújt segítséget az általam ismertetett algoritmus és az algoritmust megvalósító program, melynek a működését a 2. ábra szemlélteti.

Ahogy az ábrán is látszik, a program több bemenettel és ezeknek megfelelően több kimenettel is rendelkezik. A bemenetek megadása több módon is történhet és az algoritmus szempontjából irreleváns a megadás módja. A Magic esetében a bemeneteket egy SVN repository-ból és egy menedzselő eszközből [10] nyeri.

A bemenetként kapott információk az alábbiak:

- **Tesztesetek:** A bemenetként kapott információk már teljesen szűrtek, azaz csak olyan információt kapunk, amely szükséges is, viszont ahhoz, hogy ez így is történjen szükséges tudnunk, hogy miből szűrtünk ki és micsodát. A tesztesetek fő leírásai általában trace fájlokkal történnek. Ezekről a fájlokról, már korábban esett szó. Megjegyzem, hogy a trace fájlok felül számos más információ is a rendelkezésre áll a menedzselő eszközben, de ezek közül számomra csak a teszteset egyedi azonosítója (hogy a későbbiekben lehessen konkrétan egy esetre hivatkozni) és a hozzá tartozó trace fájlokból már kinyert érintett taszkok szükségesek.
- **A projekt forrása:** A projekt forrása egy svn repository-ból áll a rendelkezésünkre. Alapvetően két nagyon fontos információt szükséges megtudnunk a projektről ebből a forrásból. Az egyik az adat objektumok leírásai. Ebből tudhatjuk meg az adat objektumok pontos specifikációit. A továbbiakban ez azért lesz fontos, mert ezekre az adat objektumokra hivatkozunk a taszkok feldolgozása során. A másik fontos információ a taszkok hierarchiája. Ez kiépítésre kerül a megfelelő fájlok feldolgozásával (Progs.xml és az összes Prog_{n}.xml). Így megkapjuk a taszkok egy gráf szerkezetbe épült hierarchiáját. Ami a legfontosabb, hogy ebben a gráf szerkezetben, már az is szerepet kap, hogy egy taszk milyen adat objektumokat használ fel. (Tekintet nélkül arra, hogy ír vagy olvas egy adat objektumot)
- **A változások:** Bemenetként kapja a program azt az információt is, hogy mely taszkokban történt változás. Valójában ez a leírás nem pontos, mivel nem konkrét taszk listát kap a program, csupán csak két svn revision-t, amelyekből meg kell

határozni azt, hogy mely taszkokban történt változás. Ez a művelet, szükséges előfeltétel a végrehajtáshoz, azonban az algoritmus számára irreleváns.

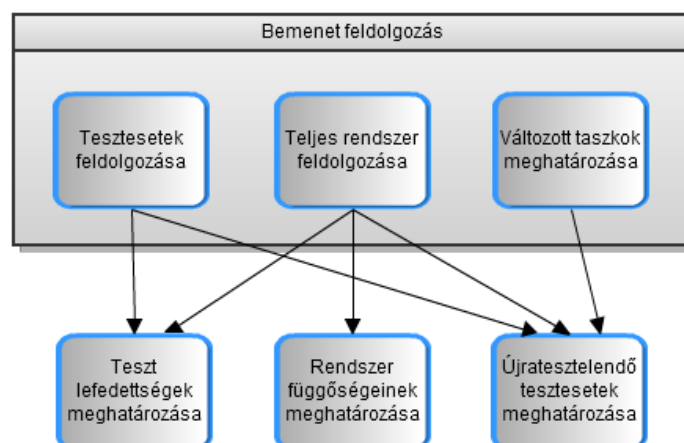
Ezen információkból a program már képes az elemzést végrehajtani, így ennél több információ megadása már felesleges lenne.

Az elkészült programnak a bemeneti információk alapján kell meghatároznia a mindenképpen újrafuttatásra szánt teszteseteket. Továbbá információkat kell, hogy nyújtson mind a rendszerről, mind a tesztesetekről. Mindazonáltal az elkészült rendszernek bizonyos kritériumokat ki kell, hogy elégítsen. Ezeket a feltételeket egy későbbi fejezetben tárgyaljuk részletesebben.

4.1. Algoritmus

Az algoritmus tervezése során az egyszerűséget kellett szem előtt tartani, ugyanis a legfontosabb művelet, a teljes projekt elemzése, egy lépésben kell, hogy történjen. Ennek az oka az, hogy a teljes projekt nagyon nagyméretű lehet, ezért szükséges ezt a vizsgálatot minél kevesebbszer, lehetőleg csak egyszer elvégezni. Ebben az egy futásban kell, hogy minden szükséges információt kinyerjen belőle a program.

Maga a feldolgozás ugyan egy lépésben történik, de a bemenetek feldolgozása is fontos a számunkra, ezért ezeket a lépéseket is külön tárgyaljuk. Ezeknek egy átfogó képét mutatja be a 3. ábra.



3. ábra A feldolgozás során végrehajtandó műveletek.

Az ábrán szereplő műveletek az alábbiak:

- **Tesztesetek feldolgozása:** Ebben a lépésben, a bemenetként kapott tesztet leíró információk feldolgozása történik. Minden tesztet egy vagy több trace fájl megadásával történik, ennek a lépésnek a feladata, hogy ezeket a trace fájlokat elemezze. Az elemzés úgy történik, hogy a fájlokat, mint egyszerű szöveges állományokat soronként beolvassuk és adott mintára illeszkedő sorokat keresünk. Ezek a mintákra illeszkedő sorok definiálják a különböző taszk hívásokat. Minden taszk hívásakor a trace fájlban egy adott szöveget tartalmazó sor kerül rögzítésre, tehát nincs más dolgunk, mint ezeket megkeresni. Miután ezeket feldolgoztuk, rendelkezésünkre áll egy halmaz, mely halmazban a tesztet által érintett taszkok szerepelnek. (Természetesen a trace fájlban egyedi névvel hivatkozik egy taszkra a Magic, ezt az egyedi nevet használja az általam készített program is az azonosításra.)

Ezt a műveletet végrehajtó metódus pszeudó kódja az alábbi:

```
processTestCases ():  
tasks = ∅  
Az összes tesztesetre  
    A tesztesethez tartozó összes trace fájlra  
        Minden trace fájlbeli sorra  
            Ha egy sikeresen megkezdett taszk vagy altaszk  
            hívás, akkor  
                taszk = taszk U aktuális taszk  
            Elágazás vége  
        Ciklus vége  
    Ciklus vége  
Ciklus vége  
Metódus vége
```

- **Teljes rendszer feldolgozása:** A teljes rendszer elemzése úgy történik, hogy az svn-ből a megfelelő XML fájlokat feldolgozva kiépül egy gráf szerkezet a taszkokból és az adat objektumokból.

Első körben megtörténik az adat objektumok feldolgozása, itt a szükséges információk az alábbiak: Egyedi azonosító (egy egész érték), fizikai név, típus. A feldolgozás természetesen a DataSource.xml fájl elemzésével történik. Ebben a lépésben kapunk egy halmazt, amely az összes adat objektumot azok pontos leírásaival tartalmazza.

A következő lépésben a Prg_{n}.xml fájlok feldolgozása történik. Ebben a lépésben minden Prg_{n}.xml fájlban szereplő taszk és azok altaszkjai bekerülnek kezdetben egy fa szerkezetbe. Itt az apa-fiú kapcsolat a taszk-altaszk kapcsolatot jelzi. A gráf úgy alakul ki, hogy minden, ebben a fában szereplő taszk összekapcsolásra kerül az általa használt adat objektumokkal. Így kapunk egy gráfot melynek minden pontja taszk vagy adat objektum. Továbbá taszkot taszkkal akkor köt össze él, ha azok egymásba ágyazottak. Taszkot adat objektummal pedig akkor köt össze él, ha a taszk használja az adat objektumot.

Ezeket a műveleteket elvégző metódus az alábbi:

```
processSystemSource ():
```

```
dataObjects =  $\emptyset$ 
```

```
tasks =  $\emptyset$ 
```

```
V = E =  $\emptyset$ 
```

Az összes dataSoruces.xml beli data object bejegyzésre

```
dataObjects = dataObjects  $\cup$  aktuális data object
```

```
V = V  $\cup$  aktuális data object
```

Ciklus vége

Az összes Prg_{n}.xml-re

A fő taszk kinyerése

```
getSubTask(fő taszk, tasks, V, E)
```

Ciklus vége

```
G = (V, E)
```

Metódus vége

```
getSubTasks (task, tasks, V, E):
```

```
tasks = tasks  $\cup$  task
```

```
V = V  $\cup$  task
```

Az összes task által használt data object-re

//Már a gráfban van, mivel már egyszer feldolgoztuk

```
E = E  $\cup$  (task - aktuális data object)
```

Ciklus vége

```
subTasks = a task összes altaszkja
```

Az összes subTasks-beli elemre

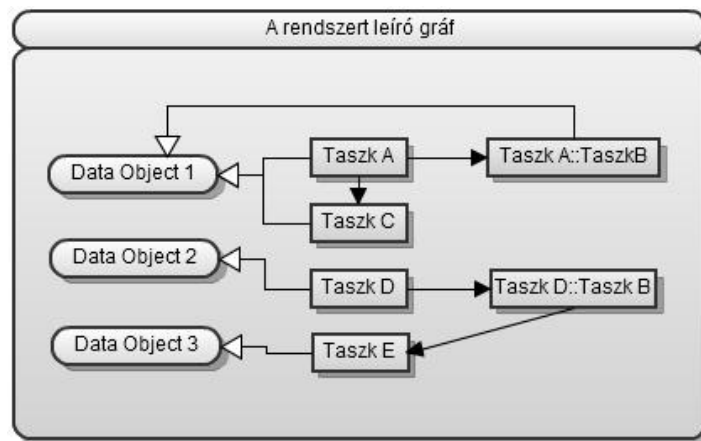
```
E = E  $\cup$  (task - aktuális altaszk)
```

```
getSubTasks(aktuális altaszk, tasks, V, E)
```

Ciklus vége

Metódus vége

Ezt a kiépült gráfot mutatja be a 4. ábra.



4. ábra A rendszerből kiépült gráf.

- **Változott taszkok meghatározása:** A két revision-t összehasonlítva képesek vagyunk meghatározni azokat a taszkokat, melyekben változás történt. A subversion parancssoros alkalmazásának a diff utasítását megfelelően paraméterezve megkaphatjuk azt a listát, amiben a módosítások egyértelműen meghatározottak. Számunkra csak azok a módosítások lesznek relevánsak, amelyek egy Prg_{n}.xml fájlt érintettek. A módosítás pontos pozíciója is szükséges, ugyanis ha csak egy altaszkban történt módosítás, akkor csak azt az altaszkot kell felvennünk a változott elemek listájára. Miután végigjártuk és beazonosítottuk az összes változást már a kezünkben van egy olyan halmaz, mely csak olyan, egyértelműen meghatározott taszkokat tartalmaz, melyekben változás történt a két svn revision között.

A következő pszeudó kód leírja a művelet viselkedését:

```
processChanges () :
```

```
tasks = ∅
```

```
Az svn-beli diff minden elemére
```

```
    Ha Prg_{n}.xml módosult, akkor
```

```
        task = Meghatározzuk a módosulás helyéhez tartozó
        taszkot
```

```
        tasks = tasks U task
```

```
    Elágazás vége
```

```
Ciklus vége
```

```
Metódus vége
```

- **Teszt lefedettségek meghatározása:** A tesztek lefedettségének meghatározása jelen esetben azt jelenti, hogy minden egyes teszthez hozzárendelünk egy értéket (százalékot), hogy a rendszer mekkora hányadát érinti.

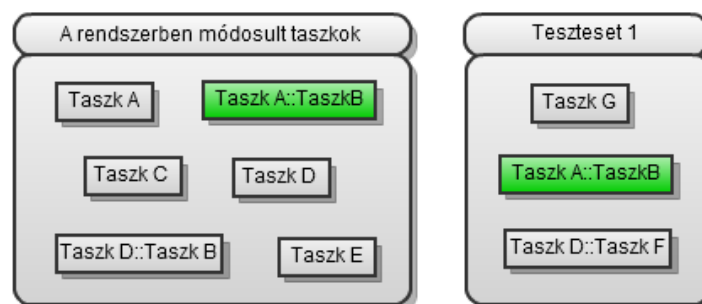
A korábbi lépésekben már meghatározott adatokat felhasználva könnyedén meg tudjuk határozni a teszt által érintett taszkok halmazának a számosságát. Jó eredményt ad, ha ezt az értéket elosztjuk a rendszerben előforduló összes taszk számával.

- **Rendszer függőségeinek a meghatározása:** Azt határozzuk meg, hogy a rendszer egyes elemei a teljes rendszer hány százalékában, mekkora részében vannak jelen.

A számítás során annyi a feladatunk, hogy minden egyes adat objektumhoz meg kell határozni, hogy melyek azok a taszkok amelyekben szerepet kap. Ezeknek a taszkoknak a számát el kell osztanunk a rendszerben előforduló összes taszk számával, így ismét kapunk egy arányt, amely leírja, hogy egy adat objektum a rendszer hány százalékában van használva.

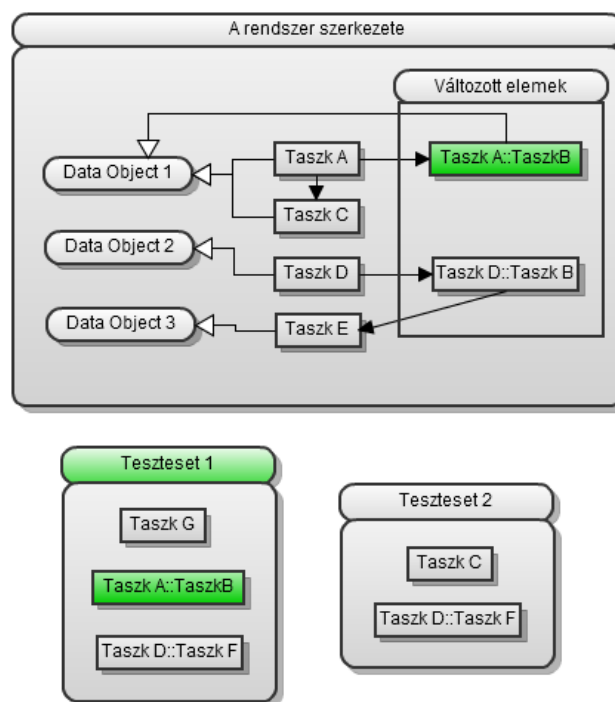
- **Újratesztelendő tesztesetek meghatározása:** Ennek a műveletnek az elvégzése a legfontosabb számunkra. A feladat érdemi részét végezzük ebben a lépésben. A tesztesetek szűrése az alábbi módon történik:

Első lépésben meghatározzuk a közvetlenül érintett teszteseteket, ez a könnyebbik feladat, ugyanis itt minden egyes tesztesethez a már meghatározott taszkok halmazának és a változott taszkok halmazának a metszetét képezzük. Ha a metszet nem üres, úgy a teszteset számunkra hasznos, így bekerül az eredmény halmazba. Az 5. ábra ezt a műveletet szemlélteti.

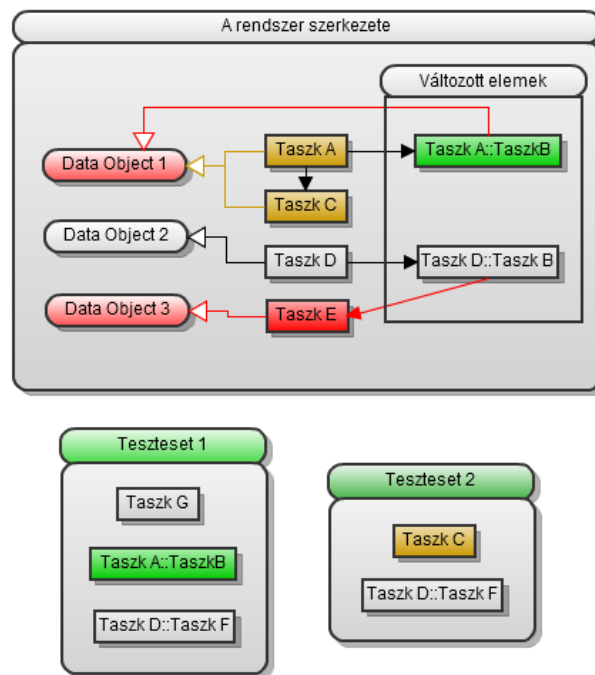


5. ábra A módosítások és tesztesetek taszkjai.

Második lépésben a közvetetten érintett elemeket határozzuk meg, ezt úgy tesszük meg, hogy az érintett taszkok halmazát egy gráfbeli lépéssel kibővítjük. Ez azt jelenti, hogy nem csak azokat a taszkokat tekintjük a tesztesethez tartozónak, amik közvetlenül változtak, hanem azokat a taszkokat is, amelyek olyan adat objektumot érintenek, amiket egy közvetlen taszk is érint. Ezt a műveletet a 6. és a 7. ábrák jól szemléltetik.



6. ábra Közvetlenül érintett tesztesetek meghatározása.



7. ábra Közvetetten, adat objektumon keresztül érintett teszteset meghatározása.

Az ábrákon jól látszik, hogy adat objektumon keresztül érintett elemek is bekerülnek a halmazba, így érjük el a hatáshalmaz elemzést. Ezt a műveletet jól kalibrálhatónak tekintjük, mivel számos beállítási lehetőség adódhat, például a hatáshalmaz keresésének a gráfbeli mélysége. (Azaz ne csak „egy lépésben” keresse a közvetett elemeket, hanem tetszőleges számú lépésben)

A következő pszeudókód bemutatja a művelet működését:

```
//Ezen a ponton már rendelkezésünkre áll:
//- a G gráf,
//- az összes teszteset (testCases) és azok érintett taszkjai,
//- a rendszerbeli módosítások által érintett taszkok
(modifiedTasks).
```

```
getRetesteredCases1():
```

```
retesterCases =  $\emptyset$ 
```

```
Az összes tesztesetre
```

```
tasks = A teszteset által érintett taszkok  $\cap$ 
modifiedTasks
```

```
retesterCases = retesterCases  $\cup$  aktuális teszteset
```

```

Ciklus vége
//Ekkor a retestedCases-ben azok a tesztek vannak amik
közvetlenül érintettek
//Most jön a második kör
changedTasks =  $\emptyset$ 
Az összes modifiedTask-ra
    changedTasks = changedTasks  $\cup$  Az aktuális módosult taszk
    changedTasks = changedTasks  $\cup$  Az aktuális módosult taszk
    G gráfbeli pontjának az 1 lépéssel való kibővítésének az
    eredmény taszkjai.
Ciklus vége
Az összes tesztesetre
    tasks = A teszteset által érintett taszkok  $\cap$  changedTasks
    retestedCases = retestedCases  $\cup$  aktuális teszteset
Ciklus vége

```

Az észrevétel, miszerint a két kör egybe is olvasható teljesen jogos. A program végleges verziójában természetesen a két művelet egyben kerül végrehajtásra, de a tesztelések miatt, a fejlesztés szakaszában a műveletet két lépésben hajtjuk végre, hogy elemezni tudjuk melyek a közvetlenül érintettek és melyek azok amelyek a hatáshalmaz bővítésének az elemei. A végleges algoritmus az alábbi:

```

getRetestedCases2():
retestedCases =  $\emptyset$ 
changedTasks =  $\emptyset$ 
Az összes modifiedTask-ra
    changedTasks = changedTasks  $\cup$  Az aktuális módosult taszk
    changedTasks = changedTasks  $\cup$  Az aktuális módosult taszk
    G gráfbeli pontjának az 1 lépéssel való kibővítésének az
    eredmény taszkjai.
Ciklus vége
Az összes tesztesetre
    tasks = A teszteset által érintett taszkok  $\cap$  changedTasks
    retestedCases = retestedCases  $\cup$  aktuális teszteset
Ciklus vége

```

4.2. Implementáció

Az algoritmust megvalósító alkalmazás elkészítése fontos, mivel így képesek vagyunk a valóságban is ellenőrizni, hogy az algoritmus valóban helyesen működik. Továbbá képesek vagyunk valós, életszerű példákon tesztelni a programot, így következtetéseket levonni a 4 GL nyelvekben írt alkalmazások regressziós teszt szelekciójának a működéséről. Az elkészült alkalmazás bizonyos kritériumoknak meg kell, hogy feleljen, mivel más eszközökkel szükséges kommunikálnia, ezért a pontos specifikálás elengedhetetlen.

4.3. Kritériumok

A tervezett rendszernek az alábbi elvárásokat kell, hogy kielégítse:

- **„Rugalmas” végrehajtás:** A rugalmas végrehajtás ez esetben azt jelenti, hogy a rendszernek alkalmazkodnia kell mindennemű bemenethez és kapcsolatot kell létrehoznia bármilyen szabványos svn repository-hoz.
- **Gyors végrehajtás:** Az elkészült programnak gyorsan kell eredményekkel szolgálnia, mivel a fejlesztés Magic környezetben kifejezetten gyors, ezért nem engedhetjük meg magunknak, hogy az elemzés sok időt emésszen fel.
- **Minimális rendszerigény:** Az alkalmazásnak a lényege, hogy bármelyik programozó a saját gépén képes legyen az elemzést végrehajtani, ezért a szükséges erőforrásokat a minimumra kell szorítani.

4.3.1. Funkciók

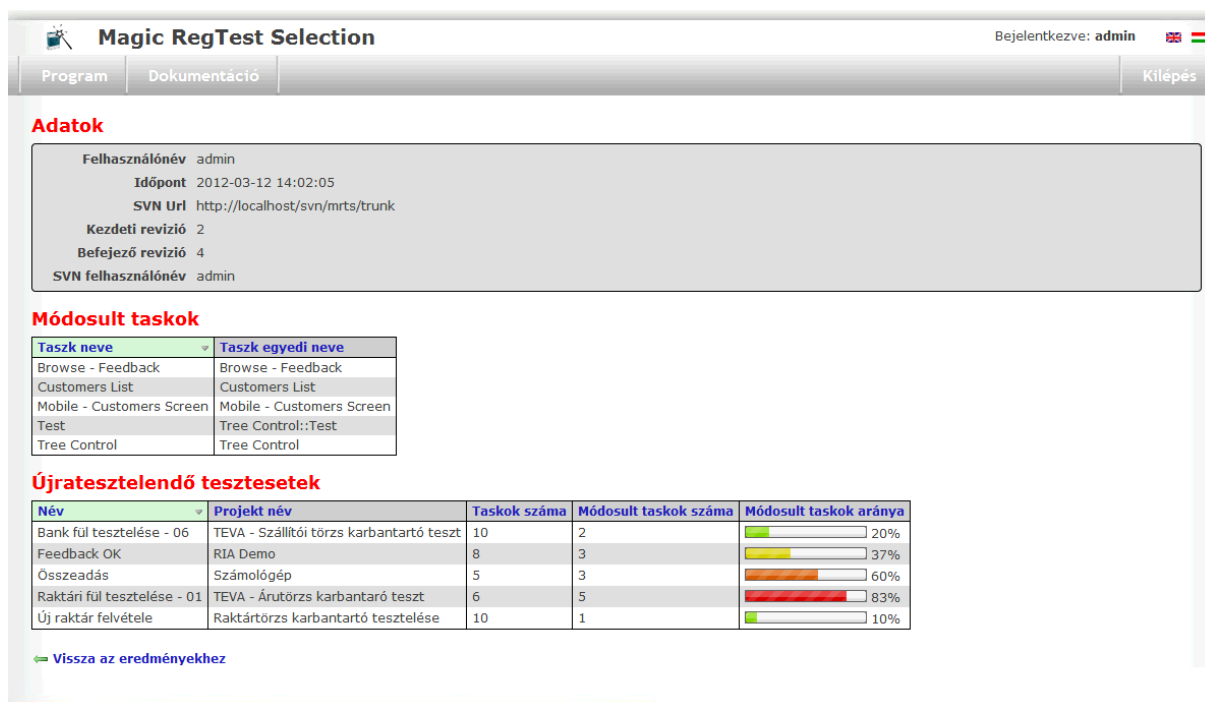
Az elkészült alkalmazás sikerrel implementálja a korábban tervezett algoritmusokat egy hatásos eszközt ad a 4 GL nyelvekben fejlesztett programok regressziós teszteléséhez. Az alkalmazás Java nyelven íródott. Az alkalmazás indítása és a bejelentkezés után a 8. ábrán látható kép fogad minket.

The screenshot shows the 'Magic RegTest Selection' application window. The title bar includes the application name and a user status indicator 'Bejelentkezve: admin' with a small Hungarian flag. Below the title bar is a navigation menu with 'Program', 'Dokumentáció', and 'Kilépés' buttons. The main content area is titled 'Változások újratestelése' in red. It contains a form with the following fields: 'SVN URL', 'Kezdeti revízió', 'Befejező revízió', 'SVN felhasználónév', 'SVN jelszó', and 'Projekt' (a dropdown menu showing '0 kijelölve'). A 'Start' button is located at the bottom of the form.

8. ábra A program bejelentkezés után.

Ezen a felületen kell megadnunk minden szükséges információt, ami a futáshoz kell. Például az SVN elérését és a hozzá tartozó felhasználónév jelszó párost, továbbá a két revision számot, amely a módosításokat tartalmazza. A tesztesetekhez kötődő trace állományokat külön felületen kell megadni. Az adatok helyes feltöltésével és az elemzés indításával három műveletet, funkciót hajtunk végre, ezek az alábbiak:

Újratesttelendő tesztesetek meghatározása: Ez a program fő funkciója, ebbe a műveletbe tartozik a változások kinyerése, a tesztesetek feldolgozása és az újratesttelendő teszteset meghatározása, megjelenítése. Ez a megjelenítés viszonylag egyszerű módon történik, a már korábban meghatározott algoritmussal azokat a tesztesetek jelenítjük meg, amelyeket szükséges újratestelni. Ennek az eredményét a 9. ábrán láthatjuk.



9. ábra Az újratesztelendő tesztesetek.

Ahogy az ábrán is látszik az eredmény a módosult taskok listája és az újratesztelendő tesztesetek. Az újratesztelendő teszteseteknél a tesztesethez tartozó információk között szerepel a tesztesetben előforduló taskok száma, illetve az ezek közül módosultak száma is. Mintegy extra információként megjelenik továbbá ezek aránya, azaz egy százalékos érték, ami leírja, hogy a teszteset taskjainak hány százaléka érintett a módosításokkal.

Az ezen felül keletkezett extra információkat a program már nem webes felületen jeleníti meg, csupán csak egy állományt készít, mely tartalmazza a további információkat. Ennek a háttérben zajló műveletvégzésnek az oka az, hogy ezek az információk nem minden esetben szükségesek, illetve hatalmas mennyiségűek lehetnek, ezért felesleges lenne ezeket megjeleníteni. Ilyen további információ például a tesztesetek lefedettsége.

Tesztesetek lefedettsége: A program egy másik kimenete, amely ugyan nem olyan látványos, mint a korábbi, de annál hasznosabb lehet. Ez az eredmény egy fájl formájában készül el, amit szükség esetén elérhetünk, és az alábbi információkat tartalmazza:

- Teszteset egyedi azonosítója.
- Teszteset rendszerrel összevetett lefedettség aránya. Ez a tesztesetben szereplő taskok számának és a rendszer összes taskjának a számának az arányára.

A rendszer elemzése: A rendszer elemzésének az eredménye szintén egy állományban kerül kiadásra, ennek oka, hogy nagyon sok információ kerülhet ebbe az állományba, továbbá az elemzés során nem is minden esetben (sőt kifejezetten ritkán) szükséges. Ebben az állományban a rendszerben szereplő adat objektumok egyedi azonosítói és nevei mellett szerepelnek a százalékos értékek. Ez az érték, a már korábban említett lefedettség, ami az adat objektumot használó taszkok száma és a rendszerben előforduló taszkok számának a százalékos aránya.

5. Eredmények

Ugyan a tesztek nagy része egy viszonylag kis projekten zajlott (3-400 taszk, 20 adat objektum, 329 teszteset) de az eredmény magáért beszél. A több mint 300 tesztesetből minimális változásokkal az újratesttelendő tesztesetek számát minimálisra csökkentette.

5 változott taszk hatására a 329 tesztet a hatáshalmaz vizsgálata nélkül 5 darabra redukálta (Ezek a közvetlenül érintett tesztesetek). Hatáshalmaz vizsgálatával az eredmény kb. 50-100 (A hatáshalmaz számítás kalibrációjától függően) érintett teszteset, amely az eredeti tesztek csupán 15-30 %-a. Ha a tesztesetek újrafuttatásához konkrét költséget rendelhetünk, úgy a költségek 70-85 %-át spórolhatnánk meg ezzel az eszközzel, amely tízezres nagyságrendű teszteseteknél óriási összegeket jelent.

Természetesen globális változások esetén (amikor a rendszer jelentős része módosult) az eredmény nem ilyen látványos, ugyanis akkor a hatáshalmaz vizsgálata miatt szinte a teljes teszteset halmazt újratesttelésre kínálja fel a program. Természetesen ez nem hiba, ilyen esetekben ez megszokott viselkedés a regressziós tesztelésnél.

6. Kapcsolódó munkák

A teszt szelekcióval, illetve a regressziós tesztekkel komoly összegeket lehet spórolni a tesztesetek futtatásán, éppen ezért ez a téma rendkívül érdekes és fontos a fejlesztő cégek számára. Számos könyv született a tesztelések, kifejezetten a regressziós tesztek kapcsán. Egy átfogó, inkább gyakorlati képet ad a szoftver tesztelés mikéntjéről a *The Art of Software Testing* [11] című könyv. Az egyik legfontosabb szempont a könyvben, hogy olyan leírást adjon, amellyel mind költségben, mind erőforrásigényben hatékony, és bármilyen tesztelési célnak megfelelően legyen lehetőségünk tesztek tervezni.

Emelie Engström és társai által írt cikk [12] témája, hogy egy alapot találjanak a későbbi kutatásokra ehhez egy szisztematikus visszatekintés található a korábban megjelent empirikus értékelésekről a regressziós teszt szelekciók technikák témaköréből. Összességében 28 technikát sorolnak föl, amivel egy regressziós teszt szelekciót ki lehet értékelni. Az eredményről minőségi elemzést is találhatunk a cikkben, amiből azt derítették ki, hogy nincs egyértelműen jobb módszer a többinél mivel az eredmények különböző tényezőktől függnék. Ez a cikk a már meglévő regressziós tesztelési módszereket összesíti és próbálja rangsorolni.

Egy nagyon érdekes eredményt publikáltak Mary Jean Harrold társaival 2001-ben [13]. A cikk témája, hogy hogyan működik a Java nyelven készült programok biztonságos regressziós tesztelése. Olyan regressziós teszt szelekciós módszert mutattak be, amellyel akár félkész Java szoftverek tesztelése is lehetővé válik. A módszer megbízható abban az esetben is, hogyha a program külső könyvtárakat is használ. Emellett a cikk bemutat egy RETEST nevű algoritmust, amely egy regressziós teszt szelekció algoritmus, mely hatásosan csökkenti a teszt környezet méretét. Ez a megoldás hasonló az ebben a dokumentumban bemutatotthoz, viszont a Java-ra, mint 3GL nyelvre épít.

A [18] cikkben egy, az ehhez a dokumentumhoz igen hasonló eredményt publikáltak. Ebben a cikkben a rendszer vezérlési folyamat gráfját [19] használják az regressziós teszt szelekció megvalósításához. A publikált algoritmus nagy előnye, hogy bármilyen 3GL programozási nyelvben és bármilyen módosítást képes követni és elemezni.

A 4 GL nyelvek közül a Magic specifikus tesztelést támogató eszközök között az általam kidolgozott módszerhez hasonlót más még nem készített, ugyanakkor a SAP Integrált vállalatirányítási rendszer saját 4GL nyelvének az ABAP-nak léteznek teszteléssel foglalkozó szakirodalmi. Az egyik, ezt a témát feltáró szakirodalom a *Testing SAP Solutions* című könyv [20].

7. Összegzés és jövőbeli tervek

A dokumentumban bemutatott alkalmazás sikerrel támogatja a Magic nyelvben készült alkalmazások regressziós tesztelését. A teszt szelekción kívül számos segítséget nyújt mind a tesztesetek rangsorolásában és lefedettség mérésének, mind a rendszer struktúrájának a refaktorálásában, így az eszközzel, egy komplex, 3 területre kiterjedő elemzőt kapunk.

Az eszköz maga, a viszonylag egyszerű algoritmusával, mely a Magic forráskód egyszerűségének köszönhető, és a gyors végrehajtással egy új fegyvert ad a Magic fejlesztők kezébe, amivel még gyorsabban, még költség kímélőbben lehet alkalmazásokat fejleszteni.

Az eszközt egyszerűen, gyorsan lehet alkalmazni bármilyen olyan területen, melyre Magic nyelvű alkalmazás fejlesztését szánják. Annak köszönhetően, hogy az alkalmazás a legtöbb verziókövető rendszert támogatja, a fejlesztők számára semmilyen hátrányt vagy megkötést nem jelent. Mivel a fejlesztések túlnyomó többsége verziókövető rendszerekben történik ezért az alkalmazás tökéletes erre a célra. Az eszköz hátrányaként a rendszeres futtatás szükségességét lehetne felsorolni. Mivel sok változás esetén az algoritmus maga azt a eredményt adja, hogy szinte az összes tesztesetet újrafuttatásra kínálja fel, ezért a megoldás erre az, hogy rendszeres időközönként kell futtatni a regressziós teszt szelekciót, így a kevesebb változást sokkal hatékonyabban képes követni az alkalmazás.

A jövőbeli tervek közé a nagyobb Magic projekteken való tesztelés és a 4 GL nyelvek között „univerzálissá” tétel szerepel. A legfőbb cél természetesen egy nagyon hasznos regressziós teszt szelekciós eszközt adni a fejlesztők kezébe, mellyel gyorsabban, pontosabban, költség hatékonyabban lehetne 4 GL-ben készült rendszereket fejleszteni.

8. Irodalomjegyzék

- [1] Rico, DF; HH Sayani, RF Field (2008). "History of computers, electronic commerce and agile methods". Advances in Computers (Academic Press) 73: Emerging Technologies.
- [2] "IEEE Standard for Software Test Documentation," IEEE Std 829-1998 , vol., no., pp.i, 1998 doi: 10.1109/IEEESTD.1998.88820
- [3] British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST) Standard for Software Component Testing 1997 BS7925-2
- [4] Kapros Gábor, Beszédes Árpád, Dévényi Gábor, Dusha Kiss Gábor, Kovács Attila, Balla Katalin, Csonka Béla György - Hungarian standard glossary of terms used in Software Testing, ISTQB Glossary - Hungarian V3.1, 2011
- [5] Magic Software Enterprises: The 5 New Essentials for Building Business Applications (White Paper)
- [6] <http://irokez.szegedsw.hu/szegedswdoc/index.php?section=6&mode=browse&helyeske>
- [7] Magic Software Enterprises: uniPaaS Technology Overview (White Paper)
- [8] Mamas, E.; Kontogiannis, K.; , "Towards portable source code representations using XML," Reverse Engineering, 2000. Proceedings. Seventh Working Conference on , vol., no., pp.172-182, 2000, doi: 10.1109/WCRE.2000.891464
- [9] <http://www.w3schools.com/dom/default.asp>
- [10] Daniel Fritsi, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimothy. A Layout Independent GUI Test Automation Tool for Applications Developed in Magic/uniPaaS. In Proceedings of the 12th Symposium on Programming Languages and Software Tools , Tallinn, Estonia, October 5-7, 2011.
- [11] The Art of Software Testing, 2nd Edition Glenford J. Myers, Corey Sandler, Tom Badgett, Todd M. Thomas, ISBN: 978-0-471-46912-4

- [12] Emelie Engström, Per Runeson, Mats Skoglund, A systematic review on regression test selection techniques, *Information and Software Technology*, Volume 52, Issue 1, January 2010, Pages 14-30, ISSN 0950-5849, 10.1016/j.infsof.2009.07.001.
- [13] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression test selection for Java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01)*. ACM, New York, NY, USA, 312-326. DOI=10.1145/504282.504305
- [14] Beck, K. *Test-Driven Development by Example*, Addison Wesley, 2003, ISBN 978-0321146533
- [15] Bohner, S.A. and R.S. Arnold, Eds. *Software Change Impact Analysis*. Los Alamitos, California, USA, IEEE Computer Society Press. 1996
- [16] Kilpinen, M.S.. *The Emergence of Change at the Systems Engineering and Software Design Interface: An Investigation of Impact Analysis*. University of Cambridge. Cambridge, UK. 2008
- [17] Richard J. Turver, Malcolm Munro, *An early impact analysis technique for software maintenance*, 2006, DOI: 10.1002/smr.4360060104
- [18] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (April 1997), 173-210. DOI=10.1145/248233.248262
- [19] O. Shivers. 1988. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI '88)*, R. L. Wexelblat (Ed.). ACM, New York, NY, USA, 164-174. DOI=10.1145/53990.54007
- [20] Markus Helfen, Hans Martin Trauthwein, *Testing SAP Solutions*. 2007. ISBN: 1592293662