Optimalizációs és visszatervezési technikák kiértékelése adat-intenzív rendszereken

Nagy Csaba

Szoftverfejlesztés Tanszék
Szegedi Tudományegyetem

Témavezető: Dr. Gyimóthy Tibor

Ph.D. értekezés tézisei

Szegedi Tudományegyetem
Informatika Doktori Iskola

Szeged, 2013 December

1. Bevezetés

Napjaink információs rendszerei már nem egyszerű alkalmazások, amikkel egy-egy fontosabb feladatot oldunk meg. Ma már hatalmas méretű, összetett architektúrájú rendszerekkel dolgozunk, amik részei a mindennapjainknak, ott vannak a táblagépeinken, okos telefonjainkon, mindenhol. Ezeknek a rendszereknek a célja, hogy a helyes információt a megfelelő embereknek pontos időben és formában juttassák el [15].

Pawlak 1981-ben megjelent cikkében ír a Varsói Egyetem Információs Rendszerek Munkacsoportjának eredményeiről [14]. Tanulmányában bemutat egy információs rendszert, ami egy könyvtári rendszer és mintegy 50.000 dokumentumot kezel. Azóta az információs rendszerek rengeteget fejlődtek, és a kezelt adatmennyiség is jelentősen megnőtt. Ismerünk olyan rendszereket a rádiócsillagászatból, amik napi 138 PB (peta byte) adatot kezelnek [16]. Jól ismert a részecskefizika világából a CERN 2008-ban átadott Large Hadron Collider részecskegyorsítója is, ami másodpercenként 2 PB adatot kezel [7]. Az ilyen rendszereket a jelentős adatterhelés miatt *adat-intenzív rendszerek*nek nevezzük [2, 10–12].

A hatalmas mennyiségű adat, amit az adat-intenzív rendszerek kezelnek, általában egy *adatbázis*ban kerül eltárolásra, amit egy *adatbázis-kezelő rendszer* (*database management system*, DBMS) kezel valamilyen adat *séma* szerint rendszerezve. *Relációs DBMS*-ekben (RDBMS) ez a séma táblákat tartalmaz, amik általában egy entitást jelölnek különböző tulajdonságokkal, amiket a tábla oszlopai tárolnak.

Az ilyen rendszerek karbantartásának támogatására több módszert is kidolgoztak mind a forráskód, mind pedig az adatbázis elemzésének segítségével is. Kevés olyan módszer van viszont, ami valóban figyelembe veszi az adat-intenzív rendszerek sajátosságait (pl. adatelérésen keresztüli függőségek vizsgálata). Ahogy Cleve et al. megjegyzik azt az adat-intenzív rendszerek evolúcióját vizsgáló tanulmányukban [3]: „*mind a szoftver, mind az adatbázis rendszerek fejlesztői keresik a megoldásokat a szoftver evolúció problémáira. Mégis, meglepően kevés kutató munka vizsgálja a két területet együttesen, ahol a szoftver és az adat találkozik.*”.

1.1. Tézis célkitűzései

Jelen tanulmányban adat-intenzív rendszerek visszatervezési módszereit vizsgáljuk statikus elemzési módszerekkel. Olyan módszerekkel foglalkozunk, amik a Cleve et al. által is felvetett módon, a szoftver és az adat komponensek együttes vizsgálatával nyernek ki rejtett kapcsolatokat adat-intenzív rendszerekből. A kinyert információ segítségével megoldást keresünk adat-intenzív rendszerek architektúrájának feltérképezésére; egy speciális negyedik generációs nyelvben, Magicben fejlesztett alkalmazások minőségbiztosítására; input adat okozta biztonsági hibák felderítésére; valamint információs rendszerek optimalizálására lokális refaktoring műveletek segítségével. A bemutatott módszerekkel nagyméretű, ipari rendszereket elemzünk, egyebek mellett egy több, mint 4 millió soros banki rendszer esettanulmányát is bemutatjuk, ahol a rendszer architektúra térképét állítjuk elő automatikus eszközökkel, illetve minőségproblémákat tárunk fel benne.

Az alábbi kutatási kérdésekre keressük a válaszokat:

1. Lehetséges-e automatikus forráskód elemzési módszerekkel, adateléréseket vizsgálva, információt kinyerni, ami segíthet egy adat-intenzív rendszer architektúrájának feltérképezésében?

2. Adaptálható-e egy harmadik generációs nyelvekhez kifejlesztett automatikus elemzési módszer egy negyedik generációs nyelvre, mint amilyen a Magic? Amennyiben igen, úgy lehetséges-e statikus kódelemzéssel támogatni egy Magic alkalmazás újabb verzióra történő migrálását?

3. Hatékonyan használhatóak-e a vezérlési folyam és adatfolyam elemzések a felhasználói input okozta biztonsági hibák felderítéséhez?

4. Milyen mértékben lehetséges csökkenteni kód faktoring algoritmusok segítségével egy fordító által előállított binárisok méretét?

Az elért eredményeinket hat tézispontban foglaljuk össze, amelyek az alábbiak:

I Örökölt, adat-intenzív rendszerek architektúrájának visszatervezése

(a) Architekturális függőségek feltérképezése adat-intenzív rendszerekben

(b) Nagyméretű, örökölt rendszerek architekturális problémáinak vizsgálata

II A Magic világa

(a) Magic alkalmazások visszatervezését támogató elemzőcsomag kifejlesztése

(b) Új komplexitás metrikák definiálása és kiértékelése Magic rendszereken

III Biztonsági elemzés és optimalizálás

(a) Felhasználói input okozta biztonsági hibák felderítése

(b) Információs rendszerek optimalizálása: kód faktoring a GCC fordítóban

1.2. Publikációk

A tézisben felhasznált publikációk jelentős része a szakma rangos, nemzetközi konferenciáinak kiadványaiban, valamint folyóirataiban került közlésre. A tézispontok és a publikációk kapcsolatát összegzi az 1.1. táblázat.

	Tézispont	Publikációk
I/a.	Architekturális függőségek feltérképezése adat-intenzív rendszerekben	[23]
I/b.	Nagyméretű, örökölt rendszerek architekturális problémáinak vizsgálata	[20, 25]
II/a.	Magic alkalmazások visszatervezését támogató elemzőcsomag kifejlesztése	[18, 24, 27]
II/b.	Új komplexitás metrikák definiálása és kiértékelése Magic rendszereken	[26]
III/a.	Felhasználói input okozta biztonsági hibák felderítése	[21]
III/b.	Információs rendszerek optimalizálása: kód faktoring a GCC fordítóban	[19, 22]

1.1. táblázat. Tézispontok és a publikációk kapcsolatának összegzése.

2. Örökölt, adat-intenzív rendszerek architektúrájának visszatervezése

Ebben a fejezetben olyan elemzési módszereket ismertetünk, amik azt használják ki adat-intenzív rendszerekben, hogy az architektúra középpontjában egy adatbázis kezelő rendszer van.

Először ismertetjük azt a módszert, amivel forráskódelemek és adat táblák közötti kapcsolatokat (*Create-Retrieve-Update-Delete*, *CRUD* függőségek) térképezünk fel beágyazott SQL utasítások elemzésével. A kapcsolatok tanulmányozásával biztonságos relációkat keresünk, pl. hatásanalízis vagy architektúra rekonstrukció céljából. Ezt követően egy esettanulmányban szemléltetjük, hogy a kinyert kapcsolatok hogyan használhatók egy rendszer architektúrájának feltérképezésére. A tanulmányban egy nagyméretű, örökölt Oracle PL/SQL rendszert elemzünk először bottom-up megközelítésben a kapcsolatok kinyerésével, majd top-down megközelítésben a fejlesztőket interjúztatva.

2.1.1. Beágyazott SQL utasítások kinyerése a forráskódból

Egy RDBMS-sel általában SQL utasításokkal kommunikálunk az alkalmazás oldaláról, egy library segítségével, mint például a JDBC. Napjainkban az ORM technológiák (pl. Hibernate) is egyre elterjedtebbek, alacsony szinten viszont ezek is SQL lekérdezéseket küldenek az adatbázis felé. Sok visszatervezési módszer épít ezért a forráskódba beágyazott SQL utasítások kinyerésére.

Tanulmányunkban [24] egy olyan módszert ismertetünk, aminek a segítségével egy speciális procedurális nyelvből, ForrásSQL-ből nyerhetünk ki beágyazott SQL utasításokat. Ez a programozási nyelv, olyan információs rendszerek fejlesztéséhez lett kifejlesztve, amik szoros kapcsolatban állnak egy adatbázissal. A forráskódban ezért gyakran fordulnak elő beágyazott SQL utasítások, amik adott eljárások segítségével küldhetőek el az adatbázisnak, hasonlóan, mint ahogy a JDBC esetében is. Az általunk bemutatott módszer ezért könnyen általánosítható lehet más procedurális nyelvekre is, annak ellenére, hogy ForrásSQL-re lett kifejlesztve.

A bemutatott módszer azon az egyszerű megfigyelésen alapszik, hogy azok az utasításrészletek, amik a string műveletekkel összeállított SQL utasításokban nem ismerhetőek fel, egyszerűen helyettesíthetőek a fel nem ismert utasításrészletet tartalmazó változó nevével. Ha például a beágyazott utasítás valamely részletét a `name` változóból kapjuk meg, akkor az utasításban a változó helyén '@@name@@' string kerül behelyettesítésre. Az általunk kifejlesztett SQL parser az ilyen utasításrészleteket speciális azonosítókként kezeli, biztosítva ezzel az SQL utasítás szintaktikai elemzését. Egy ilyen kinyert SQL utasításra látható egy példa a 2.1. ábrán.

Ennek az egyszerű ötletnek a segítségével beazonosítjuk azokat az utasításokat a forráskódban, ahol SQL utasítást küldenek az adatbázis felé, és megpróbáljuk minél hatékonyabban felépíteni az ott beágyazott utasítást. Azoknak a változóknak a helyén, amiknek a tartalmát nem tudjuk kinyerni, a korábban ismertetett behelyettesítést alkalmazzuk. Valahányszor az SQL elemző számára elemezhető utasítást kapunk, az a módszernek köszönhetően meg fogja őrizni az eredeti utasítás fő tulajdonságait (utasítás típusa, elért táblák, oszlopok).

```
name=readString();
sql="SELECT firstname, lastname " +
   "FROM customers " +
   "WHERE firstname " +
   "LIKE('%" + name + "%')";
executeQuery(sql);
```

(a)

```
SELECT firstname, lastname
  FROM customers
  WHERE firstname
    LIKE('%@@name@@%');
```

(b)

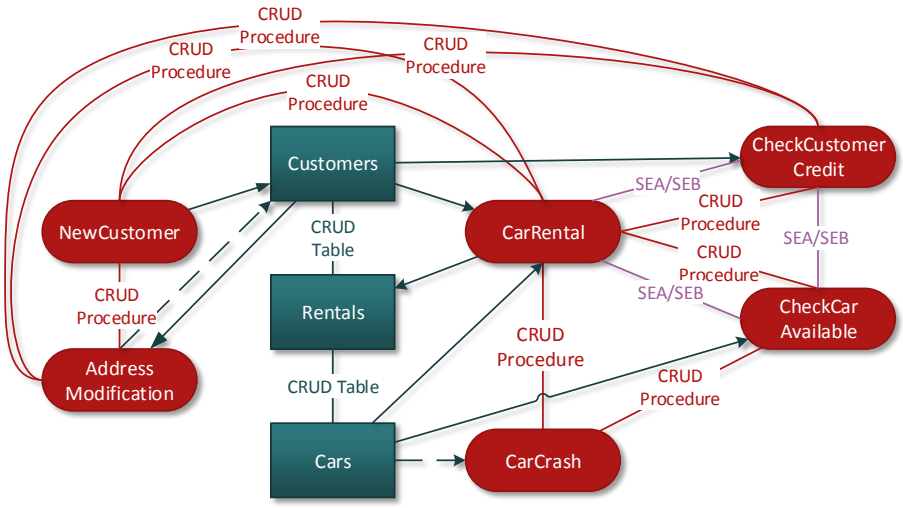
2.1. ábra. Egy minta kódrészlet (a) egy beágyazott SQL utasításról és (b) a kinyert SQL utasításról, amiben a LIKE paramétere egy változó helyettesítéséből származik.

ForrásSQL esetében azt figyeltük meg, hogy a fejlesztők szeretik az adatbázisnak küldendő utasítást az elküldés helyéhez közel összeállítani. A módszerünk ezt kihasználva, először megpróbálja a változók értékét meghatározni a korábbi értékadásokon keresztül, vissza-vissza lépve a vezérlési folyamban. Amennyiben nem sikerül az értéket meghatározni, a változó nevét a korábban ismertetett módon helyettesíti.

A módszer előnye, hogy kis számításigénnyel implementálható. Persze számos olyan eset előfordulhat, ahol összetettebb módszerrel a beágyazott SQL utasítás pontosabban kinyerhető lenne. Egy ForrásSQL rendszeren vizsgálva mégis nagyon meggyőző eredményt értünk el: a kódban összesen $7,434$ ponton küldtek SQL utasítást az adatbázis felé, amiből $6,499$ SQL utasítást sikerült feldolgozni, 87%-ban kinyerve ezzel a beágyazott SQL utasításokat.

2.1.2. Adatelréseken keresztül fellépő függőségek adat-intenzív rendszerekben

A beágyazott SQL utasítások segítségével, adatelréseken keresztül fellépő, rejtett kapcsolatokat (*Create, Retrieve, Update, Delete*; röviden *CRUD* kapcsolatok) vizsgálunk, amihez egy ún. *CRUD* mátrixot állítunk elő. A *CRUD* mátrixot korábban sikerrel használták a kód megértését, illetve minőségét vizsgáló elemzési módszerekben [1, 17]. Mi a *CRUD* mátrixot forráskódelemek közötti kapcsolatok feltérképezéséhez használjuk [24]. A mátrix egyébként szemléltethető egy gráffal is, amire egy minta látható a 2.2. ábrán.



2.2. ábra. *CRUD* és *SEA/SEB* kapcsolatok táblák és eljárások között.

A módszert egy ForrásSQL rendszer eljárásai és adattáblái közötti *CRUD* kapcsolatok feltérképezésével vizsgáljuk, majd vetjük össze *SEA/SEB* kapcsolatokkal [6]. A 2,936 eljárást és 317 táblát tartalmazó rendszerben megmutatjuk, hogy mind a *CRUD*, mind a *SEA/SEB* által kinyert kapcsolatok kiegészítik egymást, ezért olyan elemzésekkor, amikor biztonságos módszerek kellenek (pl. hatásanalízis) mindkét kapcsolattípus használata javasolt lehet.

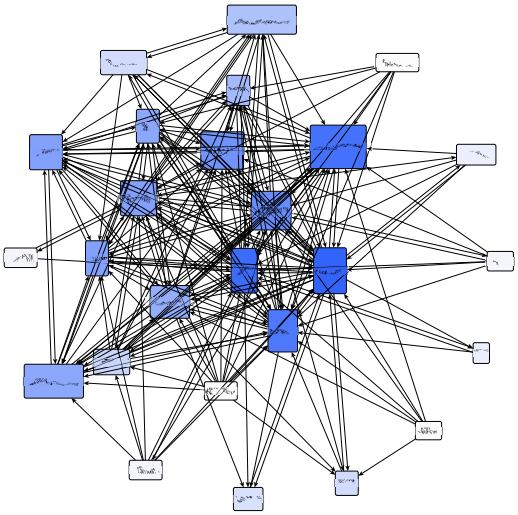
2.1.3. Saját hozzájárulás

A bemutatott SQL kinyerési algoritmus és a *CRUD* kapcsolatok kinyerésének módszere, valamint az elemzés végrehajtása és az eredmények kiértékelése a szerző saját hozzájárulása. A szerző munkájának nagy része továbbá az elemzés alapjául szolgáló MS SQL és Transact SQL séma és nyelvi elemző megtervezése és kidolgozása [23]. A ForrásSQL kódbázis elemzéséhez a Columbus elemzőcsomag ForrásSQL elemzőjét használtuk, amit a szerző egészített ki a *SEA/SEB* kapcsolatok számításáért felelős komponenssel. A tanulmány megjelenését követően Liu et al. a módszert használva hasonló eljárást dolgoztak ki PHP rendszerekre [8].

2.2. Nagyméretű, örökölt rendszerek architekturális problémáinak vizsgálata

Egyik ipari partnerünk azzal keresett meg minket, hogy segítsünk nekik a nagyméretű adatbázis rendszerük karbantartási problémáiban. A cégnél egy Oracle PL/SQL rendszert tartottak karban, ami az évek alatt egy több, mint 4.1 millió soros adatbázis dumppal (csak a nem-üres és nem-komment, adatbeszúrásokat nem tartalmazó sorokat számítva) rendelkező rendszerré nőtt.

A rendszerről először egy architektúra térképet készítettünk. A fejlesztőkel folytatott interjúk során beazonosítottunk felsőszintű komponenseket és közöttük lévő kapcsolatokat, majd az alacsony szinten, forráskódelemzéssel kinyert kapcsolatokat emeltük fel a komponensek szintjére. A végeredményben előálló, a komponensek



2.3. ábra. Kapcsolatok egy nagy adat-intenzív rendszer felső szintű komponensei között. Az ábra jól mutatja, hogy az évek során ad-hoc módon fejlődött rendszer architektúrája teljesen átláthatatlan; a 26 meghatározott komponens mindegyike szinte minden másikkal kapcsolatban áll. (A neveket szándékosan eltorzítottuk.)

kapcsolatát mutató architektúra diagramon jól látható, hogy a meghatározott 26 komponens között szinte minden mindennel kapcsolatban van, a rendszer architektúrája teljesen ad-hoc módon fejlődött az évek során. Az elemzés egy másik eredményeként olyan adatbázis objektumokat azonosítottunk be, amelyeket már nem használtak, vagy logikailag rossz komponensbe soroltak be.

A függőségek meghatározása segített továbbá egy olyan komponens eltávolításában, amit már törölni akartak a rendszerből, mert azóta újraimplementálták Java nyelven. Az elemzés segítségével olyan kapcsolatokra mutattunk rá, amiket még nem szüntettek meg a komponens eltávolításához. Az architekturális problémák mellett statikus elemzőeszközökkel konkrét kódolási problémákat és copy&paste kódrészleteket beazonosítunk.

2.2.1. Saját hozzájárulás

Az Oracle PL/SQL rendszerek elemzéséhez a Columbus rendszert Oracle PL/SQL elemzővel kellett bővítenünk. A szerző munkája volt meghatározó az Oracle PL/SQL séma és elemző kidolgozásában, az architektúra térkép visszatervezési módszerének kidolgozásában, és a nem használt komponens kapcsolatait feltérképező módszer kidolgozásában is. A szerző végezte el továbbá az esettanulmányban használt elemzéseket és interjúkat a fejlesztőkkel. Az egyéb eredmények a társszerzőkkel végzett közös munka eredményei [20, 23].

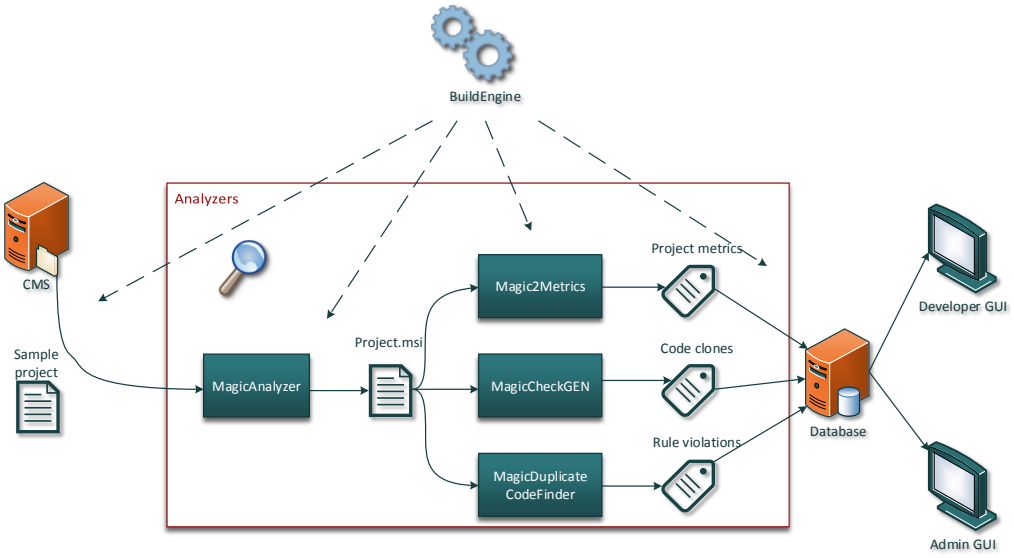
3. A Magic világa

Ebben a fejezetben azt vizsgáljuk, hogyan adaptálható a Columbus módszertan Magic-re, mint egy speciális negyedik generációs programozási nyelvre. Egy teljes elemző csomag kifejlesztése volt a célunk, ami Magic alkalmazások minőségbiztosítása mellett, a korábban ismertetett architekturális függőségek kinyerésére is képes.

Ismertetjük, hogyan adaptáljuk a Columbus módszertant Magic nyelven fejlesztett alkalmazások elemzéséhez. Megmutatjuk, hogy a harmadik generációs nyelvekhez fejlesztett elemzési technikák (pl. minőségmérések, architekturális információk kinyerése) 4GL környezetben is segítik a fejlesztők munkáját. A módszertan adaptálása közben szembesültünk azzal, hogy a fejlesztők nem ugyanazokat a nyelvi elemeket találják komplexnek, mint amiket az adaptált metrikák mutatnak. Ezért egy új komplexitás metrika bevezetésére tett kísérletet is ismertetünk.

A negyedik generációs nyelveket (4GL) gyakran nagyon magas szintű nyelveknek is hívják. A fejlesztők, akik ilyen nyelven fejlesztenek, nem írnak a hagyományos értelemben vett forráskódot, hanem egy magasabb absztrakciós szinten, gyakran egy alkalmazás generátorban állítanak össze egy programot.

A Magic egy tipikus 4GL, amit a Magic Software Enterprises vezetett be a 80-as évek elején, mint egy innovatív technológiát, ahol egy meta-model segítségével lehet alkalmazást készíteni. Üzleti alkalmazások fejlesztéséhez tervezték, amiknek a fejlesztése erősen adatbázis központú. Ezzel együtt a nyelv legtöbb eleme is adat entitásokhoz kapcsolódik: egy adattábla mezői közvetlenül változókon keresztül érhetőek el, amiket task-ok kérdeznek le vagy módosítanak. Emiatt az adat-központúság miatt a Magic alkalmazások is adat-intenzív rendszerek.



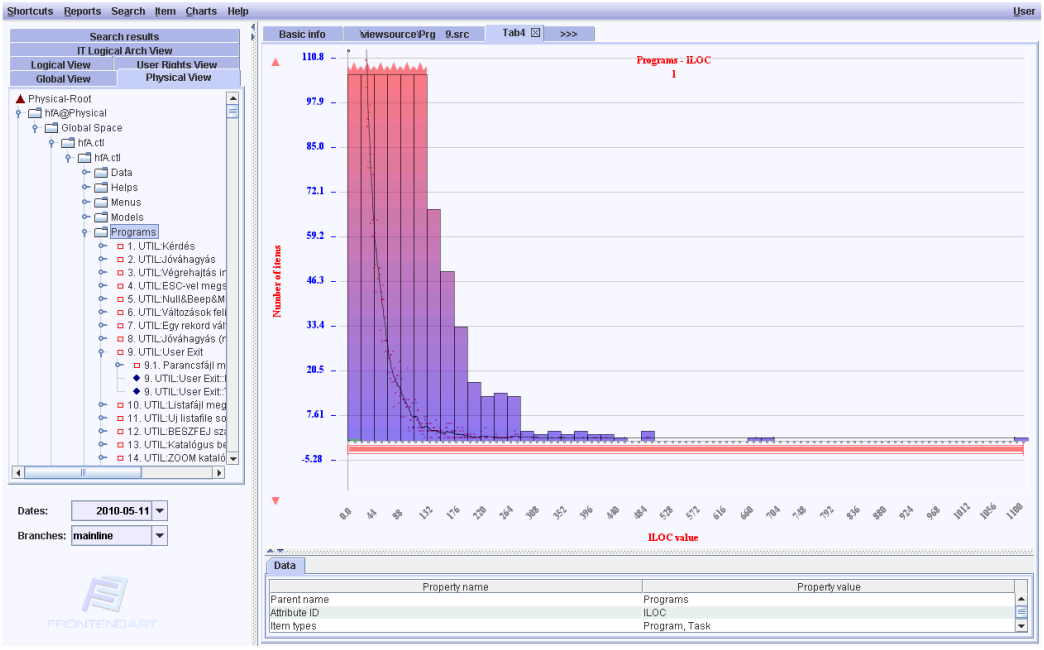
3.1. ábra. Columbus módszertan adaptálva Magic környezetben.

Egy ipari partnerünkkel, a SZEGED Szoftver Zrt.-vel, közösen azt kutattuk, hogy a Columbus módszertan adaptálható-e Magic alkalmazások visszatervezésére. A célunk az volt, hogy Magic rendszerek minőségbiztosítására [24], valamint migrálásának támogatására adjunk statikus elemzéssel automatikus megoldásokat [27]. A teljes Columbus módszertant implementáltuk Magic rendszerekre a nyelvi elemzéstől, a metrikák számításán át a kódolási problémák és architekturális nézetek kinyeréséig (3.1. ábra).

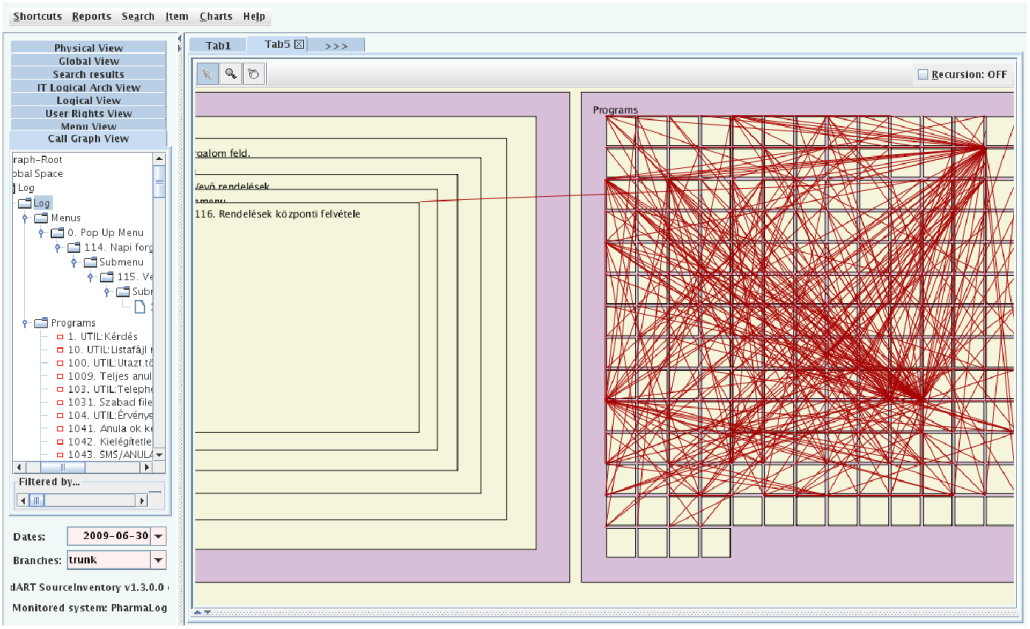
Metrika	Érték
Programok száma	2 761
Logikai sorok száma	305 064
Összes Task száma	14 501
Összes Adattábla száma	786

3.1. táblázat. Elemzett Magic rendszer főbb metrikái.

A módszereket sikeresen adaptáltuk, a folyamat közben viszont az alábbiakat figyeltük meg: (1) az adaptált minőségmutatókat óvatosan kell kezelni, a tipikusan használt méret és komplexitás metrikákat is máshogy értelmezik a fejlesztők; (2) a fejlesztői környezet olyan információt is eltárol az alkalmazásról, amit 3GL nyelvek



3.2. ábra. A Taskok eloszlása a logikai sorok száma alapján.



3.3. ábra. Menü elérésékkel bővített program hívások.

esetében csak nagyon nehezen, vagy egyáltalán nem lehetne kinyerni. Ilyen információ például a task-ok tábla-elérése, ami közvetlen lekérdezhető a fejlesztői környezet mentéseiből.

Az ipari partnerünknek köszönhetően az adaptált eszközöket valós, ipari környezetben tesztelhettük és validálhattuk. Mi több, első kézből kaphattunk visszajelzéseket tapasztalt Magic fejlesztőktől. A 3.1 táblázat a tesztrendszer fő metrikáit szemlélteti, a 3.2. és a 3.3. ábra pedig rendre a metrikák eloszlását, illetve egy architekturális nézetet szemléltetnek.

A kifejlesztett elemző csomag jó alapját adta további kutatásoknak is, egy tanulmányban [18] például a Magic alkalmazások layout-független automatikus UI tesztelésére dolgozunk ki egy módszert, kihasználva, hogy az alkalmazás grafikus felületéről is tárol a Magic adatokat (pl. ablakok és rajtuk lévő control-ok pozíciói).

3.1.1. Saját hozzájárulás

A szerző munkája meghatározó volt a kifejlesztett eszközök megtervezésében és implementálásában is. Bár a Magic nyelvi feldolgozóját a SZEGED Szoftver Zrt. munkatársai fejlesztették, a szerző tervezte a Magic sémát és implementálta az azt kezelő API-t. A szerző definiálta a Magic-re a metrikákat illetve a kinyert architekturális nézeteket. A kódolási szabálysértések és azok tesztelése a Magic fejlesztőkkel közösen történt. A szerző tervezte továbbá a Magic alkalmazások layout-független automatikus UI teszteléséért felelős alkalmazást, amit egy kapcsolódó tanulmányban mutatunk be [18]. Megjegyezzük, hogy az eredmények számos további kutató munkának adtak alapot. Az eredményekre támaszkodnak hallgatói szakdolgozatok, TDK munkák, valamint tudományos konferenciákon előadott munkák is [4, 5, 13]; mindemellett több, az Európai Unió támogatásával megvalósuló innovációs projekt elméleti alapját is adja [24, 27].

3.2. Új komplexitás metrikák definiálása és kiértékelése Magic rendszereken

A Magic alkalmazások belső szerkezetének leírása közben több, 3GL metrikák adaptálásával mérhető tulajdonságot is sikerült meghatározni (pl. méret alapú metrikák, csatolás metrikák, komplexitás). A legnagyobb kihívást a komplexitás metrika definiálása jelentette, ugyanis az első elemzési eredményeket megmutatva a fejlesztőknek, azt a visszajelzést kaptuk, hogy a komplexnek ítélt kódelemek szerintük nem komplexek. A metrikákat módosítottuk a fejlesztők visszajelzései alapján, és egy kísérletben összevetettük az összes kidolgozott metrikát a tapasztalt Magic fejlesztők véleményével.

Először a 3GL nyelvekből jól ismert komplexitás metrikákat adaptáltuk (McCabe és Halstead komplexitás), majd a fejlesztők visszajelzése alapján a McCabe komplexitást módosítottuk a 3.4. ábrán látható módon. A 3.5. ábra a kísérletben használt Magic programok egy rangsorolását mutatja a komplexitás mutatóik alapján. *EC* (Experiment Complexity) mutatja a fejlesztők átlagolt rangsorolását, $McCC_2$ a módosított ciklomatikus komplexitást, a *HPV* és *HE* metrikák pedig a Halstead komplexitás metrikák.

$$McCC(LU) = Number\ of\ decision\ points\ in\ LU + 1$$
$$WLUT(T) = \sum_{LU \in T} McCC(LU)$$
$$McCC_2(LU) = Number\ of\ decision\ points\ in\ LU + \sum_{TC \in LU} McCC_2(TC) + 1$$
$$McCC_2(T) = \sum_{LU \in T} McCC_2(LU)$$

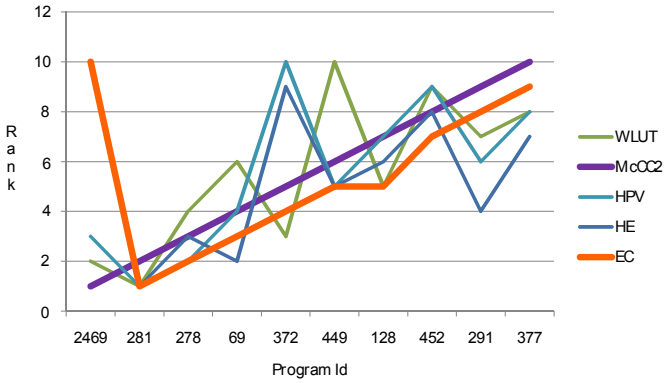
T: Task a Projektben

LU: a Task egy Logic Unit-ja

TC: LU-ból hívott Task

3.4. ábra. Logic Unit-ra adaptált ciklomatikus komplexitás (*McCC*), Task-ra adaptált ciklomatikus komplexitás (*WLUT*), módosított ciklomatikus komplexitás ($McCC_2$).

A fejlesztőkkel végzett kísérlet során úgy találtuk, hogy nem korrelál egymással a kezdeti adaptált McCabe komplexitás mutatónk és a fejlesztők rangsorolása, ugyanakkor erős a kapcsolat a módosított McCabe komplexitás és a fejlesztők, valamint a Halstead komplexitás és a fejlesztők rangsorolása között.



3.5. ábra. A fejlesztők rangsorolása (*EC* értékek) összehasonlítása a metrikák szerinti rangsorolással. (Task-ok ciklomatikus komplexitása (*WLUT*), módosított ciklomatikus komplexitás ($McCC_2$), Halstead komplexitások (*Program Volume*, *HPV*; *Effort to implement*, *HE*).

3.2.1. Saját hozzájárulás

A szerző munkája volt meghatározó a metrikák definiálásban és a fejlesztőkkel végzett kísérlet végrehajtásában. A metrikák implementálása és a végeredmények kiértékelése a kapcsolódó cikk társszerzőinek közös munkájaként történt [26]. A módosított komplexitás metrika definícióját fontos eredménynek tekintjük, hiszen hasonló, a fejlesztők komplexitás elképzelését megfelelően tükröző mutatót még nem dolgoztak ki Magic rendszerekre.

4. Biztonsági elemzés és optimalizálás

Ebben a fejezetben biztonsági elemzésre és optimalizálásra kidolgozott módszereket ismertetünk. Ezek a technikák általánosabbak a korábban bemutatottaknál abban az értelemben, hogy nem függnek egy adatbázis-központú architektúrától.

Először egy statikus elemzési módszert ismertetünk olyan alkalmazásokra, amik külső forrásokból kapott (pl. felhasználó, I/O műveletek) adattal dolgoznak. Ezt követően lokális refaktoring algoritmusok hatékonyságát vizsgáljuk C, C++ rendszereken. Az algoritmusokat a GCC fordító különböző, belső reprezentációs szintjein implementáltuk, és azt mérjük, melyik szinten, milyen százalékos kódméret csökkenés érhető el a segítségükkel.

4.1. Felhasználói input okozta biztonsági hibák felderítése

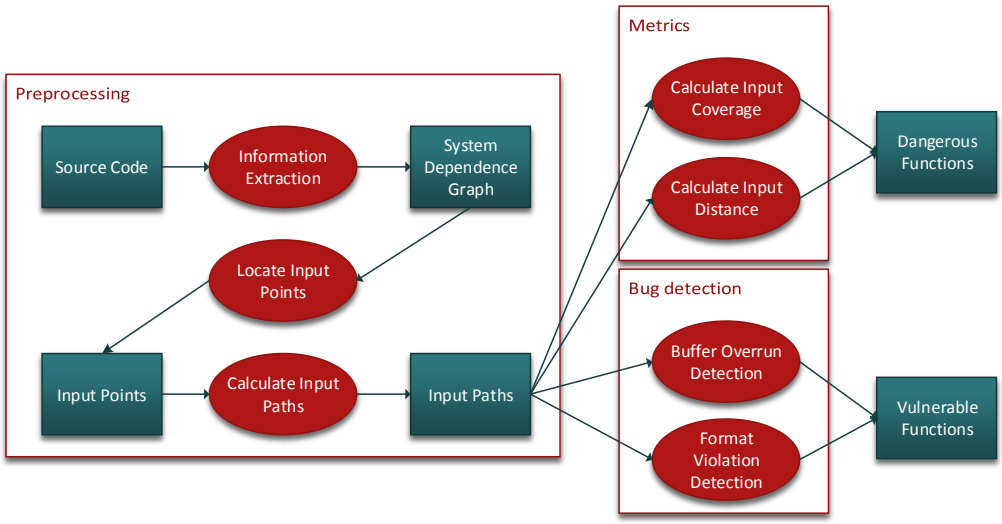
A bemutatott módszerben [21] a forráskódnak arra a részére korlátozzuk az elemzést, ami a felhasználói inputtól függ. Ez az a kódrészlet, ami feldolgozza a felhasználótól kapott adatot, és ami ha hibát tartalmaz, akkor az egy támadó által könnyen kihasználható. Az út, amit az inputként kapott adat bejár, adatfolyam elemzéssel nyomon követhető, így az érintett kódrészlet is meghatározható. Hibák természetesen bárhol előfordulhatnak a kódban, de amik ebben a kódrészletben helyezkednek el, azok különösen veszélyes biztonsági hibákat rejthetnek.

A bemutatott módszer fő lépései az alábbiak:

1. Megkeressük azokat a helyeket a forráskódban, ahol I/O műveletekkel adatot olvasunk be. Ezek a helyek lesznek az ún. *input pontok*.

2. Meghatározzuk azokat a kódrészleteket, amik az input pontoktól függnek.

3. Metrikák segítségével meghatározzuk az input pontoktól függő, veszélyes metódusokat.



4.1. ábra. A módszer főbb lépéseinek áttekintése.

Név	Előfordulás
read()	55
fread()	12
fgets()	10
gg_read()	9
gethostname()	6
getpwuid()	2
fscanf()	1
getenv()	1
getpass()	1
char *argv[]	1

(a)

Függvény	Sorok száma	Lefedettség (%)
yahoo_roomlist_destroy	12	83.33
aim_info_free	13	84.62
s5_sendconnect	22	77.27
purple_ntlm_gen_type1	35	77.14
gtk_imhtml_is_tag	91	76.92
jabber_buddy_resource_free	25	72.00
peer_oft_checksum_destroy	8	75.00
qq_get_conn_info	12	75.00
_copy_field	8	75.00
qq_group_free	8	75.00

(b)

4.1. táblázat. A Pidginben előforduló (a) input műveletek, (b) valamint a tíz legnagyobb input lefedettséggel rendelkező függvény.

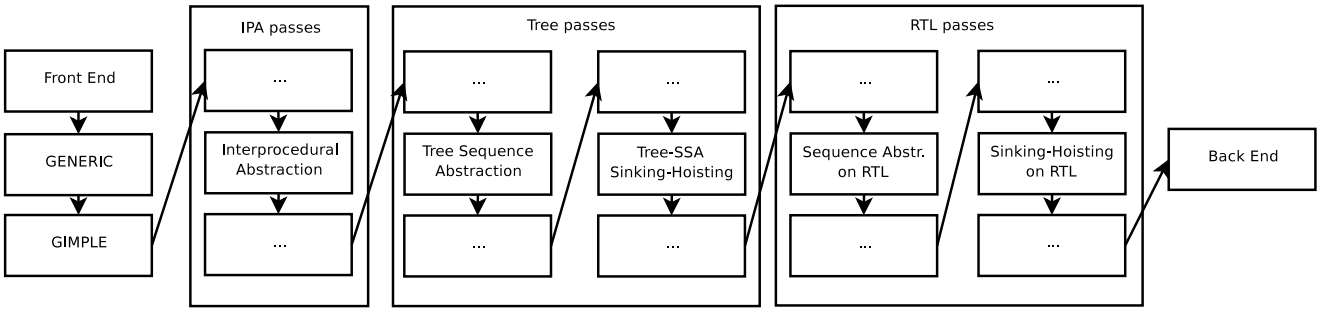
4. Automatikus hibakereső algoritmusokkal hibás kódrészleteket keresünk.

A módszert nyílt forráskódú rendszereken teszteljük, és bemutatunk egy esettanulmányt is, amiben a közismert Pidgin chat klienst elemezzük. A módszerrel Pidginben is és a többi elemzett rendszerben is valós hibákat találtunk. A bemutatott módszer abban az értelemben is új, hogy a konkrét hibafelderítés mellett metrikákat definiál, amik rossz tervezésre vagy hibákra különösen érzékeny függvényekre mutatnak rá. A Pidgin tanulmány jól demonstrálja a bemutatott módszer hatékonyságát egy közepes méretűnek mondható rendszeren, ami 7173 függvényt és 229825 kódsort tartalmat. Néhány mért adatot mutat be a 4.1. táblázat. A mérések eredménye azt mutatja, hogy a rendszernek alig több, mint 10%-a érintett a felhasználói inputban.

4.1.1. Saját hozzájárulás

A C és C++ forráskód elemzéséhez a GrammTech Inc. CodeSurfer eszközét használtuk. Az eszközhöz a szerző implementálta azt a plugint, amivel a bemutatott algoritmust teszteltük. A tesztelést is és az eredmények kiértékelését is a szerző végezte el. A bemutatott eredményeket fontos eredményeknek tekintjük a statikus forráskód-elemzés biztonsággal foglalkozó területén, ahogy a kapcsolódó cikkre [21] több külső hivatkozás is található.

Ebben a részben új optimalizációs algoritmusokat mutatunk be, amiket a GCC fordító különböző belső reprezentációs szintjein implementáltunk. Az algoritmusok úgynevezett kód faktoring algoritmusok, optimalizációs technológiáknak egy olyan családja, amiket kódméret csökkentésre dolgoztak ki. A fejlesztők már korábban felismerték a lehetőségeket ezekben az algoritmusokban, ahogyan már más eszközökben implementálták is őket (pl. a The Squeeze Project'[1] az egyik első ilyen projekt volt).



4.2. ábra. Az implementált algoritmusok egy áttekintése.

A bemutatott algoritmusokat a GCC Tree-SSA és RTL szintjein is implementáljuk, az ún. szekvenciális absztrakcióra egy interprecodedurális változatot is bemutatunk. A 4.2. ábra ad egy áttekintést az algoritmusok sorrendjéről az egyes szinteken.

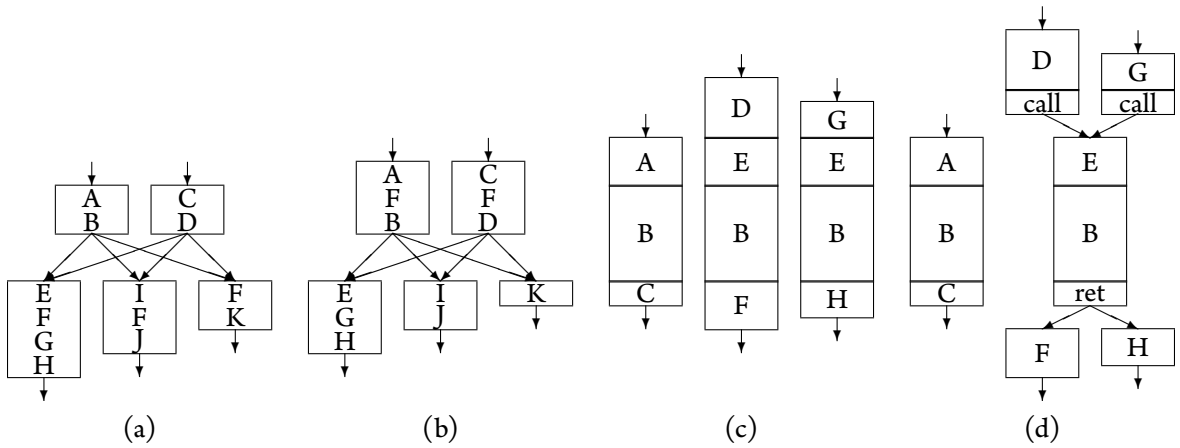
A *lokális faktoring* (*local factoring* vagy *code hoisting/sinking*) azon az egyszerű ötleten alapszik, hogy basic blockokban, amiknek a futását közös basic blockok előzik meg, vagy éppen követik, gyakran előfordul, hogy azonos utasítások találhatóak meg, amiket egyszerűen át lehetne mozgatni a közös szülőkbe vagy gyerekekbe.

Tekintsünk példának egy `if` utasítást, aminek a `then` és `else` ága is ugyanazokkal az utasításokkal kezdődik. Ha a feltételvizsgálattól nem függnek, akkor ezeket az utasításokat könnyedén az `if` elé mozgathatjuk (ezt nevezik *code hoisting*-nak), amivel fölösleges kód duplikációt szüntethetünk meg (4.3/a-b. ábra). Ez az alap ötlet kiterjeszthető egyéb, összetettebb esetekre is, mint például a `switch` utasítás vagy a `goto` utasítás okozta elágazások a vezérlési folyamban. Sőt, az utasításokat nem csak az `if` elé mozgathatjuk a `then` vagy `else` ágakból, hanem az `if` mögé is. Ezt nevezik *code sinking*-nek, amit csak akkor lehet megtenni, ha a mozgatott utasításoktól nem függnek egyéb utasítások az eredeti blockon belül.

Szekvenciális kiszervezés (*sequence abstraction*) a lokális faktoringgal szemben egy bemenetű és egy kimenetű (*single-entry single-exit*, *SESE*) kódrészletekkel foglalkozik, nem önálló utasításokkal. A technika lényege, hogy egymás utáni, azonos utasítás sorozatokat találjunk, amiket eljárásokká lehet kiszervezni (4.3/c-d. ábra). Egy új eljárás létrehozása után a kiszervezett kódrészletet egyszerűen az eljáráshívással helyettesítjük. A módszer hasonlóan alkalmazható (*multiple-entry single-exit*, MESE) kódrészletek esetében is.

Az algoritmusok implementációjának helyességét és a kódméret csökkentésének hatékonyságát a GCC hivatalos, kódméret mérésre kialakított tesztkörnyezetén mértük (*Code-Size Benchmark Environment*, CSiBE). A méréseket több architektúrára is elvégeztük, amik közül az ARM architektúrán a legmagasabb kódcsökkenés 61.53% volt, az átlagos pedig 2.58%-os az egyszerű -Os' kapcsolóval összevetve, ami jelentős eredménynek tekinthető. Néhány kiemelt mérési eredményt szemléltet a 4.2. táblázat.

[1] http://www.cs.arizona.edu/projects/squeeze/



4.3. ábra. Basic block-ok közös szülőkkel és gyerekekkel a lokális faktoring (a) előtt és (b) után. Különböző hosszúságú, szekvenciális absztrakcióval kiszervezhető kódrészletek (c) a leghosszabb kódrészlet kiemelésével (d). Azonos betűk, azonos utasítássorozatokat jelölnek.

Kapcsolók	i686-elf			arm-elf		
	méret (byte)	avg (%)	max (%)	méret (byte)	avg (%)	max (%)
-Os	2900177			3636462		
-Os -ftree-lfact -frtl-lfact	2892432	0.27	6.13	3627070	0.26	10.29
-Os -frtl-lfact	2894531	0.19	4.31	3632454	0.11	4.35
-Os -ftree-lfact	2897382	0.10	5.75	3630378	0.17	10.34
-Os -ftree-seqabstr -frtl-seqabstr	2855823	1.53	36.81	3580846	1.53	56.92
-Os -frtl-seqabstr	2856816	1.50	30.67	3599862	1.01	42.45
-Os -ftree-seqabstr	2888833	0.39	30.60	3610002	0.73	44.72
-Os -fipa-procabstr	2886632	0.47	56.32	3599042	1.03	59.29
Összes	**2838348**	**2.13**	**57.05**	**3542506**	**2.58**	**61.53**

4.2. táblázat. Átlagos és maximális kódméret csökkenés adatok i686-elf és arm-elf rendszerekre. *Méret* a binárisok összmérete byteban megadva; *avg* a számított átlagos méret csökkenés a '-Os'-hez viszonyítva; *max* pedig a legnagyobb kódméret csökkenés, amit egy objektumon elértünk százalékban kifejezve.

4.2.1. Saját hozzájárulás

Az algoritmusok a korábban publikált [9] módszer alapján kerültek megtervezésre. A sinking-hoisting és a sequence abstraction algoritmusok implementálása nagyrészt a szerző munkája volt, közösen Lóki Gáborral, a [22] publikáció második szerzőjével. A bemutatott mérések elvégzése és az eredmények kiértékelése a szerző saját hozzájárulásának eredménye. A szerző hozzájárulása továbbá az a kezdeti munka, aminek során a Columbus ASG-je kerül átalakításra a GCC belső reprezentációjára [19].

5. Összefoglalás

Jelen munka különböző technikákat mutat be adat-intenzív rendszerek elemzésére és automatikus transzformációk végrehajtására. Ebben a fejezetben a korábban feltett kutatási kérdésekre adunk válaszokat az eredményeinket összefoglalva.

5.1. Eredmények összefoglalása

Összességében, az eredmények azt mutatják, hogy statikus kódelemző módszerekkel hatékonyan lehet támogatni az adat-intenzív rendszerek fejlesztési folyamatait. Egy alkalmazás legjobb dokumentációja a forráskód, a forráskódot elemezve ezért olyan implicit információt nyerhetünk a rendszerről, ami más módszerek számára rejtett maradhat. Megmutatjuk, hogy az adatelérések (pl. beágyazott SQL utasításokon keresztül) ilyen rejtett függőségeket hordoznak, ugyanakkor jó forrásai architekturális kapcsolatoknak. A bemutatott módszerek alkalmazhatóak Magic-re is, mint egy speciális negyedik generációs programozási nyelvre. Mindemellett, egy statikus elemzési módszert mutatunk be felhasználó input okozta biztonsági hibák felderítésére, és optimalizációs eljárásokat ismertetünk a kódméret csökkentésére.

Fontosnak tartjuk megjegyezni, hogy a bemutatott eredmények általában valós, ipari motivációs igényt elégítenek ki, aminek eredményeként kidolgozott módszerek tesztelését is ipari környezetben végezhettük el. A kutatási munkák eredményeire ezért a külső hivatkozások mellett Európai Uniós támogatással megvalósuló, innovációs projektek is támaszkodnak. Emellett a Magic rendszereken elért eredmények több szakdolgozatnak és TDK munkának az alapját is adták, amelyek nemzetközi konferenciákon is bemutatásra kerültek.

1) Lehetséges-e automatikus forráskód elemzési módszerekkel, adatelréseket vizsgálva, információt kinyerni, ami segíthet egy adat-intenzív rendszer architektúrájának feltérképezésében? Bemutattunk egy új módszert adat-intenzív rendszerek architekturális kapcsolatainak kinyerésére (*CRUD* kapcsolatok), ami az adateléréseket vizsgálja a beágyazott SQL utasítások elemzésével. Az ötlet alapja, hogy a program alkalmazás oldalát és az adatbázist együttesen elemezzük, felderítve ezzel olyan függőségeket, amik adatbázis használat miatt jöhetnek létre. Egy nagyméretű, pénzügyi rendszert vizsgálunk, amit ForrásSQL nyelven fejlesztettek Transact SQL és MS SQL utasításokat beágyazva a kódba. A kinyert kapcsolatokat a Static Execute After/Before kapcsolatokkal vetjük össze, aminek az eredményeként azt tapasztaljuk, hogy a *CRUD* kapcsolatok olyan függésekre mutatnak rá, amiket más módszerek nem ismernek fel. Ezt a módszert használjuk ezért egy későbbi tanulmány során is, ahol egy nagyméretű Oracle PL/SQL rendszer architektúráját térképezzük fel.

A tanulmányokból kiderül, hogy automatikus elemzési módszerekkel vizsgálva az adateléréseket olyan hasznos információt nyerhetünk egy rendszerről, amit más módszerekkel nem tudnánk felderíteni. A technika alkalmazása ezért javasolt lehet olyan elemzéseknél mint pl. a hatásanalízis, architektúra visszatervezés vagy minőségbiztosítás.

2) Adaptálható-e egy harmadik generációs nyelvekhez kifejlesztett automatikus elemzési módszer egy negyedik generációs nyelvre, mint amilyen a Magic? Amennyiben igen, úgy lehetséges-e statikus kódelemzéssel támogatni egy Magic alkalmazás újabb verzióra történő migrálását? A disszertációban bemutatunk egy újszerű módszert Magic alkalmazások statikus elemzésére. Ebben a módszerben az alkalmazásfejlesztő környezet mentését tekintjük az alkalmazás 'forráskódjának'. Ez a mentési állomány ugyan nem tekinthető a hagyományos értelemben vett forráskódnak, mégis minden fontos információt tartalmaz az alkalmazás felépítéséről. Erre támaszkodva, egy teljes elemző eszközcsomagot fejlesztünk a Columbus módszertanból kiindulva, amit célzottan C, C++ és Java nyelven fejlesztett alkalmazások visszatervezésére terveztek. Az ipari partnerünk segítségével megmutatjuk, hogy a kifejlesztett eszközcsomag jól használható Magic alkalmazások visszatervezéséhez.

Megmutatjuk, hogy az ismert komplexitás metrikák közül a McCabe és a Halstead komplexitás metrikák sem tükrözik a fejlesztők komplexitás elképzelését, ezért Magic-re egy új komplexitás metrikát javaslunk.

Azt is megmutatjuk, hogy statikus kódelemzéssel felfedezhetőek olyan kapcsolatok az alkalmazásban (pl. CRUD relációk, táblák közötti külső kulcs kapcsolatok), amik jelentősen segíthetik egy Magic alkalmazás migrálását egy korábbi verzióról egy újabb verzióra.

3) Hatékonyan használhatóak-e a vezérlési és adatfolyam elemzések a felhasználói input okozta biztonsági hibák felderítéséhez? Bemutatunk egy olyan elemzési módszert, ami a vezérlési és adatfolyam elemzéseket felhasználói input okozta biztonsági hibák felderítéséhez használja, C nyelven íródott alkalmazásokban. A módszer a különböző I/O műveletekből származó adatot követi nyomon az adatfolyamban, és jelez, ha a vezérlés olyan, hibára érzékeny ponthoz jut, ahol nem lett leellenőrizve korábban az külső forrásból érkező adat. A módszert GrammaTech CodeSurfer pluginként implementáljuk és nyílt forráskódú rendszereken teszteljük. A közel 200.000 kódsoros Pidginben és cyrus-imapd-ben is rámutatunk tényleges hibákra a segítségével.

4) Milyen mértékben lehet csökkenteni kód faktoring algoritmusok segítségével egy fordító által előállított binárisok méretét? Kód factoring algoritmusok családjába tartozó lokális faktoring és sequence abstraction algoritmusokat implementálunk a GCC különböző optimalizációs szintjein, hogy azt vizsgáljuk milyen kódméret csökkenés érhető el az algoritmusok segítségével. Az algoritmusokat a GCC hivatalos, kódméret mérésre kialakított tesztkörnyezetén teszteljük (Code-Size Benchmark Environment, CSiBE). A méréseket több architektúrára is elvégeztük, amik közül az ARM architektúrán a legmagasabb kódcsökkenés 61.53% volt, az átlagos pedig 2.58%-os az egyszerű -Os' kapcsolóval összevetve, ami jelentős mértékű csökkentésnek tekinthető.

Köszönetnyilvánítás

Mindenek előtt szeretném megköszönni témavezetőmnek, Dr. Gyimóthy Tibornak, az érdekes kutatási témákat és célokat, a biztos hátteret és a vezetését éveken keresztül. Külön köszönettel tartozok szerzőtársamnak, Dr. Ferenc Rudolfnak, aki mint mentorom irányította a munkámat és segített az évek alatt. Köszönöm továbbá David P. Curleynek a munka nyelvi helyességének ellenőrzését és javítását. Köszönettel tartozom még kollégáimnak és társszerzőimnek, Spiros Mancoridisnak, Lóki Gábornak, Dr. Beszédes Árpádnak, Gergely Tamásnak, Vidács Lászlónak, Bakota Tibornak, Pántos Jánosnak, Kakuja-Tóth Gabriellának, Fischer Ferencnek, Hegedűs Péternek, Jász Juditnak, Sógor Zoltánnak, Fülöp Lajos Jenőnek, Siket Istvánnak, Siket Péternek, Kiss Ákosnak, Havasi Ferencnek, Fritsi Dánielnek, Novák Gábornak és Dévai Richárdnak. Köszönet továbbá a tanszék minden dolgozójának az évek során nyújtott támogatásért.

A disszertáció egy fontos része foglalkozik a Magic programozási nyelvvel. Ez a munka nem jöhetett volna létre a SZEGED Szoftver Zrt. együttműködése nélkül. Külön köszönöm ezért a cég munkatársainak az együttműködését, különösen Kovács Istvánnak, Kocsis Ferencnek és Smohai Ferencnek.

Végül, de nem utolsó sorban, köszönöm családomnak, szüleimnek és testvéreimnek, hogy mindenben támogattak és biztattak a munkám során.

Nagy Csaba, December 2013.

Hivatkozások

Hivatkozások

[1] Huib van den Brink, Rob van der Leek, and Joost Visser. Quality assessment for embedded SQL. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 163–170. IEEE Computer Society, 2007.

[2] Anthony Cleve. *Program Analysis and Transformation for Data-Intensive System Evolution*. PhD thesis, University of Namur, October 2009.

[3] Anthony Cleve, Tom Mens, and Jean-Luc Hainaut. Data-intensive system evolution. *IEEE Computer*, 43 (8):110–112, August 2010.

[4] Richárd Dévai, Judit Jász, Csaba Nagy, and Rudolf Ferenc. Designing and implementing control flow graph for magic 4th generation language. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools (SPLST 2013)*, pages 200–214, Szeged, Hungary, August 26-27 2013.

[5] Dániel Fritsi, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 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 (SPLST 2011)*, pages 248–259, Tallinn, Estonia, Oct 4-7 2011.

[6] Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. StaticExecute After/Before as a Replacement of Traditional Software Dependencies. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM 2008)*, pages 137–146. IEEE Computer Society, 2008.

[7] R. T. Kouzes, G. A. Anderson, S. T. Elbert, I Gorton, and D. K. Gracio. The changing paradigm of data-intensive computing. *IEEE Computer*, 42(1):26–34, January 2009.

[8] Kaiping Liu, Hee Beng Kuan Tan, and Xu Chen. Extraction of attribute dependency graph from database applications. In *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, pages 138–145. IEEE Computer Society, 2011.

[9] Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes. Code factoring in GCC. In *Proceedings of the 2004 GCC Developers' Summit*, pages 79–84, June 2004.

[10] C.A. Mattmann, D.J. Crichton, J.S. Hughes, S.C. Kelly, and M. Paul. Software architecture for large-scale, distributed, data-intensive systems. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 255–264, 2004.

[11] Chris Mattmann and Paul Ramirez. A comparison and evaluation of architecture recovery in data-intensive systems using focus. Technical report, Computer Science Department, University of Southern California, 2004.

[12] Chris A. Mattmann, Daniel J. Crichton, Andrew F. Hart, Cameron Goodale, J.Steven Hughes, Sean Kelly, Luca Cinquini, ThomasH. Painter, Joseph Lazio, Duane Waliser, Nenad Medvidovic, Jinwon Kim, and Peter Lean. Architecting data-intensive software systems. In *Handbook of Data Intensive Computing*, pages 25–57. Springer Science+Business Media, 2011.

[13] Gábor Novák, Csaba Nagy, and Rudolf Ferenc. A regression test selection technique for Magic systems. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools (SPLST 2013)*, pages 76–89, Szeged, Hungary, August 26-27 2013.

[14] Z. Pawlak. Information systems theoretical foundations. *Information Systems*, 6(3):205 – 218, 1981.

[15] R. Kelly Rainer and Casey G. Cegielski. *Introduction to Information Systems: Enabling and Transforming Business*. John Wiley & Sons, Inc., 4 edition, January 11 2012.

[16] H. Rottgering. Lofar, a new low frequency radio telescope. *New Astronomy Reviews*, 47(4-5, High-redshift radio galaxies - past, present and future):405–409, Septepmber 2003.

[17] A. Van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC 1998)*, page 90. IEEE Computer Society, 1998.

Felhasznált publikációk

[18] Dániel Fritsi, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. A methodology and framework for automatic layout independent GUI testing of applications developed in Magic xpa. In *Proceedings of the 13th International Conference on Computational Science and Its Applications - ICCSA 2013 - Part II*, pages 513–528, Ho Chi Minh City, Vietnam, June 24-27 2013. Springer.

[19] Csaba Nagy. Extension of GCC with a fully manageable reverse engineering front end. In *Proceedings of the 7th International Conference on Applied Informatics (ICAI 2007)*, January 28-31 2007. Eger, Hungary.

[20] Csaba Nagy. Static analysis of data-intensive applications. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*. IEEE Computer Society, March 5-8 2013. Genova, Italy.

[21] Csaba Nagy and Spiros Mancoridis. Static security analysis based on input-related software faults. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*, pages 37–46, Fraunhofer IESE, Kaiserslautern, Germany, March 24-27 2009. IEEE Computer Society.

[22] Csaba Nagy, Gábor Lóki, Árpád Beszédes, and Tibor Gyimóthy. Code factoring in GCC on different intermediate languages. *ANNALES UNIVERSITATIS SCIENTIARUM BUDAPESTINENSIS DE ROLANDO EOTVOS NOMINATAE Sectio Computatorica - TOMUS XXX*, pages 79–96, 2009.

[23] Csaba Nagy, János Pántos, Tamás Gergely, and Árpád Beszédes. Towards a safe method for computing dependencies in database-intensive systems. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pages 166–175, Madrid, Spain, March 15-18 2010. IEEE Computer Society.

[24] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. MAGISTER: Quality assurance of Magic applications for software developers and end users. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–6, Timisoara, Romania, Sept 2010.

[25] Csaba Nagy, Rudolf Ferenc, and Tibor Bakota. A true story of refactoring a large Oracle PL/SQL banking system. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011)*, Szeged, Hungary, Sept 5-9 2011.

[26] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Complexity measures in 4GL environment. In *Proceedings of the 2011 International Conference on Computational Science and Its Applications - Volume Part V*, ICCSA'11, pages 293–309, Santander, Spain, June 20-23 2011. Springer-Verlag.

[27] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Solutions for reverse engineering 4GL applications, recovering the design of a logistical wholesale system. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pages 343 –346, Oldenburg, Germany, March 1-4 2011. IEEE Computer Society.